Agnieszka Chodorek, Robert R. Chodorek

Współpraca klienta sygnalizacji z prostym serwerem sygnalizacyjnym

JEL: L97 DOI: 10.24136/atest.2019.079
Data zgłoszenia: 15.12.2018 Data akceptacji: 08.02.2019

W artykule przedstawiono budowę klienta sygnalizacji współpracującego z prostym serwerem sygnalizacyjnym. Klient taki może być prostym klientem, wymagającym jedynie podstawowej sygnalizacji, lub bardziej złożonym. Jako przykład systemu o większym stopniu złożoności pokazano aplikację zgodną z architekturą WebRTC. W artykule przedstawiona została przykładowa komunikacja pomiędzy klientem i serwerem zebrana podczas sesji WebRTC. Artykuł zawiera również przykłady dwóch rozwiązań aplikacji czatu internetowego, które wykorzystują ten sam interfejs użytkownika oraz ten sam system sygnalizacji (drugie rozwiązanie rozszerza pierwsze o obsługę komunikatów sygnalizacyjnych techniki WebRTC), różnią się natomiast sposobem transmisji danych użytkownika (wiadomości tekstowych czatu). W pierwszym rozwiązaniu dane użytkownika przenoszone są kanałem sygnalizacyjnym (via serwer sygnalizacyjny) z wykorzystaniem interfejsu WebSocket. W drugim rozwiązaniu dane użytkownika przenoszone są bezpośrednio pomiędzy przeglądarkami (kanałem WebRTC przeznaczonym do transmisji danych niemedialnych) z wykorzystaniem interfejsu RTCDataChannel techniki WebRTC.

Słowa kluczowe: czat internetowy, klient-serwer, klient serwera sygnalizacyjnego, JavaScript, nawiązanie sesji, WebRTC.

Wstęp

Technika WebRTC (ang. *Web Real-Time Communications*) [1][2] umożliwia budowę systemów natywnej komunikacji multimedialnej w przeglądarkach WWW (ang. *World Wide Web*). Transmisja mediów i informacji niemedialnej pomiędzy systemami końcowymi (przeglądarkami) jest realizowana bezpośrednio (w relacji przeglądarka-do-przeglądarki, bez systemu pośredniczącego) [7]. Przesyłać można dane medialne (ang. *media data*, czyli wideo i/lub audio) w czasie rzeczywistym oraz dane niemedialne (ang. *nonmedia data*) bez zachowania warunków czasu rzeczywistego. Taka architektura upraszcza budowę systemu, zmniejsza opóźnienia transmisyjne i uniezależnia transmisję od systemów pośredniczących. Pozwala ona na budowanie aplikacji komunikacyjnych zintegrowanych ze stroną (np. strona służąca do prezentacji danego produktu pozwala na komunikację głosową z konsultantem) [10]. Technika WebRTC wykorzystywana bywa również do transmisji danych medialnych i niemedialnych z urządzeń IoT (ang. *Internet of Things*).

Jak w wielu innych systemach telekomunikacyjnych, tak i tutaj wymiana danych użytkownika pomiędzy systemami końcowymi wymaga zestawienia połączenia, kontrolowania jego przebiegu oraz zakończenia. Pociąga to za sobą konieczność zapewnienia niezawodnej transmisji informacji kontrolno-sterującej (sygnalizacyjnej). WebRTC implementuje podstawy sygnalizacji (jak np. predefiniowane typy komunikatów protokołu SDP, ang. *Session Description Protocol*). Jednak ze względu na spodziewaną dużą różnorodność zastosowań WebRTC, twórcy techniki nie wyposażyli jej we wbudowane podsystemy klienta i serwera sygnalizacji. Podsystemy takie, gdyby zostały zdefiniowane w WebRTC API (ang. *Application Programming Interface*), musiałby spełniać szereg różnych, często sprzecznych ze sobą, wymagań stawianych przez poszczególne aplikacje. Aby zapewnić elastyczność systemu wykorzystującego technikę WebRTC, każdy z systemów zbudowanych w tej technice posiada własne rozwiązania klienta i serwera sygnalizacji.

Klient sygnalizacji (pierwszy system końcowy) łącząc się z serwerem zestawia pierwszy fragment (od siebie do serwera) połączenia sygnalizacyjnego. Drugi fragment połączenia sygnalizacyjnego (również od siebie do serwera) zestawia drugi system końcowy. Fragmenty zestawiane są przez przeglądarki w sposób asynchroniczny, a komunikacja sygnalizacyjna jest możliwa dopiero po ustanowieniu obu fragmentów połączenia. Ponieważ na etapie ustanawiania połączenia nie działają jeszcze specjalizowane interfejsy WebRTC, klient sygnalizacji wykorzystuje do transmisji metodę komunikacji bezpośredniej - interfejs gniazd WebSocket [4] lub (zalecane) Secure WebSocket.

Serwer sygnalizacyjny może posiadać bardzo prostą funkcjonalność, przenosząc jedynie komunikaty pomiędzy przeglądarkami w ramach jednej sesji WebRTC. Może też mieć postać złożonego systemu, zarządzającego wieloma połączeniami konferencyjnymi. Serwery sygnalizacyjne często są budowane z wykorzystaniem środowiska uruchomieniowego node.js. Środowisko to można uruchamiać na komputerze PC (ang. *personal computer*), na wydajnych serwerach wieloprocesorowych oraz w środowisku chmury. Dzięki temu rozwiązanie takie jest w pełni skalowalne.

Niniejszy artykuł omawia założenia projektowe oraz przykładowe implementacje klienta sygnalizacji współpracującego z serwerem sygnalizacyjnym [3] uruchamianym w środowisku node.js. Serwer ten jest rodzajem stacji przekaźnikowej. Odbiera dane przesłane mu przez jeden z systemów końcowych i wysyła je do drugiego systemu końcowego.

Dalsza część artykułu składa się z czterech rozdziałów. W rozdziale 1 zostały przedstawione założenia projektowe i bazowa implementacja klienta sygnalizacyjnego współpracującego z serwerem sygnalizacyjnym. Rozdział 2 prezentuje przykładową komunikację pomiędzy klientem sygnalizacji i serwerem sygnalizacyjnym, zebraną za pomocą oprogramowania narzędziowego WireShark podczas eksperymentalnej sesji WebRTC. Rozdział 3 omawia budowę czatu internetowego: wspólnego interfejsu użytkownika oraz dwóch, alternatywnych implementacji podsystemu komunikacyjnego (realizujących transmisję krótkich tekstów za pośrednictwem serwera komunikacyjnego oraz bezpośrednio pomiędzy przeglądarkami). Ostatni rozdział podsumowuje artykuł.

1.Klient serwera sygnalizacyjnego

Aby możliwe było ustanowienie połączenia WebRTC, przeglądarki muszą najpierw wymienić się informacjami sesyjnymi (danymi sygnalizacyjnymi). Informacje te pozyskiwane są lokalnie przez przeglądarkę, następnie zapisywane są tekstowo w formacie wiadomości protokołu SDP, po czym przesyłane są do drugiej przeglądarki za pośrednictwem serwera sygnalizacyjnego.

Część danych sygnalizacyjnych przeglądarki mogą pozyskać lokalnie, inne są dostępne dopiero po udostępnieniu mediów lokalnych przeglądarce. Adresy IP (ang. *Internet Protocol*) i numery portów mogłyby zostać pozyskane lokalnie, jednakże obecnie (gdy typowy użytkownik Internetu korzysta z adresacji prywatnej IP i translatora adresów NAT, ang. *Network Address Translation*) przeglądarki najczęściej pozyskają jedynie swoje prywatne adresy. To

nie wystarczy, by połączyć się ze zdalnym systemem znajdującym się w innej sieci, i przeglądarki muszą skorzystać z serwera ICE (ang. *Interactive Connectivity Establishment*).

1.1. Ogólna budowa klienta sygnalizacji

Przesyłanie informacji sygnalizacyjnych pomiędzy przeglądarka a serwerem sygnalizacji wymaga bezpośredniej, dwukierunkowej komunikacji. W proponowanym rozwiązaniu klienta sygnalizacyjnego zastosowana została metoda znana jako WebSocket [4], która zapewnia bezpośrednią, dwukierunkową komunikację klient-serwer przy wykorzystaniu pojedynczego połączenia TCP (ang. *Transmission Control Protocol*). WebSocket wykorzystuje protokół HTTP (ang. *Hypertext Transfer Protocol*) jako warstwę transportową[1] - sieć podkładowa pracuje w stosie protokołowym HTTP/TCP/IP. WebSocket API [5] jest obecnie zaimplementowany we wszystkich znaczących przeglądarkach internetowych.

W systemie sygnalizacyjnym serwer oczekuje na podłączenie się klientów sygnalizacji (serwer otwiera połączenie w trybie pasywnym). Aktywne otwarcie połączenia jest realizowane przez systemy klienckie. Gdy systemy klienckie podłączą się do serwera, serwer pośredniczy w przesyłaniu informacji między nimi.

W implementacji serwera sygnalizacyjnego wykorzystana została biblioteka `socket.io`, dodająca do pakietów pewne dodatkowe metadane [4]. Z tego względu klient sygnalizacyjny do współpracy z serwerem sygnalizacyjnym potrzebuje biblioteki `socket.io.js`. Na Rys. 1a zademonstrowano dołączenie tej biblioteki po stronie klienta (w kodzie HTML strony).

Kod języka JavaScript klienta sygnalizacji rozpoczyna się podłączeniem klienta do serwera sygnalizacyjnego. Pierwszą instrukcją przykładowego kodu tej części (Rys. 1b) jest deklaracja adresu serwera sygnalizacyjnego. W przykładzie, adres serwera sygnalizacyjnego przechowywany jest w zmiennej `adres`.

Podłączenie do serwera sygnalizacyjnego wiąże się z powołaniem nowej instancji (tu: `socket`) obiektu `io`. Jako parametr inicjujący tę instancję podawany jest URL (ang. *Uniform Resource Locator*) serwera sygnalizacyjnego, zawierający adres IP serwera sygnalizacyjnego i wykorzystywany przez niego port. Ponieważ wykorzystywany jest protokół WebSocket, URL rozpoczyna się od prefiksu protokołu 'ws://'. W przypadku wersji szyfrowanej WebSocket byłby to prefiks "wss://" [8].

Rys. 1c zawiera podstawowy kod klienta sygnalizacji. Na rysunku rozpoczyna się on funkcją `wysylanie`, służącą do wysyłania wiadomości do serwera sygnalizacyjnego. W tym celu wywoływana jest metoda `emit` obiektu `io` (tu: `socket.emit`) z dwoma parametrami: identyfikatorem zdarzenia (tu: `'komunikat'`) i przenoszoną wiadomością `'wiad'`. Wysłana wiadomość jest przesyłana kontrolnie na konsolę przeglądarki internetowej (konsolę można wywołać w przeglądarkach klawiszem F12).

Nawiązanie połączenie z serwerem kontroluje zmienna `gotowy` (wartość początkowa `false`). Klient, który już nawiązał połączenie z serwerem wysyła do drugiego systemu końcowego komunikat `'gotowy'`, sygnalizując swoją gotowość do pracy.

Odbiór komunikatów odbywa się poprzez metodę `on` obiektu `io` (tu: `socket.on`). Podobnie jak w serwerze, w wyrażeniu funkcyjnym określana jest reakcja serwera na zdarzenia związane z określonym identyfikatorem (tu: `'komunikat'`). Odebrana wiadomość jest wyświetlana na konsoli przeglądarki internetowej. W dalszej części skryptu, przy wykorzystaniu instrukcji warunkowych, definiowana jest reakcja na konkretną wiadomość.

a)

```
<script src='socket.io.js'></script>
```

b)

```
// Deklaracja adresu serwera sygnalizacyjnego
var adres = '192.168.8.100';
// Polaczenie do serwera sygnalizacyjnego
// podane adres IP (adres) i port (5000)
var socket =
io.connect('ws://'+adres+':5000');
```

c)

```
function wysylanie(wiad){
  socket.emit('komunikat', wiad);
  console.log('Wysłany komunikat:', wiad);
}
// po zainicjowaniu połączenia z serwerem
// sygnalizacyjnym wysyłany jest komunikat
// 'gotowy'
var gotowy = false;
wysylanie('gotowy');
// Metoda odbiera komunikaty 'komunikat'
// i je przetwarza
socket.on('komunikat', function (wiad){
  console.log('Odebrany komunikat:', wiad);
  if (wiad === 'gotowy') {
     if (!gotowy) wysylanie('gotowy');
     gotowy = true;
  } else {
// tu wstawiamy przetwarzanie komunikatu
// innego niż 'gotowy'
  };
});
```

Rys. 1. Przykładowy skrypt klienta sygnalizacyjnego: a) dołączenie biblioteki socket.io.js w kodzie HTML strony, b) podłączenie do serwera sygnalizacyjnego, c) podstawowy kod klienta – funkcje wysyłanie i odbiór komunikatów.

Jeżeli otrzymany komunikat przenosi wiadomość `'gotowy'`, oznacza to, że drugi z klientów podłączył się do serwera sygnalizacyjnego. W tej sytuacji zmienna `gotowy` jest ustawiana na wartość `true` i wysyłana jest wiadomość `'gotowy'` do drugiego systemu. Jest to konieczne, gdyż system, który podłączył się jako pierwszy, wysłał już wprawdzie komunikat `'gotowy'`, jednak należy założyć, że system, który podłączył się jako drugi, nie był w stanie go odebrać (w ogólnym przypadku, jeszcze nie podłączył się do serwera). Ponieważ odebranie wiadomości `'gotowy'` oznacza tylko tyle, że istnieje odbiorca treści WebRTC (medialnych lub niemedialnych), wielokrotne odebranie `'gotowy'` nie przynosi żadnych skutków (pozytywnych ani negatywnych).

1.2. Rozszerzenia klienta sygnalizacji na potrzeby WebRTC

Przetwarzanie innych komunikatów niż podane w rozdziale 1.1 wymaga utworzenia kolejnych wpisów, zarówno sprawdzającego zawartość otrzymanego komunikatu, jak i określającego sposób reakcji na ten komunikat. W ten sposób tworzone są rozszerzenia klienta sygnalizacji, np. realizujące wymagania techniki WebRTC.

W technice WebRTC pomiędzy klientami przesyłane są dane niezbędne do nawiązania połączenia oraz do jego zakończenia. Na rysunku 2 zdefiniowano reakcję na komunikaty powiązane z poszczególnymi etapami ustanawiania i kończenia połączenia medialnego i niemedialnego pomiędzy klientami WebRTC.

Media lokalne (dźwięk z mikrofonu, wideo z kamery) nie są automatycznie dostępne dla przeglądarki. Otrzymuje je ona na swoje żądanie, za zgodą użytkownika. Gdy aplikacja uzyska dostęp do mediów lokalnych, wysyła informację o tym fakcie do drugiego systemu (`'mam media'`). Po otrzymaniu komunikatu `'mam media'` od systemu zdalnego, system lokalny może (choć nie musi) zmienić status systemu zdalnego z nieaktywnego (medialnie)

[1] Ponieważ model OSI (ang. *Open System Interconnection*) jest modelem funkcjonalnym, HTTP pracujący na potrzeby WebSockets lokowany jest w warstwie transportowej.

```
function wysylanie(wiad){
  socket.emit('komunikat', wiad);
  console.log('Wysłany komunikat:', wiad);
}

var gotowy = false;
wysylanie('gotowy');

socket.on('komunikat', function (wiad){
  console.log('Odebrany komunikat: '+ wiad);
  if (wiad === 'gotowy') {
     if (!gotowy) wysylanie('gotowy');
     gotowy = true;
  } else if (wiad === 'mam media') {
// w tym miejscu jest definiowana
// reakcja aplikacji WebRTC na dostępne media
  } else if (wiad.type === 'offer') {
// klient sygnalizacyjny otrzymał ofertę SDP
// i przekazuje ją do sesji
  } else if (wiad.type === 'answer') {
// klient sygnalizacyjny ma wysyłać
// swoją ofertę SDP
  } else if (wiad.type === 'candidate') {
// otrzymano dane z protokołu ICE
  } else if (wiad === 'koniec') {
// klient sygnalizacyjny kończy sesję
// i rozłącza połączenia
  } else {
// tu należy wstawić przetwarzanie komunikatów
// innych niż powyższe
  };
});
```

Rys. 2. Szkielet przykładowego skryptu klienta sygnalizacyjnego.

na aktywny. Zmiana statusu może się wiązać, dajmy na to, z uaktywnieniem przycisku lub zmianą koloru (przycisku, symbolu) np. z czerwonego na zielony.

W następnym kroku pomiędzy przeglądarkami nawiązywane jest połączenie medialne i/lub niemedialne. Po zainicjowaniu połączenia przez jeden z systemów wysyłana jest oferta SDP (ang. *SDP offer*). Oferta jest dostarczana przez system sygnalizacyjny do drugiego systemu, który odpowiada na ofertę przesyłając zwrotnie odpowiedź SDP (ang. *SDP answer*).

Wiadomości sygnalizacyjne przesyłane są w obiekcie typu `RTCSessionDescription`, który posiada zdefiniowane pola: `type` i `sdp`. W API WebRTC zdefiniowane zostały predefiniowane typy wiadomości SDP (przenoszone w polu `type`): `'offer'`, `'answer'`, `'pranswer'` i `'rollback'` [9]. Typ `'offer'` przenosi ofertę SDP, a typ `'answer'` odpowiedź na tę ofertę. Typ `'pranswer'` jest używany jako odpowiedź na poprzednią ofertę (gdy system już wcześniej udzielił odpowiedzi) lub jako odpowiedź tymczasowa, po której musi nastąpić odpowiedź `'answer'`. Typ `'rollback'` jest to pusty opis sesji SDP, wymuszający powrót do poprzedniego stabilnego stanu opisu sesji SDP (anuluje bieżącą negocjację SDP). Ponieważ w praktyce wykorzystywane są tylko dwa pierwsze typy wiadomości SDP, kod klienta przedstawiony na rysunku 2 ma zdefiniowana reakcje tylko na wiadomości `'offer'` i `'answer'`.

W polu `sdp` przenoszony jest opis sesji SDP. Pole `sdp` interpretowane jest jako pole typu `DOMString` ciąg znaków zakodowanych zgodnie z UTF-16.

Aby umożliwić przesyłanie pakietów UDP (ang. *User Datagram Protocol*) przez systemy pośredniczące NAT, przeglądarki muszą wymienić się informacjami protokołu ICE .Gdy system lokalny otrzyma informacje z ICE, przekazuje je do systemu zdalnego w komunikacie `'candidate'`. Otrzymując taki komunikat (Rys. 2) klient musi uzyskane informacje przekazać do metody `addIceCandidate` klasy `RTCPeerConnection`.

Komunikat `'koniec'` jest komunikatem kończącym sesję. Po otrzymaniu tego komunikatu od systemu zdalnego, system lokalny powinien wywołać metody odpowiedzialne za rozłączenie połączeń medialnych i/lub niemedialnych.

2.Przykładowa komunikacja

Przykładowe komunikaty przesyłane pomiędzy serwerem a klientem, zebrane z wykorzystaniem oprogramowania Wireshark, przedstawiono na rysunkach 3 i 4. Rysunki obrazują fragmenty realnych komunikatów przesyłanych pomiędzy serwerem sygnalizacyjnym a jednym z klientów.

a)

```
Internet Protocol Version 4, Src:
192.168.8.108, Dst: 192.168.8.100
Transmission Control Protocol, Src Port: 4406,
Dst Port: 5000, Seq: 1, Ack: 1, Len: 620
Hypertext Transfer Protocol
    GET
/socket.io/?EIO=3&transport=websocket&sid=m6S5
_8nMvW0oUEMFAAAA HTTP/1.1\r\n
    Host: 192.168.8.100:5000\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0;
WOW64; rv:49.0) Gecko/20100101 Fire-
fox/49.0\r\n
(...)
    Sec-WebSocket-Version: 13\r\n
    Origin: http://192.168.8.100:8080\r\n
    Sec-WebSocket-Extensions: permessage-
deflate\r\n
    Sec-WebSocket-Key: ScduUILE-
fazCHhz6Qz3QIw==\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
```

b)

```
Internet Protocol Version 4, Src:
192.168.8.100, Dst: 192.168.8.108
Transmission Control Protocol, Src Port: 5000,
Dst Port: 4406, Seq: 1, Ack: 621, Len: 175
Hypertext Transfer Protocol
    HTTP/1.1 101 Switching Protocols\r\n
    Upgrade: websocket\r\n
    Connection: Upgrade\r\n
    Sec-WebSocket-Accept:
eCf+MpgLXRpupQsJ3MGsgvXCNv0=\r\n
    Sec-WebSocket-Extensions: permessage-
deflate\r\n
    \r\n
```

Rys. 3. Przykładowe komunikaty rozpoczynające połączenie, przesyłane pomiędzy klientem i serwerem: a) rozpoczęcie połączenia przez klienta, b) odpowiedź serwera.

Połączenie WebSocket rozpoczyna się w ramach protokołu HTTP. Na rysunku 3 zostały pokazane fragmenty komunikatu rozpoczynającego połączenie, wysyłanego przez klienta sygnalizacyjnego (Rys. 3a) oraz odpowiedź serwera sygnalizacyjnego (akceptacja) na rozpoczęcie połączenia (Rys. 3b). Klient sygnalizacji ma adres IP 192.168.8.108, a serwer sygnalizacyjny adres IP 192.168.8.100. Klient łączy się z serwerem przesyłając żądanie `HTTP GET` (rys. 3a). W żądaniu podany jest URL biblioteki `/socket.io/` z parametrami określającymi, w jaki sposób są przenoszone wiadomości („`transport=websocket`" oznacza, że wykorzystywany jest protokół WebSocket) oraz jaki jest identyfikator sesji (parametr `sid`). W komunikacie `GET` podawany jest parametr HTTP `upgrade`, w którym wskazywany jest protokół

WebSocket (`Upgrade: websocket`) oraz obsługiwana wersja WebSocket (`Sec-WebSocket-Version: 13`). W odpowiedzi serwer akceptuje połączenie (Rys. 3b) podając kod 101 (`Switching Protocols`). Oznacza to poprawne zaakceptowanie połączenia WebSocket.

Rysunek 4 pokazuje typową wymianę danych sygnalizacyjnych za pośrednictwem serwera. Na rysunku 4a komunikat z danymi sygnalizacyjnymi przesyłany jest od serwera do klienta (komunikat pochodzi od systemu zdalnego), a na rysunku 4b od klienta do serwera (komunikat wysyła system lokalny). Dane mogą być przesyłane w formacie tekstowym lub binarnym (określa to pole `Opcode` komunikatu WebSocket). W przykładzie pokazanym na rysunku 4 przesyłane są dane tekstowe.

a)

```
Internet Protocol Version 4, Src:
192.168.8.100, Dst: 192.168.8.108
Transmission Control Protocol, Src Port: 5000,
Dst Port: 4406, Seq: 176, Ack: 635, Len: 16
WebSocket
    1... .... = Fin: True
    .100 .... = Reserved: 0x4
    .... 0001 = Opcode: Text (1)
    0... .... = Mask: False
    .000 1110 = Payload length: 14
    Payload
Line-based text data
    2.(\312OJ\005\000\000\000\377\377\002\000
```

b)

```
Internet Protocol Version 4, Src:
192.168.8.108, Dst: 192.168.8.100
Transmission Control Protocol, Src Port: 4406,
Dst Port: 5000, Seq: 635, Ack: 192, Len: 9
WebSocket
    1... .... = Fin: True
    .100 .... = Reserved: 0x4
    .... 0001 = Opcode: Text (1)
    1... .... = Mask: True
    .000 0011 = Payload length: 3
    Masking-Key: 9d83e22f
    Masked payload
    Payload
Line-based text data
    2\005\000
```

Rys. 4. Przykładowe komunikaty z danymi użytkownika przesyłanymi pomiędzy klientem i serwerem: a) komunikat z danymi przesyłanymi z serwera do klienta, b) komunikat z danymi przesyłanymi od klienta do serwera.

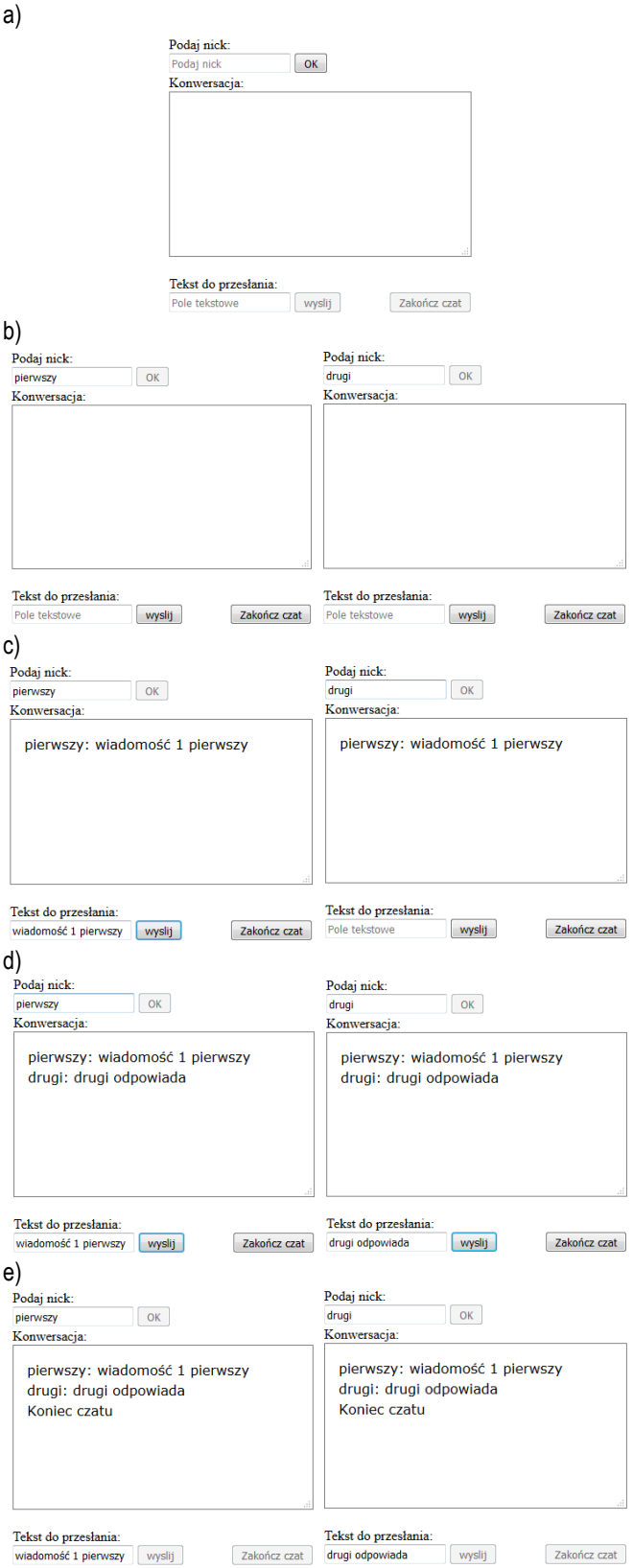
Komunikaty z danymi wysyłane w ramach połączenia WebSocket posiadają dwie formy: bez maskowania (Rys. 4a) i z maskowaniem danych (Rys. 4b). Dane z maskowaniem są to dane przetworzone funkcją XOR z maską (również przesyłaną w komunikacie). W komunikacji WebSocket klient zawsze musi przesyłać dane używając maskowania danych.

3.Czat internetowy jako przykład aplikacji korzystającej z usług serwera sygnalizacyjnego

Czat internetowy umożliwia prostą, interaktywną wymianę wiadomości tekstowych pomiędzy użytkownikami (czasem wzbogaconą o dodatkowe elementy, np. graficzne). Czat może korzystać z serwera sygnalizacyjnego na dwa sposoby: albo wyłącznie do celów sygnalizacyjnych, albo do przenoszenia sygnalizacji i (poniekąd niezgodnie z przeznaczeniem) danych użytkownika.

3.1. Aplikacja czatu internetowego

Przykładowa aplikacja czatu internetowego została zaimplementowana z użyciem języków HTML5, CSS (ang. *Cascading Style Sheets*) i JavaScript. Pozwala ona na wymianę krótkich wiadomości tekstowych. Interfejs użytkownika aplikacji został pokazany na rysunku 5. Zawiera on trzy okna tekstowe (z czego dwa to okna dialogowe, służące do wprowadzania danych) oraz trzy przyciski.



Rys. 5. Interfejs użytkownika aplikacji czatu internetowego i przykładowa wymiana wiadomości: a) ekran startowy, b) aplikacja uruchomiona w dwóch przeglądarkach, podane nicki użytkowników, c) pierwszy użytkownik wysłał komunikat, d) drugi użytkownik odpowiedział na komunikat, e) zakończenie czatu.

Użytkownik pobiera z serwera WWW stronę z wbudowanym kodem aplikacji czatu internetowego. Tuż po uruchomieniu aplikacji jest aktywny tylko przycisk „OK" zatwierdzający pseudonim internetowy (tzw. nick) użytkownika (rys. 5a). Użytkownik podaje swój nick, zatwierdza go i oczekuje na przyłączenie się drugiego użytkownika. Przycisk „OK" staje się nieaktywny, nie pozwalając użytkownikowi na zmianę pseudonimu podczas konwersacji. Drugi przycisk, „Wyślij", staje się aktywny dopiero po podłączeniu się drugiego użytkownika (Rys. 5b). Od tej pory możliwa jest wymiana komunikatów pomiędzy użytkownikami. Wówczas też uaktywnia się trzeci przycisk, „Zakończ czat", kończący połączenia sygnalizacyjne (via serwer sygnalizacji) i transmisji danych (via serwer lub bezpośrednio pomiędzy przeglądarkami).

Komunikaty (zarówno komunikat przychodzący od systemu zdalnego, jak i echo własnego komunikatu) wyświetlane są w oknie tekstowym umieszczonym w centralnej części strony, w kolejności ich pojawienia się w danym systemie komputerowym. Teksty komunikatów opatrzone są nickiem ich twórcy. Rysunki 5b, 5c i 5d zawierają przykładowe widoki ekranów użytkownika, który podłączył się jako pierwszy (po lewej) i użytkownika, który podłączył się jako drugi (po prawej). Dla uproszczenia, użytkownicy przyjęli pseudonimy *pierwszy* i *drugi*.

```
  Podaj nick:
  <br>
  <input type="text" name="nick1"
        id="nick1" placeholder="Podaj nick">
  <button id="nick2">OK</button>
  <br>
  Konwersacja:
  <br>
  <textarea id="wy1" name="wy1"
     cols="35" rows="6"
     style="padding: 1em;
     border: 0.062em solid #777;
     background-color: #fff;
     overflow: auto;
     direction: ltl;
     text-align: left;
     white-space: pre;
     font: 1em/150% verdana, arial,
        helve-tica, sans-serif;"
     readonly="readonly">
   </textarea>   <br>
  <br>
  Podaj tekst do przesłania:
  <br>
  <input type="text" name="we1" id="we1"
      placeholder="Wpisz tekst do przesłania">
  <button id="wyslij">wyslij</button>
  <button id="koniec"
      style="position:absolute; left:
      20em;">Zakończ czat</button>
<br>
```

Rys. 6. Kod aplikacji czatu internetowego - interfejs użytkownika w języku HTML.

Interfejs użytkownika aplikacji czatu internetowego zbudowany został przy wykorzystaniu języka HTML (Rys. 6). Dane podawane przez użytkownika (nick użytkownika i tekst konwersacji wysyłany do drugiego systemu) wprowadzane są poprzez dwa pola tekstowe – odpowiednio: `nick1` i `we1`. Z polami tymi podwiązano dwa przyciski, które wywołują odpowiednie akcje w aplikacji czatu internetowego. Treść całej konwersacji (przesyłane wiadomości wraz z nickami użytkowników) wyświetlane są w obszarze tekstowym `wy1` (por. Rys. 5c, 5d). Konwersja jest kończona poprzez dowolnego z użytkowników poprzez naciśnięcie przycisku „Zakończ czat". Po zakończeniu konwersacji w obszarze tekstowym wyświetlany jest komunikat „Koniec czatu" (por. Rys. 5e).

Obszar tekstowy `wy1` ma zadany rozmiar, który widoczny jest bez przewijania tu: (Rys. 6) 35 kolumn i 6 wierszy. We współczesnych przeglądarkach możliwa jest dynamiczna zmiana rozmiaru obszaru tekstowego przez użytkownika za pomocą znacznika umieszczonego w prawym dolnym rogu. Obszar tekstowy został sformatowany przy wykorzystania języka arkusza stylów CSS (ang. *Cascading Style Sheets*) zdefiniowanego jako parametr `style` polecenia `textarea`. Parametry podane w CSS mają priorytet nad ustawieniami podawanymi w podstawowej definicji obszaru tekstowego np. wymiarami ustawionymi przez `rows` i `cols`. Istotnym parametrem podanym w CSS jest `white-space: pre`, który umożliwia przejście do nowej linii czatu po wysłaniu standardowej sekwencji znaków końca linii.

```
<script>
var nickPole =
    document.getElementById('nick1');
var polewejsciowe =
    document.getElementById('we1');
var polewyjsciowe =
    document.getElementById("wy1");

var nickPrzycisk =
    document.getElementById('nick2');
nickPrzycisk.disabled = false;
nickPrzycisk.onclick = nickfunkcja;

var wyslijPrzycisk =
    document.getElementById('wyslij');
wyslijPrzycisk.disabled = true;
wyslijPrzycisk.onclick = wyslijfunkcja;

var koniecPrzycisk =
    document.getElementById('koniec');
koniecPrzycisk.disabled = true;
koniecPrzycisk.onclick = koniecwyslij;

var nowalinia = String.fromCharCode(10);
var nick = '';

function nickfunkcja(){
  nickPrzycisk.disabled = true;
  var nick = nick1.value;
}

var znaki = '';
var dlLini = 20;

function wyslijfunkcja(){
  wysylanietekstu(nick1.value + ': ' +
            polewejsciowe.value);
  znaki += nick1.value + ': ' +
           polewejsciowe.value + nowalinia ;
  polewyjsciowe.textContent = znaki;
}
function koniecwyslij() {
  wysylanie('koniec');
}
</script>
```

Rys. 7. Kod aplikacji czatu internetowego – obsługa podstawowych elementów strony w języku JavaScript.

Obsługa podstawowych elementów interfejsu użytkownika jest realizowana z wykorzystaniem języka JavaScript (Rys. 7). Zdefiniowane zostały tutaj, m.in., funkcje obsługi przycisków „OK", „Wyślij" i „Zakończ czat" (odpowiednio: `nickfunkcja`, `wyslijfunkcja` i `koniecwyslij`) oraz określono, które przyciski w danej sytuacji są aktywne. Przykładowo, na początku uaktywniany jest tylko przycisk umożliwiający podanie nicka (zmiennej `nickPrzycisk.disabled` przyjmuje wartość `false`).

Funkcja nickfunkcja pobiera wartość podaną w polu nick1 i wyłącza przycisk służący do akceptacji nicka. Po nawiązaniu połączenia z drugą przeglądarką uaktywniany jest przycisk „Wyślij" w funkcji socket.on odbierającej komunikaty sygnalizacyjne (opisanej w rozdziale 3.2 i 3.3). Funkcja wyslijfunkcja wywołuje funkcję wysyłającą tekst wysylanietekstu (przedstawioną w rozdziale 3.2 i 3.3), a następnie aktualizuje obszar tekstowy, w którym prezentowana jest konwersacja. Funkcja koniecwyslij wywołuje funkcję wysyłającą komunikat sygnalizacyjny 'koniec'.

3.2. Komunikacja między przeglądarkami: WebSockets

Do przesyłania wiadomości czatu między przeglądarkami można wykorzystać kanał sygnalizacyjny. Przenoszenie prostych danych użytkownika za pomocą systemu sygnalizacyjnego ma miejsce np. w telefonii komórkowej, gdzie kanałem sygnalizacyjnym przesyłane są wiadomości SMS (ang. *Short Message Service*) lub MMS (ang. *Multimedia Messaging Service*). Wersja czatu opisana w niniejszym podrozdziale przesyła wiadomości czatu przez serwer sygnalizacyjny. Wersja ta korzysta z interfejsu WebSocket.

a)

```
function wysylanietekstu(tekstczatu){
    wysylanie({
      typ: 'czattekst',
      tresc: tekstczatu});
}
```

b)

```
socket.on('komunikat', function (wiad){
  console.log('Odebrany komunikat: '+ wiad);
  if (wiad === 'gotowy') {
    wyslijPrzycisk.disabled = false;
    koniecPrzycisk.disabled = false;
    if (!gotowy) wysylanie('gotowy');
      gotowy = true;
  } else if (wiad.typ === 'czattekst') {
// wyświetlenie otrzymanej
// wiadomosci w polu wy1
    console.log("odebrane: " + wiad.tresc);
    znaki += wiad.tresc + nowalinia;
    polewyjsciowe.textContent = znaki;
  } else if (wiad === 'koniec') {
// klient sygnalizacyjny kończy sesję
    znaki += 'Koniec czatu' + nowalinia;
    wyjscie.textContent = znaki;
    koniecPrzycisk.disabled = true;
    wyslijPrzycisk.disabled = true;
    gotowy = false;
    console.log('koniec połączenia');
  } else {
    console.log('nieznany typ wiadomosci');
  };
});
```

Rys. 8. Implementacja komunikacji między przeglądarkami w aplikacji z wykorzystaniem WebSockets: a) implementacja funkcji wysylanietekstu, b) implementacja funkcji odbierającej komunikaty sygnalizacyjne.

Użycie serwera sygnalizacyjnego do przesłania danych użytkownika wymaga rozszerzenia funkcjonalności serwera w ramach tego samego połączenia sygnalizacyjnego. Można to zrobić wysyłając jako wiadomość sygnalizacyjną nie prostą zmienną wiad (jak ma to miejsce np. w przypadku wiadomości 'gotowy'), ale strukturę wiad. Struktura ta musi zapewniać podstawową funkcjonalność wiadomości sygnalizacyjnej (czyli identyfikować typ komunikatu) oraz dodatkowo przechowywać dane przeznaczone do wysłania kanałem sygnalizacyjnym. Przykład takiej struktury pokazany został na rysunku 8a. Zawiera ona dwa pola: typ i tresc. W polu typ zapisywany jest ciąg znaków identyfikujący komunikat przenoszący wiadomość tekstową – 'czattekst'. Treść wiadomości przesyłana jest w polu tresc.

Wysyłanie wiadomości tekstowych realizuje funkcja wysylanietekstu (Rys. 8a), która tworzy komunikat (przygotowuje pola typ i tresc) i wysyła go za pomocą funkcji wysylanie, przedstawionej w rozdziale 1.1. Struktura wiad jest wysyłana w postaci łańcucha znaków. W JavaScripcie automatycznie nastąpi serializacja takiej struktury u nadawcy i właściwe jej poskładanie u odbiorcy. W efekcie pomiędzy systemami jest przesyłany strumień bajtowy, który w przypadku 'gotowy' był mapowany na łańcuch znaków, a teraz jest mapowany na strukturę zawierającą dwie zmienne typu łańcucha znaków: typ i tresc.

Odbiór komunikatów tekstowych (Rys. 8b) wykorzystuje metodę on obiektu io (patrz rozdział 1). Podobnie, jak w rozdziale 1, podłączenie się uczestnika czatu do serwera sygnalizowane jest za pomocą wiadomości 'gotowy'. Po odebraniu takiego komunikatu uaktywniane są przyciski „Wyślij" i „Zakończ czat" (zmiennym wyslijPrzycisk.disabled i koniecPrzycisk.disabled przypisywana jest wartość false), a zmienna gotowy przyjmuje wartość true. Przypomnijmy, że ustawienie gotowy na true oznacza, że obydwaj uczestnicy czatu są dołączeni do serwera sygnalizacyjnego i możliwa jest wymiana między nimi komunikatów tekstowych.

Wiadomość 'czattekst' sygnalizuje odbiór kolejnej porcji treści czatu. Po odebraniu takiego komunikatu, treść przenoszona w komunikacie jest dołączana do dotychczas odebranych wiadomości czatu (zmienna znaki zawiera zawartość całego czatu) i aktualizowany jest obszar tekstowy wy1 (reprezentowany przez polewyjsciowe.textContent).

Wiadomość 'koniec' sygnalizuje zakończenie pracy czatu. W najprostszym przypadku wystarczy wówczas poinformować użytkownika o zakończeniu czatu (np. wyświetlając w obszarze tekstowym napis „Koniec czatu") i zablokować wszystkie aktywne przyciski (zob. Rys. 8b). Dobrą praktyką jest również ustawienie zmiennej gotowy na false. W przykładzie pokazanym na rysunku 8b nie jest zamykany kanał sygnalizacyjny (klient nie zamyka socket.io). Można to zrobić wywołując od strony klienta funkcję socket.close().

3.3. Komunikacja między przeglądarkami: WebRTC

Technika WebRTC wymaga wymiany informacji sesyjnej kanałem sygnalizacyjnym prowadzącym przez serwer sygnalizacyjny. Informacja sesyjna obejmuje ofertę SDP, odpowiedź SDP oraz dane pozyskane z ICE, przesyłane w osobnej wiadomości zapisanej w formacie SDP (patrz rozdział 1.2). Oznacza to, że aplikacja czatu korzystająca z WebRTC musi, oprócz wysłania opisanego w rozdziale 3.2 komunikatu 'gotowy', również przygotowywać i wysyłać wychodzące oraz obsługiwać przychodzące wiadomości SDP i ICE. Przykładowy kod realizujący te zadania został pokazany na rysunku 9.

Terminal rozpoczynający czat jako pierwszy wysyła ofertę SDP wywołując funkcję przygotujOferte, natomiast drugi terminal wysyła swój opis sesji jako odpowiedź SDP, wywołując funkcję przygotujOdpowiedz (Rys. 9a). Kod zawiera również funkcję blad_sygnalizacji obsługującą błędy (Rys. 9b).

Dla prawidłowego funkcjonowania wewnętrznego automatu stanów opisującego sesję WebRTC niezbędne jest jednokrotne wygenerowanie zdarzeń związanych z SDP. Starsze implementacje WebRTC ignorowały nadmiarowe oferty i odpowiedzi SDP, nowsze sygnalizują błąd. Aby uniknąć takich sytuacji, uczestnicy czatu w sposób jednoznaczny określają, który z nich jest pierwszym, a który

a)

```
var gotowy = false;
var rozpoczete = false;
wysylanie('drugi');
wysylanie('gotowy');
socket.on('komunikat', function (wiad){
  console.log('Odebrany komunikat: '+ wiad);
  if (wiad === 'gotowy'){
    wyslijPrzycisk.disabled = false;
    if (!gotowy) wysylanie('gotowy');
     gotowy = true;
    if (!rozpoczete)
      startkanal();
    else
      przygotujOferte();
  } else  if (wiad === 'drugi'){
    startkanal();
  } else if (wiad.type === 'offer' && gotowy){
    console.log('SDP odebrane typ offer:\n' +
                wiad.sdp);
    poldane.setRemoteDescription(new
            RTCSessionDescription(wiad));
    przygotujOdpowiedz();
  } else if (wiad.type === 'answer' &&
             gotowy){
    console.log('SDP answer:\n' + wiad.sdp);
    poldane.setRemoteDescription(new
            RTCSessionDescription(wiad));
  } else if (wiad.type === 'candidate' &&
             gotowy){
    var candidate = new RTCIceCandidate(
          {sdpMLineIndex:wiad.label,
             candidate:wiad.candidate});
    poldane.addIceCandidate(candidate);
  } else if (wiad === 'koniec') {
    if (poldane != null)
      poldane.close();
    poldane = null;
    znaki += 'Koniec czatu' + nowalinia;
    wyjscie.textContent = znaki;
    koniecPrzycisk.disabled = true;
    wyslijPrzycisk.disabled = true;
    gotowy = false;
    console.log('koniec połączenia');
  } else {
    console.log('inny komunikat'+wiad);
  };
});
```

b)

```
// Funkcja przygotowuje ofertę SDP
function przygotujOferte() {
  console.log('Przygotowuje ofertę SDP');
  poldane.createOffer(ustaw_wyslij,
                      blad_sygnalizacji);
}
// Funkcja przygotowuje odpowiedz SDP
function przygotujOdpowiedz() {
  console.log('Przygotowuję odpowiedź SDP');
  poldane.createAnswer(ustaw_wyslij,
                       blad_sygnalizacji);
}
// Funkcja obsługująca błędy sygnalizacji
function blad_sygnalizacji(blad) {
       console.log('Błąd sygnalizacji: ' +
                   blad.name);
}
// Funkcja ustawia lokalnie SDP i wysyła
// SDP do systemu zdalnego
function ustaw_wyslij(sessionDescription){
  poldane.setLocalDescription(
                 sessionDescription);
  wysylanie(sessionDescription);
}
```

Rys. 9. Implementacja sygnalizacji do realizacji komunikacji między przeglądarkami w aplikacji z wykorzystaniem metody transmisji danych niemedialnych WebRTC: a) funkcja odbierająca komunikaty sygnalizacyjne, b) definicja funkcji obsługujących komunikaty SDP.

a)

```
var serwery =
    {'iceServers':[{'urls':
     'stun:stun.services.mozilla.com'}]};
var konfiguracja = null;
var poldane = new RTCPeerConnection(serwery,
    konfiguracja);
    poldane.onicecandidate =
               obslugaIceCandidate

// funkcja obsługi komunikatów ICE
function obslugaIceCandidate(event) {
  console.log('zdarzenie ICE: '+ event.type);
  if (event.candidate) {
    wysylanie({
      type: 'candidate',
      label: event.candidate.sdpMLineIndex,
      id: event.candidate.sdpMid,
      candidate: event.candidate.candidate});
  } else {
    console.log('Zakończenie procedury ICE');
  }
}
```

b)

```
var kanaldane = null;

function startkanal(){
  rozpoczete = true;
  kanaldane = poldane.createDataChannel(
    "mojkanal", {negotiated: true, id: 0});
  kanaldane.onopen = function(){
    console.log("kanał danych otwarty");
  };
  kanaldane.onmessage = function(event){
    console.log("odebrane: " + event.data);
    znaki += event.data + nowalinia;
    wyjscie.textContent = znaki;
  };
  kanaldane.onclose = function(){
    console.log("kanał danych zamknięty");
    znaki += 'Koniec czatu' + nowalinia;
    wyjscie.textContent = znaki;
    gotowy = false;
    wyslijPrzycisk.disabled = true;
    koniecPrzycisk.disabled = true;
  };
}
```

c)

```
function wysylanietekstu(tekstczatu){
    kanaldane.send(tekstczatu});
}
```

Rys. 10. Implementacja komunikacji między przeglądarkami w aplikacji z wykorzystaniem metody transmisji danych niemedialnych WebRTC: a) funkcja inicjująca połączenie danych niemedialnych wraz z funkcją obsługującą zdarzenia ICE, b) inicjowanie kanału transmisji danych niemedialnych, c) funkcja wysyłające dane czatu.

drugim. Aplikacja, która otrzymuje komunikat `'gotowy'` jest uznawana za tą, która podłączyła się jako druga. Informuje o tym bliźniaczą aplikację komunikatem `'drugi'`. Ponieważ dane użytkowników czatu będą przesyłane bezpośrednim, dedykowanym kanałem transmisji danych niemedialnych, odbiorca komunikatu `'drugi'` otwiera taki kanał, po czym wysyła ofertę SDP. Nadawca komunikatu `'drugi'` odpowiada na ofertę i przyłącza się do otwartego kanału. Kanał danych niemedialnych należy zamknąć po zakończeniu czatu. Obsługa komunikatu `'koniec'` została zatem rozszerzona w stosunku do opisanej w rozdziale 3.2 o wywołanie funkcji `poldane.close` zamykającej kanał.

Do zestawiania połączenia pomiędzy dwoma terminalami WebRTC służy klasa `RTCPeerConnection` (Rys. 10a). Jed-

nym z podstawowych parametrów konstruktora tej klasy jest lista serwerów ICE. Wymagane jest, by połączenie do serwera ICE było szyfrowane (szyfrowanie jest widoczne w definicji połączenia jako identyfikator „`urls`"). Po utworzeniu klasy dodawana jest funkcja obsługi komunikatów ICE (na Rys. 10a jest to `obslugaIceCandidate`).

Połączenia dla danych niemedialnych tworzone są za pomocą metody `createDataChannel` klasy `RTCPeerConnection` (Rys. 10b) [6]. Zgodnie z wymaganiami WebRTC API, dla każdej instancji kanału danych niemedialnych (`kanaldane` na Rys. 10b) należy zdefiniować trzy funkcje obsługujące podstawowe zdarzenia: nawiązano połączenie (`onopen`), odebrano wiadomość (`onmessage`) i następuje zamykanie połączenia (`onclose`). W przykładowej implementacji, pierwsza z funkcji wypisuje na konsoli przeglądarki informację o otwarciu kanału danych. Druga realizuje funkcjonalność identyczną jak reakcja na wiadomość `'czat-tekst'`, a trzecia identyczną jak reakcja na wiadomość `'Koniec czatu'` (por. rozdz. 3.2).

Do wysyłania wiadomości tekstowych do drugiej przeglądarki wykorzystywana jest metoda `send` obiektu `RTCDataChannel` (Rys. 10c).

Dyskusja i podsumowanie

W artykule przedstawiono budowę klienta sygnalizacji, uruchamianego w przeglądarce WWW i współpracującego z serwerem sygnalizacyjnym, na którego bazie można budować aplikacje internetowe. Sposób budowy takiej aplikacji zademonstrowano na przykładzie dwóch rozwiązań czatu internetowego, korzystających z tego samego interfejsu użytkownika i z tego samego sposobu budowy systemu sygnalizacji. Rozwiązania różnią się sposobem transmisji danych czatu: kanałem sygnalizacyjnym via serwer lub bezpośrednio pomiędzy przeglądarkami, kanałem transmisji danych niemedialnych techniki WebRTC.

Pierwsze rozwiązanie charakteryzuje się dużą prostotą, drugie stosunkowo dużym stopniem skomplikowania. Pierwsze jest zatem łatwiejsze w implementacji. Prostota budowy i łatwość implementacji zostały jednak okupione ograniczoną wydajnością i bezpieczeństwem systemu transmisyjnego. Pierwsze rozwiązanie wykorzystuje kanał sygnalizacyjny, który (ze względów wydajnościowych i niezawodnościowych) powinien być słabo obciążony, nie należy zatem zbytnio obciążać go ruchem przenoszącym dane użytkownika. O ile w przypadku niewielkiego czatu takie użycie kanału jest rozsądne i zasadne, o tyle w przypadku usług bardziej obciążających należałoby dokonać separacji kanału danych i kanału sterowania (występuje ona w drugim rozwiązaniu).

W pierwszym rozwiązaniu, korzystającym z interfejsu WebSocket, dane użytkownika nie są szyfrowane. Dane użytkownika przesyłane kanałem danych niemedialnych WebRTC (drugie rozwiązanie) standardowo są zabezpieczone kryptograficznie w relacji end-to-end. Wprawdzie dane w kanale sygnalizacyjnym mogą być szyfrowane (dzięki użyciu *Secure WebSocket*), jednak z samej natury sygnalizacji wynika, że system sygnalizacji zawsze udostępni tekst jawny serwerowi sygnalizacji. Użycie pierwszego rozwiązania, nawet z użyciem bezpiecznego kanału i silnej kryptografii, nie zabezpieczy zatem przed atakiem typu *man-in-the-middle*.

Podsumowując, pierwsze rozwiązanie jest zalecane do budowy mało obciążających aplikacji internetowych, nie wymagających ochrony danych lub wymagających niewielkiej ochrony danych. W zastosowaniach półprofesjonalnych i profesjonalnych, zwłaszcza biznesowych, powinno zostać użyte drugie rozwiązanie. W obu przypadkach (o ile aplikacja nie ma charakteru eksperymentalnego lub demonstracyjnego) do transmisji danych w kanale sygnalizacyjnym preferowane jest użycie połączeń szyfrowanych.

Bibliografia:

1. Loreto S., Romano S.P., Real-Time Communication with WebRTC: Peer-to-Peer in the Browser, O'Reilly Media, Inc. 2014.
2. Ilya G., High Performance Browser Networking, O'Reilly Media, 2013.
3. Chodorek A., Chodorek R. R, Prosty serwer sygnalizacyjny dla techniki WebRTC z wykorzystaniem środowiska uruchomieniowego node.js, „Autobusy: technika, eksploatacja, systemy transportowe" 2018, nr 12.
4. Fette I., Melnikov A., The WebSocket Protocol, RFC 6455, 2011.
5. The WebSocket API, http://www.w3.org/TR/websockets/
6. Chodorek A., Chodorek R. R, Transmisja danych niemedialnych z wykorzystaniem WebRTC, „Autobusy: technika, eksploatacja, systemy transportowe" 2017, nr 6.
7. Loreto S., Romano S.P., How Far are We from WebRTC-1.0? An Update on Standards and a Look at What's Next, „IEEE Communication Magazine" 2017, vol. 55, no. 7.
8. Rescorla E., WebRTC Security Architecture, draft-ietf-rtcweb-security-arch-17, IETF 2018.
9. Uberti J., Jennings C., Rescorla E. (red.), JavaScript Session Establishment Protocol, draft-ietf-rtcweb-jsep-25, IETF 2018.
10. Jansen B., Goodwin T., Gupta V., Kuipers F., Zussman G., Performance Evaluation of WebRTC-based Video Conferencing, „ACM SIGMETRICS Performance Evaluation Review" 2018, vol. 45, no 2.

Co-operation of signalling client with a simple signalling server

A simple signalling server, written in JavaScript and running in node.js run-time environment, was presented in the previous paper of the Authors [3]. In this paper, principles of building of WebRTC signalling client are presented, as well as exemplary communication between the client and the server, captured with the use of WireShark software tool during experimental WebRTC sessions. The paper includes also an example of application of Internet chat, that uses the simple signalling server [3]. The chat was build in two versions, which uses the same user interface and their signalling systems were build according to the same methodology, but differs in method of transmission of chat messages. The first version transmits user data (text messages of the chat) via signalling server, with the use of signalling channel. This version uses WebSocket interface for transmission of chat messages. In the case of the second version, user data are transmitted directly between web browsers, using WebRTC's channel dedicated for transmission of nonmedia data. This version transmits chat messages with the use of WebRTC's RTCDataChannel interface.

Keywords: chat, client-server, JavaScript, session establishment, signalling client, WebRTC.

Autorzy:

dr inż. **Agnieszka Chodorek** – Politechnika Świętokrzyska, Wydział Elektrotechniki, Automatyki i Informatyki, Katedra Elektrotechniki Przemysłowej i Automatyki; 25-314 Kielce; al. Tysiąclecia Państwa Polskiego 7. E-mail: a.chodorek@tu.kielce.pl

dr inż. **Robert R. Chodorek** – AGH Akademia Górniczo-Hutnicza, Wydział Informatyki, Elektroniki i Telekomunikacji, Katedra Telekomunikacji; 30-059 Kraków; Al. A. Mickiewicza 30.
E-mail: chodorek@agh.edu.pl