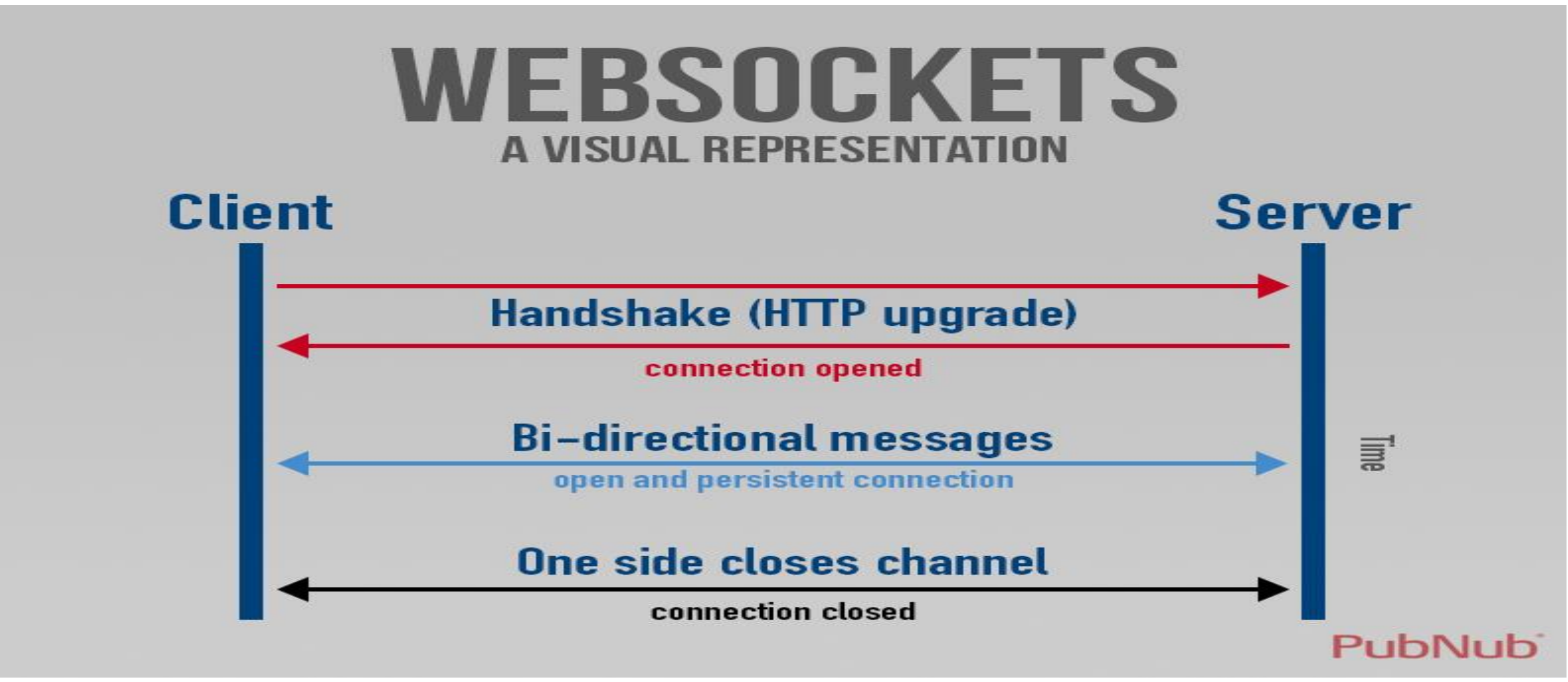
METU EE444 Introduction to Computer Networks

HW3 – WebSockets

Due: June 22, 2020, 23:55



You are going to submit your homework via **ODTUCLASS** as a **.zip** file containing the relevant source files (just *main.cpp* and *websocket.h*). Name your file as “HW3_studentid.zip”.

Using code directly taken from any kind of resource is prohibited. Cheating will result in zero grade, whereas disciplinary actions may also be taken. Late submissions are not allowed.

Notice

This homework introduces a new application layer protocol that can be useful for your future use in your workplace or academic studies. So please first read the introductory material. Alongside this document, you are given source code files (named **websockets** folder in the provided .zip) containing parts of the solution as a starting point. You will fill the left out sections indicated both in the code and in this document. **Do not change the structure of the code unnecessarily.**

What are WebSockets?

WebSocket is an application layer protocol (RFC 6455) for bidirectional communication over a TCP connection. Functionally, they are very similar to BSD Sockets (the usual sockets we have learned) in STREAM (TCP) mode. There are two major differences:

- WebSockets are initialized over HTTP connections
- WebSockets wrap the data in *frames*

Since the HTTP protocol already works over TCP, it might seem pointless to introduce a new layer of abstraction to do basically the same thing with WebSockets. But there is an important use case: Today, almost all application servers on the web work behind a **reverse proxy**. A reverse proxy is a proxy server that requests resources from the web servers on behalf of the clients. They are used to improve performance, security and reliability (sometimes also as a load balancer). They inspect the client-server communication protocol for various reasons such as caching and preventing DDoS attacks. However, since usual sockets do not follow any protocol regarding the data contained, they might cause issues with the intermediaries such as reverse proxies. Think of a TCP socket transferring text either intentionally or unintentionally containing the string **“HTTP/1.1 301 Moved Permanently”**. In this case, a reverse proxy may try to cache the result of this accidental message that looks like a HTTP request and cause numerous issues. For this and some other reasons related to security, using sockets without an application layer protocol like HTTP (or FTP, …) is not preferred. In fact, most modern browsers do not even provide an API for socket communications.

On the other hand, the need for real-time (or near-real-time, low latency) data transfer between web application frontends (the page you see in browser) and the backend (the server responsible of the API functions) became increasingly relevant as web applications demanded more and more up-to-date data. Examples to such applications may include social media, instant messaging, chat rooms, online stock market and trading websites.

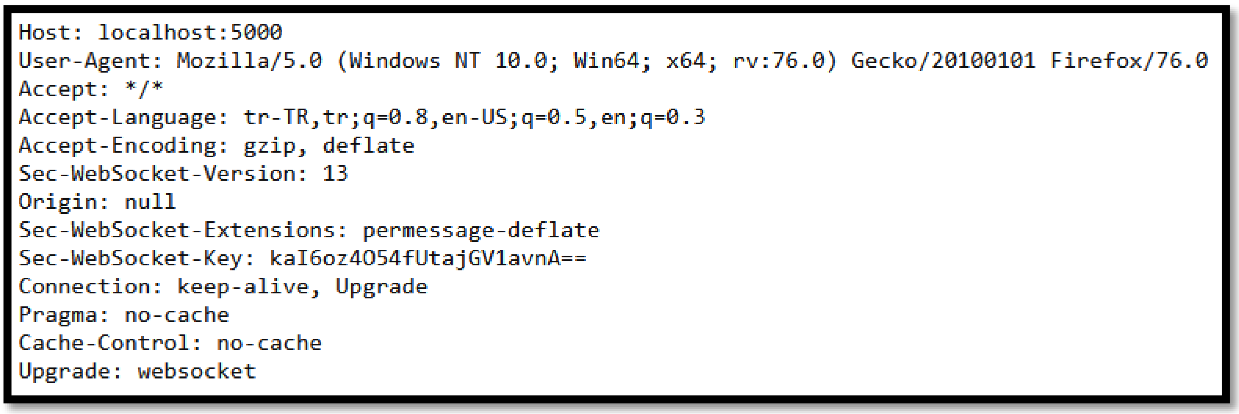
In these applications, the direction of data is generally from the server to the client, whereas the HTTP (mostly) works with a client initiated connection model. There are three well known methods to mitigate this problem:

- **Regular polling:** polling at a regular interval (too much average latency),
- **Long-polling:** a clever way of keeping HTTP connection open until the server has something to send (too much resource usage and unpredictable timeouts),
- **WebSockets.**

How do WebSockets work?

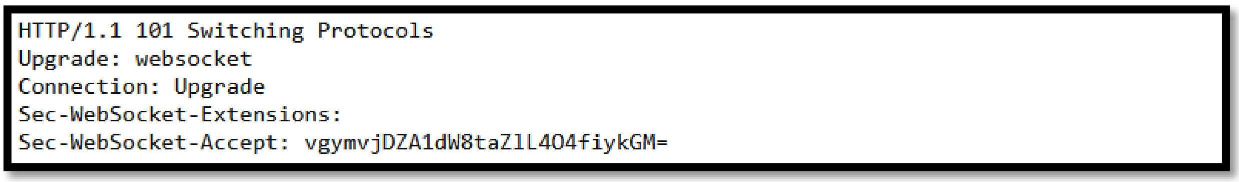
1) Initialization

WebSockets start as a usual HTTP connection with a **HTTP Upgrade Request (see Figure 1)**. This is a usual GET request with a special "*Upgrade: websocket*" header line. The server may accept the upgrade request and initiate the websocket communication, or simply refuse. This is especially useful to prevent mistakenly sending binary data to servers not supporting WebSockets. If the server proves that it is capable of WebSockets and accepts the upgrade request with a **"HTTP/1.1 101 Switching Protocols" (see Figure 2)** response, the websocket communication starts and both sides can send and receive textual or binary data in the required frame format from this point on. Refer to Mozilla Developer Network for more details on protocol upgrade mechanism.



```
Host: localhost:5000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:76.0) Gecko/20100101 Firefox/76.0
Accept: */*
Accept-Language: tr-TR,tr;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: null
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key: kaI6oz4O54fUtajGV1avnA==
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

Figure 1: HTTP Upgrade request of the WebSocket client



```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Extensions:
Sec-WebSocket-Accept: vgymvjDZA1dW8taZlL4O4fiykGM=
```

Figure 2: Protocol switch (upgrade accept) response of the server

The request contains the field *"Sec-WebSocket-Key"* whose value is actually the base64 encoded form of a 16-byte random sequence. The server is needs to take this 24 characters long string, append to it a magic GUID (globally unique identifier) valued *"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"* (constant, has no meaning, just a magic number), take its SHA1 hash, and encode the resulting 20 bytes in base64. The server puts this new string as the value of *"Sec-WebSocket-Accept"* field in the response. This ensures two things: First, since the key changes randomly, a reverse proxy cannot cache its response as the request changes every time. Second, a web server incapable of WebSockets would not be able to calculate the true accept-string and accidentally switch the protocols.

2) Data Format

WebSockets follow a fairly simple binary frame format. Meanings and possible values of all fields can be found in RFC 6455. The frame format from the RFC is given in [3].

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-------+-+-------------+-------------------------------+
 |F|R|R|R| opcode|M| Payload len |    Extended payload length    |
 |I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
 |N|V|V|V|       |S|             |   (if payload len==126/127)   |
 | |1|2|3|       |K|             |                               |
 +-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
 |     Extended payload length continued, if payload len == 127  |
 + - - - - - - - - - - - - - - - +-------------------------------+
 |                               |Masking-key, if MASK set to 1  |
 +-------------------------------+-------------------------------+
 | Masking-key (continued)       |          Payload Data         |
 +-------------------------------- - - - - - - - - - - - - - - - +
 :                     Payload Data continued ...                :
 + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
 |                     Payload Data continued ...                |
 +---------------------------------------------------------------+
```

Figure 3: WebSocket data framing

The first two bytes contain information related to processing the frame:

- FIN bit denotes whether this frame is the last one in the sequence,
- RSV bits are reserved (must be 0),
- Opcode tells the function of the frame,
- MASK bit determines if unmasking should be applied to the payload,
- Payload Length representing the length of payload in bytes

By default, the 7-bits payload length field can only represent up to **125 bytes** of payload. For more data, the payload length is set to 126 and 16 additional bits represent the length. For even more data, payload length can be 127 and the following 64 bits are used as the length.

One thing to note about this figure is that fields noted with an if-condition are simply removed from the frame (instead of being filled with zeros, for example) if the condition is not satisfied.

The Opcode is 0x0 for continuation frames, 0x1 for UTF-8 text payload, 0x2 for binary payload, 0x8 for connection close command. The rest are for some other functionality.

Masking is a feature to prevent data repetition in the case of **Secure WebSockets**. This is because when encrypted, repetitive data facilitates some forms of attacks on the encryption scheme. If the MASK bit is set, the frame should contain a 4-byte masking key. This means the payload will be masked with this masking key. The masking procedure is very simple:

- Start reading the provided payload and the masking keys first byte and XOR them to obtain the masked payloads first byte.
- Continue to next byte in both payload and the masking key

- When the masking key is consumed, start from the beginning of masking key again.

In a more formal way:

```
Masked_Payload[i] = Payload[i] XOR Masking_Key[i MOD 4]
```

Masking is applied only for Client-to-Server frames and due to the nature of XOR operation, the unmasking procedure is exactly the same.

Implementation

In the scope of this homework, we will work with a simplified version of this frame format. We will assume that no frame will be fragmented (divided into multiple parts). This implies that every frame should have its **FIN** bit set and maximum payload length is 125 bytes. Also, we transfer only text data **(opcode=0x1)** and handle only received text data **(opcode=0x1)** or connection close command **(opcode=0x8)**. We will only develop the server code (the client code is fairly similar), so we will only need to apply unmasking to received frames. The language of development is **C**. The code for calculating the **accept key** using **base64** and **SHA1** functions is provided for your ease. The relevant files for implementation of the homework are **main.c** and **websocket.h**, meaning that you do not need to modify any other file.

Start from the **main.c** file and try to follow along with the comments step by step and implement the functionality requested as you face them. You are also given a **test.html** file which implements a simple browser based WebSocket client, a central area to display messages sent by the server (the C code) and a text input to send text data through the WebSocket (should work with Mozilla Firefox, Google Chrome and Opera; may not work with Internet Explorer or Microsoft Edge).

The given files come with a preconfigured Code::Blocks project. If for any reason you cannot use the provided project, you can create a new project elsewhere and move the source and header files (*.c and *.h). **Do not forget to:**

- Add the source files to the project by right clicking the project name and "Add files…".
- Add #include search directory (Build Options > Search Directories > Add > Directory: . (dot))
- Add the linker flag for Winsocks2 library (Build Options > Linker Settings > Add > ws2_32)

You can refer to the examples provided in the "socket-demo-instructions" on ODTUCLASS.

Note 1: The code pieces left for you to implement should range from a single line to not more than 15-20 source lines of code. You may do some bitwise operations to set or test bit fields such as the MASK field.

Note 2: Put the keyword **struct** before struct types such as **sockaddr** (changes from C++ in the socket demo to C in the homework)

Note 3: During development, you can test your code multiple times to see what you receive and what you output. You may find it useful to add printf()'s for quick debugging.

1) Phase 1: Construct a socket and accept clients
    a. Initialize Windows sockets library
    b. Create the socket
    c. Configure socket address struct
    d. Bind socket to address
    e. Set maximum backlogged (waiting) connections
    f. Accept client connections
2) Phase 2: Receive the upgrade request and respond
    a. receive the http request
    b. Use send() function to send the protocol switch response
3) Phase 3: Communicate over WebSocket protocol
    a. Construct a simplified WebSocket frame in buffer from a text payload
    b. Parse a simplified WebSocket frame and copy the payload
4) Phase 4: Cleanup resources
    a. Close client socket
    b. Close server socket
    c. De-initialize Windows sockets

Advanced Websocket Example

In the provided .zip, the project **"chat-room-ws-example"** shows a fairly complete real-time chat room with chat commands (requires MinGW GCC 5.4+, its best to use the latest Code::Blocks IDE v20.03). This example uses a library called **Websocketpp** and demonstrates the modern C++ syntax, programming practices like **lambda functions** and the **Pimpl idiom**. The Websocketpp is configured to use the **standalone asio** library. If you are interested, look for **boost::asio** (**boost** is also a very important C++ library).

For a demonstration of the chat room example, you can watch:
https://www.youtube.com/watch?v=TUpZEabeA-8