Advanced Networks — Laboratory 8

Juliusz Chroboczek

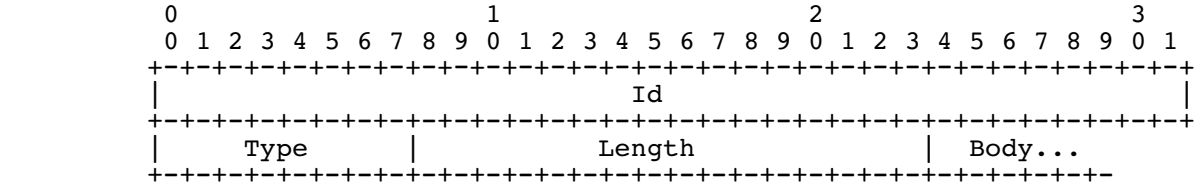
8 April 2025

The goal of this practical is to implement a hybrid chat protocol. The client-server part of the protocol consists of the following requests:

- `GET /peers/?count=n`: returns information on up to *n* peers;
- `PUT /peers/`*`nickname`*`?token=`*`token`*: updates peer information;
- `DELETE /peers/`*`nickname`*`?token=`*`token`*: deletes peer information.

The `token` parameter is used to avoid nickname hijacking (see below), and may be omitted.

The peer-to-peer part of the protocol is based on UDP, and all datagrams have the following syntax:

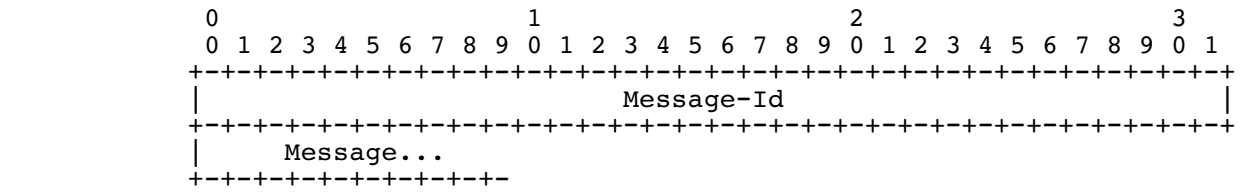


The field *Id* contains an identifier that is used for matching requests to replies. It is chosen arbitrarily by the client, and copied from the request into the response by the server.

The field *Type* contains the type of the message; if it is between 0 and 127, then the message is a request, otherwise it is a reply. The field *Length* contains the length of the body in bytes. The field *Body* contains the body of the message, whose meaning depends on the value of the Type field.

The protocol is strictly request-response: to every request corresponds exactly one response. However, requests and replies may flow in both directions. The following requests are defined:

- *Type* = 0, Ping. The peer responds with Ok.
- *Type* = 1, Message. The peer responds with Ok. The body has the following format:



The following replies are defined:

- *Type* = 128 : *Ok*. The body is empty.
- *Type* = 129 : *Error*. The body is an error message encoded in UTF-8.

Exercice 1.

1. Send a `GET` request to the server in order to retrieve the list of available peers.
2. Send a chat message to the peer called `jch` and wait for a reply (the *Message-Id* field should be randomly generated.) Make sure that you handle the case where a request from the peer arrives before the reply, and make sure that you send a reply to the request.
3. Implement retransmission of requests with exponential backoff.

Exercice 2. Since we haven't implemented NAT traversal yet, in this exercice you will need to do one of the following:

- run a private server on the same network as your peers; or
- set up port forwarding; or
- use IPv6.

1. Write a program that listens on a UDP socket. Your program will receive `ping` and `chat` requests (and display the received messages) and reply with `ok`.
2. After the UDP socket is established, send a PUT request to the server with a JSON body obeying the following grammar:

```
register = {
    port: uint,
}
```

   Omit the `token` parameter for now.
3. In order to avoid nickname hijacking, the server will ignore `register` requests that use a token different from the one used previously: if a client registers with a nickname n and a token t_1, the server will reject registrations with the same nickname n and a different token $t_2 \neq t_1$.
   a) Give an example of an attack that this mechanism successfully defends against.
   b) Give two examples of attacks that this mechanism does not successfully defend against.
   c) Modify your program so that it sends a token with each `register` request.
4. Describe a situation in which your peer might receive the same *Message-Id* twice. Implement a solution to the problem.
5. Modify your program so that it sends a `DELETE` request when it terminates.