Advanced Networks — Laboratory 6

Juliusz Chroboczek

2 April 2025

Exercice 1. Write a Go program that connects to the WebSocket `wss://galene.org:8445/chat/ws` then sends a text message containing the following string

```
{"type": "get", "count": 20}
```

then displays all of the messages received (forever or until the server closes the connection) without decoding them. You may use the package `github.com/gorilla/websocket`.

Exercice 2. The server implements a JSON-based protocol. Both requests (messages from client to server) and events (messages from server to client) can be encoded by the following Go structure:

```
type jsonMessage struct {
        Type     string        `json:"type"`
        Message  *chatMessage  `json:"message,omitempty"`
        Messages []chatMessage `json:"messages,omitempty"`
        Count    int           `json:"count,omitempty"`
        Error    string        `json:"error,omitempty"`
}

type chatMessage struct {
        Id   string    `json:"id,omitempty"`
        Time time.Time `json:"time,omitempty"`
        Body string    `json:"body"`
}
```

Modify your program so that it uses the `WriteJSON` and `ReadJSON` methods to encode and decode the messages. Why is the field `Message` a pointer?

Exercice 3. The syntax of the protocol can be described by the following CDDL (RFC 8610) grammar. The production `request` defines the syntax of client requests, the production `reply` defines the syntax of server replies, and the production `event` the syntax of unsolicited server notiications.

```
chatMessage = {
    ? id tstr,
    ? time tstr,
    body tstr,
}

request = pingRequest /
          getRequest /
          subscribeRequest /
          unsubscribeRequest /
          postRequest

pingRequest = {
    type: "ping",
}

getRequest = {
    type: "get",
    count: uint,
}

subscribeRequest = {
    type: "subscribe",
    ? count: uint,
}

unsubscribeRequest = {
    type: "unsubscribe",
}

postRequest = {
    type: "post",
    message: chatMessage,
}

reply = okReply / messagesReply / errorReply

okReply = {
    type: "ok",
}

messagesReply = {
    type: "messages",
    messages: [ * chatMessage ],
```

```
}

errorReply = {
    type: "error",
    error: tstr,
}

event = messageEvent

messageEvent = {
    type: "message",
    message: chatMessage,
}
```

The semantics of the requests is as follows[1]:

- `ping`: the server replies with `ok`;
- `get`: the server replies with a `messages` reply containing the last `count` messages;
- `subscribe`:the server sends a `messages` reply containing the last `count` messages, then sends `message` events as new messages are posted;
- `unsubscribe`: the server replies with `ok`, then stops sending `message` events;
- `post`: the server posts a new message, then replies with `ok`.

1. Why did I use CDDL rather than JSON-Schema to define the syntax of the protocol?
2. Verify that the Go structure given in the previous exercice can encode all of the protocol messages. What is the property that makes it easy?
3. Give an example of a message that can be encoded in the Go structure, but does not belong to the protocol. How would this flaw be avoided in TypeScript? In Coq? In Rust?
4. Modify the program from the previous exercice so that it displays the last 20 messages and then displays messages as they are posted on the server.
5. Why does the `subscribe` request return the last `count` messages? Would it be correct to send a `get` request immediately followed with a `subscribe` with no `count` parameter?
6. The protocol obeys a strict request-response discipline: all client requests are answered with exactly one server reply. What are the two reasons for that? Are they relevent to this protocol?
7. Suppose that we were to add a new field to one of the events. Would that break existing clients? What about adding a new event type?
8. What happens if we delete a message (using a `DELETE` method on the REST endpoint) while a client is subscribed?
9. Modify your program to send a `ping` request if it hasn't received an event in 20 s, and to disconnect from the server and display an error message if it hasn't received an event in 35 s.
10. Write a program that reads chat messages from the keyboard and posts them to the chat over the WebSocket.
11. Write a single program that can simultaneously read messages from the keyboard and display messages received from the server.

1. Notice how the syntax is defined using a formal notation, while the semantics is defined using natural language.