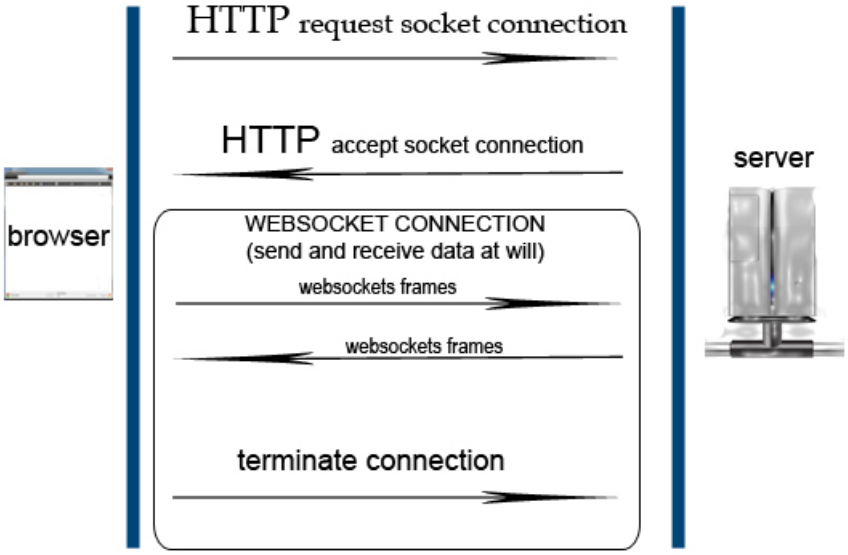
Programowanie Aplikacji Sieciowych - Laboratorium 10

WebSocket (RFC 6455) jest protokołem opartym o TCP, zapewniającym dwukierunkową komunikację pomiędzy klientem a serwerem (dwukierunkowy kanał komunikacji za pośrednictwem jednego gniazda TCP). Po zestawieniu połączenia, obie strony mogą wymieniać się danymi w dowolnym momencie, wysyłając pakiet danych. (Inaczej niż w przypadku protokołu HTTP, gdzie najpierw należało wysłać zapytanie (ang. *request*), aby otrzymać odpowiedź (ang. *response*), nie pojedyncze żądania i odpowiedzi, ale stałe połączenie.) Domyślne porty wykorzystywane przez protokół WebScket to port TCP o numerze 80 (niezabezpieczony) oraz port TCP o numerze 443 (zabezpieczony).

Podstawowe informacje:

- Identycznie jak w protokole HTTP, każda komenda w żądaniu i odpowiedzi zakończona jest parą znaków <CRLF> (czyli \r\n)
- Strona zainteresowana nawiązaniem połączenia (klient), wysyła do serwera żądanie inicjalizujące połączenie (ang. *handshake*), na które wysyłana jest odpowiedź serwera
- Żądanie to, ze względów na kompatybilność z serwerami WWW, jest niemal identyczne jak standardowe zapytanie HTTP. Format odpowiedzi serwera również jest zgodny z odpowiedziami znanymi z protokołu HTTP
- Po zestawieniu połączenia, obie strony mogą wymieniać się danymi w dowolnym momencie, wysyłając pakiet danych (inaczej niż w protokole HTTP, gdzie komunikacja wyglądała w ten sposób, iż na każde żądanie wysyłana była odpowiedź)
- Komunikacja w protokole WebSocket wygląda więc następująco:
  1. Klient wysyła żądanie do sewera (w formacie podobne do żądania HTTP)
  2. Serwer odsyła klientowi odpowiedź (podobną w formacie do odpowiedzi HTTP)
  3. Po udanym handshake'u (kroki 1 i 2), klient i serwer mogą wymieniać między sobą wiadomości, jednak wiadomości muszą być przesyłane w specjalnie sformatowanych ramkach

Komunikacja przy użyciu protokołu WebSocket



Jak wynika z powyższego rysunku, do nawiązania połączenia (handshake) wykorzystywany jest protokół HTTP. Po pomyślnym zakończeniu nawiązywania połączenia, dalsza komunikacja odbywa się poprzez socket TCP już z pominięciem protokołu HTTP. Dane przesyłane są w specjalnie sformatowanych ramkach.

Realtime Web

- Rozwiązania czasu rzeczywistego (*real-time*) dostarczają odbiorcy informacje w chwili pojawienia się ich u źródła, w przeciwieństwie do zwykłych stron, które użytkownik musi ręcznie odpytać, by sprawdzić, czy nie pojawiły sie nowe dane (tak było w przypadku protokołu HTTP)
- Protokól używany do przesyłania stron internetowych (HTTP) to protokól typu żądanie - odpowiedź, gdzie klient (przeglądarka) jest zawsze inicjatorem połączenia, czyli wysyła żądanie do serwera. Nie ma problemu, gdy to klient chcę przesłać informację w kierunku serwera (np. nadawca wiadomości na czacie), lecz jak serwer ma poinformować odbiorcę (innego klienta) o nowych danych? Przecież serwer HTTP nie może inicjować połączeń z klientem! Można by czekać, aż odbiorca sam "odświeży" stronę, ale z pewnością nie będzie to *real-time*
- Czaty, kanały RSS, gry: takie programy chcemy uruchamiać w przeglądarce bez konieczności instalowania plug-inów. Oczywiście aplikacje powinny działać w czasie rzeczywistym i automatycznie się aktualizować. Jednak klasyczna koncepcja sieci nie była opracowana z myślą o tych nowoczesnych możliwościach internetu i rzucała programistom wiele kłód pod nogi
- W historii Internetu rozwiązań, kompromisów oraz obejść powyższego problemu bylo wiele, m.in.:
  - odpytywanie
  - wykorzystanie pluginów (Flash, Java) i niestandardowych polaczen TCP
  - ...
- Aby rozwiązać powyższy problem, konieczna była kompletna zmiana w komunikacji klient-serwer: odejście od typowego dla protokołu HTTP jednokierunkowego modelu pytanie-odpowiedź na rzecz połączeń dwukierunkowych w czasie rzeczywistym
- WebSocket rozwiązuje opisane wcześniej problemy, tworząc w przeglądarce (raczej: po stronie klienta, ponieważ nie zawsze musi to być przeglądarka) socket, który przez adres IP i port utrzymuje obustronny kanał do serwera. Dzięki temu oba punkty końcowe mogą równocześnie przesyłać dane przez to samo połączenie
- Za pomocą protokołu WebSocket, w przeciwieństwie do HTTP, można wysyłać dane w dwóch kierunkach równocześnie przez jedno połączenie TCP. W efekcie zmniejszają się opóźnienia i obciążenie sieci
- WebSocket bazuje na HTTP, lecz po nawiązaniu połączenia zastępuje ten protokół
- Połączenia WebSocket, podobnie jak w HTTP, wykorzystują standardowe porty: 80. gdy nie są szyfrowane, i 443 w przeciwnym wypadku
- WebSocket ma schemat analogiczny do HTTP: ws (Web Servieces) dla połączeń nieszyfrowanych i wss (Web Secure Servieces) dla połączeń szyfrowanych

Format żądania Handshake w protokole WebSocket

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
```

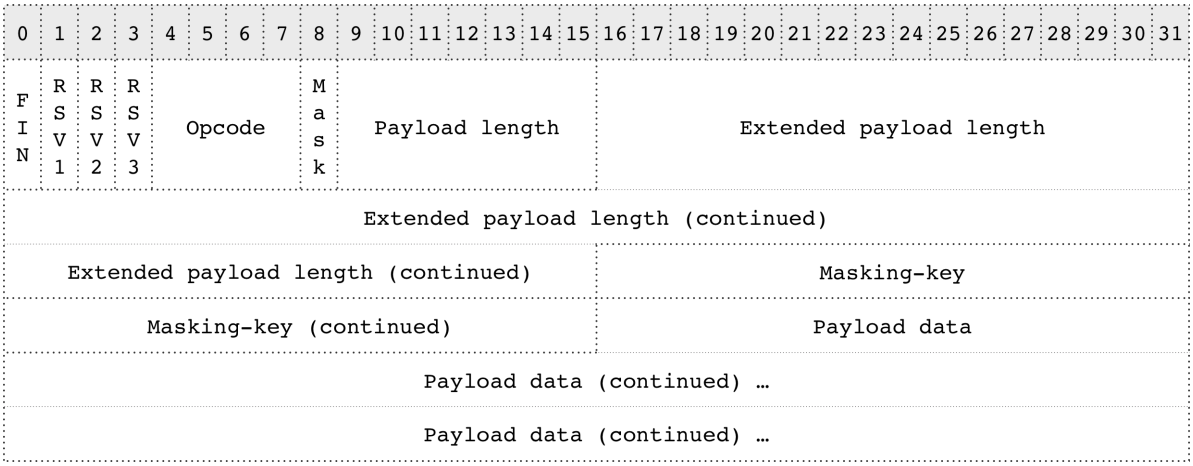
- `GET /chat HTTP/1.1` jest to początek żądania protokołu WebSocket, identyczny jak w protokole HTTP: `GET` - metoda żądania, `/chat` - zasób, `HTTP/1.1` - wersja protokołu HTTP. W żądaniu protokołu WebSocket wymagane jest, aby wersją protokołu HTTP była wersja `1.1`, oraz aby metodą HTTP była metoda `GET`
- `Host: server.example.com` nagłówek `Host` zawiera nazwę domeny, do której skierowane jest żądanie
- `Upgrade: websocket` informuje serwer, że klient chce nawiązać połaczenie za pomocą protokołu WebSocket
- `Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==` wbrew nazwie, nie zawiera klucza, lecz losowy ciag znaków zakodowanych base64 (na początku generujemy losowy ciąg znaków, który następnie kodujemy za pomocą base64, wykoniowa wartość to właśnie wartość nagłówka `Sec-WebSocket-Key`)
- `Origin: http://example.com` informuje serwer, z jakiego adresu wysyłamy żądanie
- `Sec-WebSocket-Protocol: chat` służy do poinformowania serwera, z jakiego protokołu klient chce skorzystać
- `Sec-WebSocket-Version: 13` służy do poinformowania serwera, z której wersji protokołu klient chce skorzystać

Format odpowiedzi Handshake w protokole WebSocket

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

- `HTTP/1.1 101 Switching Protocols` jedyna właściwa odpowiedź od serwera web socket, jeśli kod odpowiedzi jest inny niż `101`, oznacza to, że handshake się nie powiódł, nagłówek ten informuje klienta, że powiodła się zmiana protokołu z HTTP na WebSocket, dod odpowiedzi `101` oznacza, ze protokół dla tego połaczenia został zmieniony na taki, jaki jest wpisany w nagłówku `Upgrade`, czyli WebSocket - kod odpowiedzi 101 oznacza, że serwer wspiera protokół WebSocket i wyraża zgodę na nawiązanie połączenia
- `Upgrade: websocket` serwer potwierdza, że obsługuje protokół WebSocket
- `Connection: Upgrade` jak wyżej
- `Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=` serwer, odbiera od klienta nagłówek `Sec-WebSocket-Key: ...`, w którym to klient wysyła losowy ciąg bajtów. Serwer pobiera ten ciąg od klienta, i dokleja na końcu tego ciągu stałą (tzw. GUID): `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` (to stały ciąg znaków ustalony dla każdej odpowiedzi handshake w protokole WebSocket). Następnie, gdy serwer doklei do danych podany ciąg, wykonuje na nim funkcję `SHA1`, a następnie wynik funkcji `SHA1` ponownie koduje za pomocą `base64`
- `Sec-WebSocket-Protocol: chat` w tym nagłowku `Sec-WebSocket-Protocol` serwer zwraca nazwe protokołu, który otrzymał w zapytaniu jako potwierdzenie wyboru tego protokołu

Format ramki w protokole WebSocket



- `FIN` - (1 bit) - dla naszych zastosowań przyjmujemy zawsze wartość 1. WebSocket, w przeciwieństwie do np. TCP, nie jest strumieniem bajtów. Zamiast tego przesyła się wiadomości o określonej długości. Długość wiadomości nie musi być z góry znana. W takim wypadku wysyła się ją we fragmentach, a w ostatnim fragmencie bit `FIN` w nagłówkach WebSocket frame przyjmuje wartość 1, który mówi o tym, że nastąpił koniec wiadomości
- `RSV1, RSV2, RSV3` - (1 bit każde) - dla naszych zastosowań przyjmujemy zawsze wartość 0
- `Opcode` - (4 bity) - pole to określa, w jaki sposób należy interpretować ramkę:
  - Jeśli pole `Opcode` ma wartość `0`, wówczas dane są kontyfuacją poprzedniej ramki,
  - Jeśli pole `Opcode` ma wartość `1`, wówczas dane przesyłane są w formie tekstowej,
  - Jeśli pole `Opcode` ma wartość `2`, wówczas dane przesyłane są w formie binarnej,
  - Wartości pola `Opcode 3-7` są zarezerwowane do przyszłych zastosowań,
  - Jeśli pole `Opcode` ma wartość `8`, wówczas oznacza to chęć zakończenia połączenia
  - Jeśli pole `Opcode` ma wartość `9`, wówczas oznacza to przesłanie ramki typu `ping`
  - Jeśli pole `Opcode` ma wartość `10`, wówczas oznacza to przesłanie ramki typu `pong`
  - Wartości pola `Opcode 11-15` są zarezerwowane do przyszłych zastosowań
- `MASK` - (1 bit) - oznaczenie, czy dane sa maskowane (wartość 1), czy nie (wartość 0). Zgodnie z standardem, każdy z wysyłanych pakietów od klienta do serwera, musi posiadać ustawiony bit mask. W przypadku gdy zostanie on ustawiony, w polu payload nie zostają umieszczone przesyłane dane w postaci jawnej, ale ich zamaskowana postać. Przez zamaskowanie mamy na myśli wynik działania funkcji XOR, na ciągach znaków z pola masking-key oraz wysyłanych danych. Każdy pakiet danych od klienta do serwera 'zabezpieczony' jest za pomocą prostej maski bitowej XOR, aby włączony jako pośrednik serwer proxy błędnie nie wziął ruchu WebSocket za zapytania HTTP. Bez szyfrowania pakietów danych złośliwe skrypty mogłyby sterować serwerami proxy i wykorzystywać je do atakowania innych użytkowników. Jednak ponieważ serwery proxy nie potrafią odczytywać zaszyfrowanych danych, bezproblemowo przekazują je do podanego punktu końcowego
- `Payload length` - (7 bitów) - pole to określa długość danych w ramce (jeśli długość danych jest mniejsza lub równa 125 bajtów)
- `Extended payload length` - (16 bitów) - jeśli wartość poprzedniego pola jest równa 126, to długość danych jest zawarta w tych 16 bitach
- `Extended payload length continued` - (48 bitów) - jeśli wartość pola `payload len` jest równa 127, to długość danych jest zawarta w 8 bajtach z pól `extended payload length` oraz `extended payload length continued`
- `Masking-key` - (32 bity) - klucz, którym zamaskowano dane (jesli pole `MASK` jest ustawione). Maskowanie polega na xorowaniu kazdego bajtu danych z kolejnym (powtarzanym cyklicznie) bajtem klucza, pseudokod: `payload[i] ^ maskingkey[i % 4]`
- `Payload` - przesyłane dane

Żądanie Handshake w protokole WebSocket

```
▸ Internet Protocol Version 4, Src: 192.168.0.2, Dst: 174.129.224.73
▸ Transmission Control Protocol, Src Port: 57795, Dst Port: 80, Seq: 1, Ack: 1, Len: 575
▾ Hypertext Transfer Protocol
  ▸ GET /?encoding=text HTTP/1.1\r\n
    Host: echo.websocket.org\r\n
    User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:50.0) Gecko/20100101 Firefox/50.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-US,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Sec-WebSocket-Version: 13\r\n
    Origin: http://www.websocket.org\r\n
    Sec-WebSocket-Extensions: permessage-deflate\r\n
    Sec-WebSocket-Key: 3z3k2knjK8YWpZN5thoMAg==\r\n
  ▸ Cookie: __zlcmid=gNgQE3hhRmyyJJ\r\n
    DNT: 1\r\n
    Connection: keep-alive, Upgrade\r\n
    Pragma: no-cache\r\n
    Cache-Control: no-cache\r\n
    Upgrade: websocket\r\n
    \r\n
    [Full request URI: http://echo.websocket.org/?encoding=text]
    [HTTP request 1/1]
    [Response in frame: 132]
```

Odpowiedź Handshake w protokole WebSocket

```
▸ Internet Protocol Version 4, Src: 174.129.224.73, Dst: 192.168.0.2
▸ Transmission Control Protocol, Src Port: 80, Dst Port: 57795, Seq: 1, Ack: 576, Len: 542
▾ Hypertext Transfer Protocol
  ▸ HTTP/1.1 101 Web Socket Protocol Handshake\r\n
    Access-Control-Allow-Credentials: true\r\n
    Access-Control-Allow-Headers: content-type\r\n
    Access-Control-Allow-Headers: authorization\r\n
    Access-Control-Allow-Headers: x-websocket-extensions\r\n
    Access-Control-Allow-Headers: x-websocket-version\r\n
    Access-Control-Allow-Headers: x-websocket-protocol\r\n
    Access-Control-Allow-Origin: http://www.websocket.org\r\n
    Connection: Upgrade\r\n
    Date: Fri, 05 May 2017 12:41:02 GMT\r\n
    Sec-WebSocket-Accept: G1phXK+xuMJLAjUNMFsu2K0Uf64=\r\n
    Server: Kaazing Gateway\r\n
    Upgrade: websocket\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.140858062 seconds]
    [Request in frame: 130]
```

Dane przesyłane od klienta do serwera w protokole WebSocket

```
▸ Internet Protocol Version 4, Src: 192.168.0.2, Dst: 174.129.224.73
▸ Transmission Control Protocol, Src Port: 57795, Dst Port: 80, Seq: 576, Ack: 543, Len: 18
▾ WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0001 = Opcode: Text (1)
    1... .... = Mask: True
    .000 1100 = Payload length: 12
    Masking-Key: 89761d36
    Masked payload
    Payload
▾ Line-based text data
    hello world!
```

```
0000  c4 3d c7 b8 c1 16 5c f9  dd 59 d1 31 08 00 45 00   .=....\. .Y.1..E.
0010  00 46 48 8f 40 00 40 06  a2 ad c0 a8 00 02 ae 81   .FH.@.@. ........
0020  e0 49 e1 c3 00 50 77 ae  01 94 32 70 c6 f9 80 18   .I...Pw. ..2p....
0030  00 ed 38 32 00 00 01 01  08 0a 00 48 b9 08 3d 19   ..82.... ...H..=.
0040  63 5a 81 8c 89 76 1d 36  e1 13 71 5a e6 56 6a 59   cZ...v.6 ..qZ.VjY
0050  fb 1a 79 17                                       ..y.
```

Dane przesyłane od serwera do klienta w protokole WebSocket

```
▸ Internet Protocol Version 4, Src: 174.129.224.73, Dst: 192.168.0.2
▸ Transmission Control Protocol, Src Port: 80, Dst Port: 57795, Seq: 543, Ack: 594, Len: 14
▾ WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0001 = Opcode: Text (1)
    0... .... = Mask: False
    .000 1100 = Payload length: 12
    Payload
▾ Line-based text data
    hello world!
```

```
0000  5c f9 dd 59 d1 31 c4 3d  c7 b8 c1 16 08 00 45 00   \..Y.1.= ......E.
0010  00 42 a7 84 40 00 2d 06  56 bc ae 81 e0 49 c0 a8   .B..@.-. V....I..
0020  00 02 00 50 e1 c3 32 70  c6 f9 77 ae 01 a6 80 18   ...P..2p ..w.....
0030  00 ec 63 ad 00 00 01 01  08 0a 3d 19 64 60 00 48   ..c..... ..=.d`.H
0040  b9 08 81 0c 68 65 6c 6c  6f 20 77 6f 72 6c 64 21   ....hell o world!
```

Przesyłanie ramki WebSocket w języku Python za pomocą gniazd (Python 2)

```
frame = bytearray()
frame.append(int('10000001', 2))
frame.append(0x8c)
# ...
sock.sendall(str(frame))
```

Przesyłanie ramki WebSocket w języku Python za pomocą gniazd (Python 3)

```
frame = bytearray()
frame.append(0x81)
frame.append(int('10001100', 2))
# ...
sock.sendall(frame)
```

Przesyłanie ramki WebSocket w języku C/C++ za pomocą gniazd

```
unsigned char frame[18];
frame[0] = 0x81;
frame[1] = 0x8c;
// ...
send(sock, frame, 18, 0);
```

lub (nie wszystkie kompilatory na to pozwalają):

```
unsigned char frame[18];
frame[0] = 0b10000001;
frame[1] = 0b10001100;
// ...
send(sock, frame, 18, 0);
```

lub (zakładając, że zaimplementujemy funkcję `bin2hex`):

```
unsigned char frame[18];
frame[0] = bin2hex("10000001");
// ...
send(sock, frame, 18, 0);
```

Przesyłanie ramki WebSocket w języku Java za pomocą gniazd

```
byte frame[] = new byte[18];
frame[0] = (byte) (0x81);
frame[1] = (byte) (0x8c);
// ...
OutputStream writer = socket.getOutputStream();
writer.write(frame);
writer.flush();
```

lub (zakładając, że zaimplementujemy funkcję `bin2dec`):

```
byte frame[] = new byte[18];
frame[0] = (byte) bin2dec("10000001");
// ...
writer.write(frame);
writer.flush();
```

Uwaga W poniższych zadaniach zakładamy, iż serwer powinien obsługiwać tylko jednego klienta w danej chwili.

Pod adresem `ws://echo.websocket.org` na porcie 80 (wersja niezabezpieczona) oraz `wss://echo.websocket.org`, port 443 (wersja zabezpieczona) udostępniony jest serwer obsługujący protokół WebSocket.

Udostępniony jest też web client, do przetestowania serwera, w wersji niezabezpieczonej i zabezpieczonej, odpowiednio `http://websocket.org/echo.html` oraz `https://websocket.org/echo.html`.

1. Napisz program klienta, który nawiąże połączenie (handshake) z serwerem obsługującym protokół WebSocket, działającym pod adresem `ws://echo.websocket.org` na porcie 80.

2. Napisz program klienta, który nawiąże połączenie (handshake) z serwerem obsługującym protokół WebSocket, działającym pod adresem `ws://echo.websocket.org` na porcie 80, a następnie, po nawiązaniu połączenia, wyśle do niego krótką (nie dłuższą niż 125 bajtów) wiadomość tekstową.

3. Napisz program klienta, który nawiąże połączenie (handshake) z serwerem obsługującym protokół WebSocket, działającym pod adresem `ws://echo.websocket.org` na porcie 80, a następnie, po nawiązaniu połączenia, wyśle do niego wiadomość tekstową o dowolnej długości.

4. Napisz program serwera, który działając pod adresem 127.0.0.1 oraz na określonym porcie TCP będzie obsługiwał protokół WebSocket. Możesz ograniczyć się do wysyłania/odbierania danych w postaci tekstowej.