TECHNISCHE UNIVERSITÄT BERLIN
Fakultät IV – Elektrotechnik und Informatik
Fachgebiet Intelligente Netze und Management verteilter Systeme
Prof. Anja Feldmann, Ph. D.
Harald Schiöberg, Bernhard Ager, Vlad Manilici

10th Assignment Protocol Design WS 08/09

Question 1: (30 points) **A Simple Chat System**

The most simple case of a client-server based chat system consists of a set of client nodes {a, b, c, ...} connected to a single server node S (see *Figure 1*). All clients have to connect to the server to be able to participate on the chat system. A message that is sent by one of the clients using the client's connection to the server will be forwarded by S to all *other* client nodes (flooding) thus allowing everybody to receive and read the message.

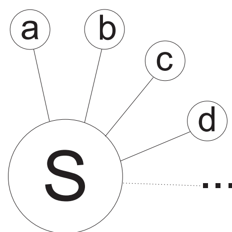

Figure 1: A rudimentary chat system with central server

The goal of this exercise is the implementation of such a simple chat system, using *TCP* as a transport protocol and to understand the functionality of *IO::Select*. To this end, write a program/script that is able to act as both client and server. As a client, it has to be able to connect to a specified server, read messages from keyboard, and send them to the server. Furthermore, it has to display incoming messages on the screen.

As server it has to listen for incoming connection attempts, accept incoming connections, and flood all incoming messages.

Deliverables:

- Your program/script (as source and, if written in C, C+, ... , as binary)
- A short description how to use the software and what doesn't quite work

Implementation of a Peer-to-Peer System

With this work sheet we start with the step-by-step implementation of a simple peer-to-peer system, that will also work as the base for further excercises. We will start with a very simple base system and will extend the system during the next weeks in small steps.

Protocol Specification

Our *P2P* system uses a simple text based protocol and shows many typical syntax elements of real protocols in the Internet. The system consists of two main components. The first component is the mechanism to simultaneously handle multiple *TCP* connections (similar to Question 1). The second component is a simple protocol for setting up *P2P* sessions, searching and downloading data, and to manage the *P2P* overlay topology.

The protocol is specified ass follows:

- The *P2P* application protocol uses *TCP* as transport protocol
- In version `0.1` every message, request or reply, consists of exactly one line of text.
- Every line is terminated by `\r\n`. This allows for easy manual simulation of protocol sessions using the client from *Question 1*.
- All requests messages end with a protcol specification, e.g., `P2P/0.1`, where `0.1` is the protocol version.
- Replies always begin with the protocol identification string and a three digit numeric response code.
- Replies to requests with a request ID always contain the same request ID in order to be able to match requests and replies. Every node uses a request ID only once, so that the tuple of node ID and request ID is unique.
- Every node is allowed to *actively* initiate up to 4 sessions to neighbour nodes. It however has to accept all incoming connections and session requests.
- A node that has a direct protocol session with another node is called *neighbour*. As is common in *P2P* systems, it is irrelevant who initiated the session in the first place.
- The maximum distance a message is allowed to travel through the *P2P* overlay is 3 hops. This is controlled using a TTL counter that is decreased upon receiving and the message is to be forwarded if the decreased TTL counter is greater than 0. The message is discarded otherwise.
- Protocol messages are either requests or replies. There are three kinds of requests/replies:
  - Messages used during session initialization. Such messages are never forwarded. Therefore such messages do not have a TTL counter.
  - Messages that are destined for a specific node. Such messages always contain a target node ID specified with `FOR` and a originator ID specified with `FROM`.
    Messages of that type are (for now) forwarded by flooding with regard to TTL. Later on we will implement a routing mechanism to forward such messages more efficiently.
  - Nodes are identified by the host name and the listening UDP port number.
  - Messages that are restricted to a maximum distance from the originating node, e.g., `PING`. Such messages always contain an originator specified with `FROM <Node-ID>` but no receiver specification. On the other hand, they always contain a request ID that, together with the node ID must be globally unique. The request ID and the node ID can be used to identify message duplicates and are used to supress multiple forwarding of the same request or response.

Below you find a more formal specification of the protocol messages in a BNF-like notation:

```
node-id ::=  <hostname>":"<local port>
_ ::=  <blank>
protoname ::=  "P2P"
version ::= "0.1"
proto ::= protoname"/"version
crlf ::= "\r\n"
return-code ::= [1245][0-9][0-9]

key ::= KEY _ <filename>
msgid ::= MESSAGE-ID _ <number>
size ::= SIZE _ <number>
ttl ::= TTL _ <number>

sender-addr ::= FROM _ node-id
```

```
target-addr ::= FOR _ node-id
addresspair ::= sender-addr _ target-addr

neighbour-list ::= node-id | node-id _ neighbour-list
setup-request  ::= HELLO _ NODE-ID _ node-id _ proto crlf |
               ::= HELLO _ NODE-ID _ node-id _ NEIGHBOURS _ neighbour_list _ proto crlf

search ::= SEARCH _ sender-addr _ key _ msgid _ ttl _ proto crlf
get ::= GET _ addresspair _ key _ msgid _ ttl _ proto crlf
put ::= PUT _ addresspair _ key _ size _ msgid _ ttl _ proto crlf
ping ::= PING _ sender-addr _ msgid _ ttl _ proto crlf

reply-head ::= proto
reply-tail ::= target-addr _ sender-addr _ msgid _ ttl

setup-reply ::= reply-head _ 200 _ node-id crlf |
                reply-head _ 200 _ node-id _ NEIGHBOURS _ neighbour-list crlf |
                reply-head _ 400 _ HANDSHAKE _ FAILURE crlf
                reply-head _ 402 _ ALREADY _ CONNECTED crlf

disconnect-request ::= DISCONNECT _ proto crlf
disconnect-reply   ::= reply-head _ 210 _ GOODBYE crlf

ping-reply ::= reply-head _ 230 _ PONG _ reply-tail crlf
search-reply ::= reply-head _ 220 _ FOUND _ reply-tail _ key crlf
get-reply ::= not-implemented
put-reply ::= not-implemented

not-implemented ::= reply-head _ 510 _ NOT _ IMPLEMENTED _ reply-tail _ key crlf
```

Description of Protocol Messages

Below we will provide a short a short description of how to send messages and how to react when a message is received.

Protocol Initialization Whenever a node A actively connects to another node, it has to register itself with its peer node. This is done by a request that might look like this:

`HELLO NODE-ID viper:2000 NEIGHBOURS viper:3000 boa:2000 P2P/0.1`

This message states that node A runs on the host viper an accepts incoming connections on port 2000. Moreover, it states that the sending node has active sessions with its neighbours `viper:3000` und `boa:2000`.

If node A does not yet have any active sessions, the message would look like this:

`HELLO NODE-ID viper:2000 P2P/0.1`

The node that accepts an incoming *TCP* connection will acknwoledge the sessions initialization request by answering like this:

`P2P/0.1 200 NODE-ID boa:3000 NEIGHBOURS boa:4000`

or,

`P2P/0.1 200 NODE-ID boa:3000`

This concludes the session setup phase und only now the nodes are allowed to send one of the other messages over the session connection. If other messages are received before the session handshake has finished, a node has to react by sending an error message like this:

`P2P/0.1 400 HANDSHAKE FAILURE`

After sending this error messages, the node has to close the *TCP* connection immediately.

It is also forbidden to have multiple sessions between two nodes. If a node makes the attempt to setup a session to a node even if there is already a session between these two nodes, a `HELLO` message has to be answered with an error message like this:

```
P2P/0.1 402 ALREADY CONNECTED
```

The session setup has to aborted after sending such an error message.

Messages used during session setup are *never* forwarded to any other nodes.

Session Teardown In the case a node wants to end a session, it sends a message to its peer that looks like this:

```
DISCONNECT P2P/0.1
```

The peer node will answer with

```
P2P/0.1 210 GOODBYE
```

and now both nodes can close the *TCP* connection.

Messages used during session teardown are *never* forwarded to any other nodes.

Search Requests A search request for a file named `readme.txt` might look like this:

```
SEARCH FROM viper:2000 KEY readme.txt MESSAGE-ID 10 TTL 3 P2P/0.1
```

Every node that receives such a message has to decrease the TTL counter by one and, if the counter is still greater than 0, has to forward the message using the decreased TTL counter by flooding.

Moreover, if the node has access to a file named `readme.txt`, it can *in addition* send a found reply that might look like this:

```
P2P/0.1 220 FOUND FOR viper:2000 FROM boa:3000 MESSAGE-ID 10 TTL 3 KEY readme.txt
```

The key has to be copied from the search request and a new message ID has to be used.

If a node receives such a found message, it has to forward it with regard to the TTL counter, *except* the node is the designated receiver of the message (in the example above: `viper:2000`).

Download and Upload Requests Such requests might look like this

```
GET FROM viper:2000 FOR boa:3000 KEY readme.txt MESSAGE-ID 11 TTL 3 P2P/0.1
```

or this

```
PUT FROM viper:2000 FOR boa:3000 KEY readme.txt SIZE 428 MESSAGE-ID 12 P2P/0.1
```

If a node receives such a found message, it has to forward it with regard to the TTL counter, *except* the node is the designated receiver of the message. In this case, the node, *at this time*, has to answer with

```
P2P/0.1 510 NOT IMPLEMENTED FOR viper:2000 FROM boa:3000 MESSAGE-ID 11 TTL 3 KEY readme.txt
```

Ping Requests Ping messages request an acknowledgement from all nodes that are not farther away than a maximum distance. This acknowledgement is called a *pong*.
An example might look like this:

```
PING FROM viper:2000 MESSAGE-ID 24 TTL 3 P2P/0.1
```

If a node receives such a ping message, it has to forward it with regard to the TTL counter.

Moreover, it has to answer the originator of the message with a reply that looks like this:

```
P2P/0.1 230 PONG FOR viper:2000 FROM boa:3000 MESSAGE-ID 24 TTL 3
```

If a node receives a pong message, it has to forward it with regard to the TTL counter, *except* the node is the designated receiver of the message.

Tasks

Question 2: (70 points) We start with the implementation of our *P2P* system by seting up and tearing down multiple connections per node using *TCP*. All other parts of the *P2P* protocol will be required in later assignments.

For now, the task is to implement a node that is able to setup *TCP* connections to other nodes and also to accept incoming *TCP* connections (see also Question 1).

Moreover, the parts of the protocol that are responsible for session setup and teardown are to be implemented. This latter part consists of the `HELLO` and the `DISCONNECT` handshake (see specification and examples above). The `HELLO` handshake has to consider the management of neighbours (including adding and removing neighbour information and detection of multiple sessions).

For this assigenment, the reporting of neighbours during the `HELLO` handshake (`NEIGHBOURS` part of the messages) can be omitted. This part will be implemented later on.

In order to be able to control the node software, it has to accept simple commands from the keyboard. ***Examples*** for such commands might be:

connect <host> <port> This command could instruct the node software to establish a *TCP* connection to the given node and automatially setup a protocol session.

disconnect <node-id> Instructs the node software to tear down the protocol session to the specified node and close the corresponding *TCP* connection.

list connections Instructs the software to display all protocol sessions the node holds, together with the node ID reported during the session setup. This is very helpful for debugging purposes and to find peer IDs for disconnecting.

Please submit the source code of your program.

Due Date: Wednesday, 21.01.2009 at 23:59 s.t.