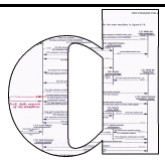
DEPENDABLE SYSTEMS AND SOFTWARE

Data Networks 2015 Dr. Arnd Hartmanns



PL2: WebSocket Chat Server

The soft deadline for this programming lab is July 05. The hard deadline is July 12.

Submission: Your submission must be a `.zip` file that contains the C# or Java source code files of your solution for this lab. In the case of C#, the resulting executable file must be named `ChatServer.exe`, and a file named `ChatServer.csproj` that can be used to compile your program using MSBuild must be at the top level of the `.zip` file (i.e. not within any subfolders). In the case of Java, a file `Compile.txt` with straightforward instructions on how to compile your program that do not require tools other than those included in the current stable Java SDK must be included at the top level of your `.zip` file. Your `.zip` file must not contain any compiled binary files (e.g. `bin` or `obj` folders, `.class` or `.jar` files).

Group work: All code you submit must have been written by members of your group, and must not reference any third-party libraries. However, you may use the .NET or Java standard libraries except for any existing WebSocket implementations. Your solution must be easily readable and well-commented.

Your task is to implement the server side of the *dnChat* protocol from the previous programming lab. It shall be a command-line program that, when started, provides a *dnChat* server listening for incoming connections from any network interface on the protocol's default port. When the user types a line consisting of the string `exit`, the server shall cleanly shut down and exit. The server shall allow multiple clients to be connected and to exchange chat messages simultaneously. It shall behave like the reference server at `modestchecker.net`, forwarding chat messages to their intended recipients and passing acknowledgments back to the sender of a message. Your server shall at least allow logins with any name and your group password. The networking and protocol layers involved in your implementation shall be cleanly separated in code.

Simplifying remark. Your server does not need to support WebSocket message fragmentation, since no common browser today fragments messages; but if it indeed does not support fragmentation, your server must cleanly close the WebSocket connection when it receives a fragmented message (in particular, it must neither crash nor completely ignore the message).

Hints. A code template for C# is available in dCMS. Its use is entirely optional. In any case, we recommend you proceed in small steps; for example:

1. Implement accepting a single TCP connection and performing the WebSocket handshake.
   Test by creating a minimal JavaScript client running in your browser that connects to your server and verifies that the WebSocket connection is opened successfully.
2. Implement sending WebSocket text messages from your server to the client.
   Test by having the server send a text message to the client upon connection, and displaying all text messages received from the server (e.g. via JavaScript's `alert` function) in your minimal client.
3. Implement sending and receiving of WebSocket control messages (`ping`, `close` etc.).
   Test by closing the WebSocket connection a) from the server and b) from the minimal client and checking that it is closed cleanly on both sides.
4. Implement receiving of WebSocket text data messages.
   Test by sending a text message to your server from the minimal client, and sending it back from the server to the client, where it is then displayed.
5. Add support for multiple clients connected simultaneously.
6. Implement the server side of the *dnChat* protocol on top of your WebSocket implementation.
   User your own client from the previous programming lab for testing.