Agnieszka Chodorek, Robert R. Chodorek

Prosty serwer sygnalizacyjny dla techniki WebRTC z wykorzystaniem środowiska uruchomieniowego node.js

JEL: L96 DOI: 10.24136/atest.2018.515
Data zgłoszenia: 19.11.2018 Data akceptacji: 15.12.2018

Technika WebRTC, bazująca na językach HTML5 i JavaScript, umożliwia natywną transmisję informacji multimedialnej w czasie rzeczywistym pomiędzy przeglądarkami WWW. Chociaż same strumienie mediów, a także transmisje niemedialne (obecne w architekturze WebRTC, choć nie wymagające czasu rzeczywistego), przesyłane są bezpośrednio pomiędzy przeglądarkami, do przesyłania informacji niezbędnych do zarządzania sesją (a konkretniej: nawiązywania sesji) potrzebny jest serwer sygnalizacyjny. Serwer taki może być traktowany jako punkt spotkań dwóch lub więcej systemów końcowych wykorzystujących technikę WebRTC. W artykule przedstawiono zasady budowy takiego serwera za pomocą aplikacji tworzonych w języku JavaScript i uruchamianych w środowisku node.js. Omówiono środowisko node.js oraz pokazano przykład prostego serwera sygnalizacyjnego, budowanego na potrzeby wymiany komunikatów i (lub) inicjowania (wideo)telefonii lub (wideo)konferencji z małą liczbą terminali. Serwer ten łączy uczestników tylko w ramach pojedynczej sesji 1-do-1 lub wielu-do-wielu. Pomimo dużej prostoty, przykładowy serwer jest w pełni funkcjonalnym serwerem sygnalizacyjnym WebRTC, umożliwiającym realizację, między innymi, (wideo)telefonii pomiędzy przeglądarkami używającymi WebRTC.

Słowa kluczowe: klient-serwer, nawiązanie sesji, node.js, serwer sygnalizacyjny, WebRTC.

Wstęp

WebRTC (ang. *Web Real-Time Communications*) [1][2] jest to technika umożliwiająca bezpośrednią, natywną transmisję informacji multimedialnej w czasie rzeczywistym pomiędzy dwiema lub więcej przeglądarkami WWW. Aplikacje WebRTC pisane są w języku HTML5 (interfejs użytkownika) i JavaScript (część komunikacyjna) i uruchamiane w przeglądarce jako strona WWW. Mogą pracować w chmurze [3], współpracować z systemami IoT [4], systemami SIP [5][6]. Skrócony opis WebRTC można znaleźć, m.in., w [7][8].

Transmisja mediów i danych (informacji niemedialnej, ang. *non-media*) pomiędzy systemami końcowymi WebRTC (przeglądarkami WWW - ang. *World Wide Web*) realizowana jest bez udziału jakiegokolwiek systemu pośredniczącego (serwera mediów, serwera proxy, mostka konferencyjnego, itp.), jednakże na etapie zestawiania sesji WebRTC wykorzystywane są trzy typy serwerów: serwer WWW, serwer sygnalizacyjny, serwer ICE.

Użytkownik, który chce nawiązać sesję WebRTC pomiędzy dwiema przeglądarkami, pobiera kod strony z aplikacją z serwera WWW. Podczas wykonywania kodu strony, przeglądarka nawiązuje łączność z serwerem sygnalizacyjnym, po czym oczekuje w stanie nieaktywnym na zgłoszenie się drugiej przeglądarki, która również musi pobrać kod strony i, podczas jego wykonywania, skontaktować się z serwerem sygnalizacyjnym. Pojawienie się drugiej przeglądarki uaktywnia połączenie. Obie przeglądarki wymieniają informację sygnalizacyjną niezbędną do zestawienia sesji (adresy IP - ang. *Internet Protocol*, metod kodowania i kompresji multimediów, parametry strumieni mediów, itp.). Informacja sygnalizacyjna przesyłana jest w formacie protokołu SDP (ang. *Session Description Protocol*), a do przenoszenia komunikatów SDP wykorzystywany jest protokół JSEP (ang. *JavaScript Session Establishment Protocol*).

Na tym etapie pomiędzy przeglądarkami przesyłana jest, m.in., informacja protokołu ICE (ang. *Interactive Connectivity Establishment*). Jest to jeden z trzech protokołów (obok TURN - ang. *Traversal Using Relays around NAT* i STUN - ang. *Simple Traversal of UDP through NATs*) umożliwiających przesyłanie pakietów UDP (ang. *User Datagram Protocol*) przez systemy pośredniczące NAT (ang. *Network Address Translators*). Po wymianie informacji ICE, obie przeglądarki łączą się z serwerem ICE, który wyszukuje pary (adres IP, port UDP) pozwalające na realizację transmisji UDP. Jeśli operacja wyszukiwania zakończy się sukcesem oraz jeśli przeglądarce zostały udostępnione media lokalne [9], przeglądarki zaczynają przesyłać między sobą strumienie audio i wideo. Do transmisji informacji multimedialnej w czasie rzeczywistym używany jest protokół RTP (ang. *Real-time Transport Protocol*), pracujący nad protokołem UDP.

Serwer WWW i serwer sygnalizacyjny mogą być postawione na tej samej fizycznej maszynie. Często oba są budowane z wykorzystaniem środowiska uruchomieniowego node.js. Serwer ICE również może być uruchamiany w środowisku node.js, częściej jednak wykorzystywane są duże, profesjonalne serwery ICE, udostępniane przez firmę Google czy Fundację Mozilla[1].

Niniejszy artykuł koncentruje się na zagadnieniu budowy serwerów sygnalizacyjnych WebRTC z wykorzystaniem środowiska uruchomieniowego node.js. W pierwszym rozdziale została przedstawiona charakterystyka środowiska node.js. Następny rozdział zawiera przykład serwera sygnalizacyjnego budowanego na potrzeby prostej wymiany komunikatów (np. czat internetowy) oraz inicjowania (wideo)telefonii lub wideokonferencji dla małej liczby terminali. Ostatni rozdział zawiera podsumowanie artykułu.

Współcześni użytkownicy Internetu, zwłaszcza ci młodsi, oczekują szybkiego ładowania stron i szybkiego uruchamiania się aplikacji. Duża szybkość działania środowiska uruchomieniowego node.js pozwala na budowanie efektywnych serwerów, zdolnych sprostać tym wymaganiom. Omawiana w artykule prosta, oszczędna konstrukcja serwera pozwala na tworzenie interaktywnych aplikacji wbudowanych w strony WWW nawet osobom nie mającym dużych doświadczeń z architekturą WebRTC. Aplikacje te mogą pracować (np. wideotelefonia) lub nie (np. czat internetowy) w reżimie czasu rzeczywistego.

1. Środowisko uruchomieniowe node.js

JavaScript (JS) to wysokopoziomowy skryptowy język programowania. Opracowany w latach 90. przez firmę Netscape był pierwotnie wykorzystywany do zapewniania interakcji w ramach strony WWW. Początkowo był on używany tylko po stronie klienta. W

[1] Adresy serwerów ICE (stun/turn) dostępne są pod adresem: https://gist.github.com/sagivo/3a4b2f2c7ac6e1b5267c2f1f59ac6c6b. Sprawdzenie działania i dostępności danego serwera ICE może być zrealizowane przez stronę: https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/.

późniejszym okresie pojawiła się możliwość wykorzystania JavaScript także po stronie serwera. Obecnie wszystkie przeglądarki internetowe wspierają JavaScript. Jest on jedną z fundamentalnych technologii wykorzystywanych podczas tworzenia serwisu WWW.

1.1. Ogólna charakterystyka środowiska node.js

Pomimo dużej popularności JavaScript po stronie klienta, po stronie serwera przez wiele lat użycie tego języka było ograniczone, głównie ze względu na problemy implementacyjne. Dopiero pojawienie się w 2009 roku środowiska uruchomieniowego node.js rozwiązało istotną część problemów związanych z wdrażaniem JavaScript po stronie serwera. Środowisko node.js powstało na bazie udostępnionego przez Google mechanizmu V8 języka JavaScript przeznaczonego dla przeglądarki internetowej Chrome[2]. Nie bez znaczenia jest tu fakt, że implementacja języka JavaScript wbudowana w przeglądarkę Chrome była dobrze zoptymalizowana pod kątem ruchu na stronach internetowych.

Niewątpliwą zaletą node.js, będącego de facto serwerową wersją mechanizmu V8 języka JavaScript, jest bardzo dobre dopasowanie środowiska po stronie serwera do środowiska po stronie klienta (w przeglądarce). Dzięki temu, w zależności od wymagań aplikacji i preferencji programisty, te same fragmenty kodu można uruchamiać po stronie klienta lub po stronie serwera.

Node.js bardzo często jest używane do uruchamiania serwerów WWW. Za pomocą tego środowiska budowane są także interfejsy programistyczne aplikacji API (ang. Application Programming Interface) dla wielu usług internetowych (np. REST API - ang. *Representational State Transfer API*). Z wykorzystaniem node.js tworzy się również różnego typu aplikacje internetowe czy komunikację pomiędzy wieloma klientami.

Node.js zawiera implementację języka JavaScript oraz szereg pakietów, które rozszerzają możliwości środowiska o funkcje biblioteczne. Wykonywanie skryptów języka JavaScript możliwe jest:

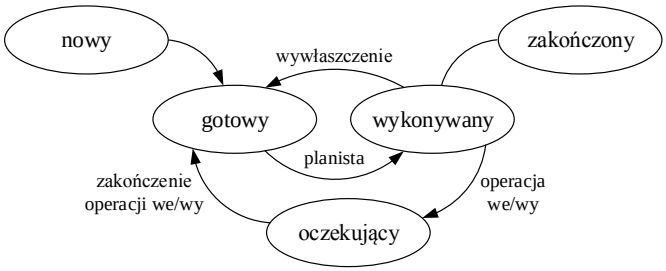
- w trybie konsoli,
- w trybie wsadowym.

Aby pracować w trybie konsoli należy wydać polecenie node bez parametrów wywołania. Następnie, w wierszu poleceń, podawane będą kolejne instrukcje języka JavaScript składające się na kod programu. Aby pracować w trybie wsadowym należy wydać polecenie node z parametrem - nazwą pliku zawierającego kod programu w języku JavaScript.

Środowisko można rozszerzać o dodatkowe moduły (pakiety) NPM (ang. *Node Packaged Module*) przy użyciu npm (ang. *Node Package Manager*). Pakiet NPM to biblioteka w postaci pakietu oprogramowania. Moduły NPM są tworzone przez różne organizacje w celu zapewnienia różnorodnych funkcji, których nie ma w podstawowej instalacji środowiska node.js. Interesującą cechą środowiska jest rejestr NPR[3] (ang. *Node Package Registry*), gdzie rejestrowane są pakiety. Rejestr umożliwia publikowanie własnych pakietów, a także pobieranie pakietów utworzonych przez innych.

1.2. Jednowątkowy model sterowany zdarzeniami

Cechą charakterystyczną środowiska node.js jest uruchamianie aplikacji w jednowątkowym modelu sterowanym zdarzeniami. W klasycznym modelu wykorzystywanym w serwerach, każde żądanie, które trafia do serwera, jest przypisywane do osobnego wątku. Obsługa żądania jest kontynuowana w tym wątku do momentu zrealizowania żądania i wysłania odpowiedzi. Wątki wykonywane są równolegle (o ile każdemu wątkowi przydzielony zostanie jeden rdzeń procesora) lub quasi-równolegle. Nowy wątek po utworzeniu trafia do kolejki gotowości (stan "gotowy" na rys. 1), gdzie oczekuje na wykonywanie (czyli przydział czasu procesora). O przydziale czasu procesora decyduje planista (ang. *scheduler*) systemu operacyjnego. Wątek, który w danej chwili oczekuje na zrealizowanie czasochłonnej operacji (np. operacji wejścia/wyjścia) przechodzi w stan oczekiwania (dzięki czemu nie marnuje czasu procesora na jałowe oczekiwanie). Po zakończeniu tej operacji wątek przechodzi ze stanu oczekiwania do stanu gotowości.



Rys. 1. Automat stanów opisujący proces/wątek w systemie operacyjnym [10].

Środowisko node.js zrywa z modelem klasycznym. Wszystkie żądania są obsługiwane w ramach jednego wątku, unikając w ten sposób przełączania (*de facto* rywalizacji) pomiędzy wątkami realizującymi kolejne żądania. Tym niemniej, zachowany został automat stanów opisujący obsługę żądania (rys. 1), choć zmieniony został poziom abstrakcji (stan jest stanem żądania, a nie wątku).

Żądania trafiające do serwera są wstawiane do kolejki zdarzeń, która pełni podobną funkcję jak kolejka gotowości w systemie operacyjnym. Następnie, w ramach jednego wątku, w pętli zdarzeń pobierane jest żądanie najwyżej położone w kolejce zdarzeń. To zadanie jest wykonywane jako pojedynczy (i jedyny) wątek obsługujący żądanie. Jeżeli pojawi się zdarzenie o dłuższym czasie obsługi (w praktyce oznacza to, że aplikacja wywołuje jakieś funkcje systemowe o dłuższym czasie wykonywania lub ma realizować dostęp do urządzeń wejścia/wyjścia), wykonywanie zdarzenia jest zawieszane, co odpowiada przejściu wątku do stanu "oczekujący". Aby możliwe było przejście ze stanu "oczekujący" do stanu "gotowy", zawieszeniu zdarzenia zawsze towarzyszy utworzenie nowego zdarzenia, które wędruje do kolejki zdarzeń. Jego zadaniem jest wznowienie zawieszonego zdarzenia. Jeżeli funkcja lub operacja będąca przyczyną zawieszenia zdarzenia nie została jeszcze wykonana, również i to zdarzenie jest zawieszane - co pociąga za sobą utworzenie nowego zdarzenia. Zdarzenia można wielokrotnie zagnieżdżać, a warunkiem stopu procedury rekurencyjnej jest zakończenie funkcji lub operacji, która była pierwotną przyczyną zawieszania zdarzeń.

Rozwiązanie zastosowane w node.js pozwala na naturalne kończenie elementarnych czynności. Planista nie przerywa ich w dowolnym miejscu, co zwiększa efektywność wykonywanych zadań.

2. Prosty serwer sygnalizacyjny

W najprostszym ujęciu, serwer sygnalizacyjny jest rodzajem stacji przekaźnikowej, która odbiera dane przesyłane mu przez jeden z systemów końcowych i wysyła je do drugiego systemu końcowego (lub rozgłasza je do pozostałych systemów końcowych). Serwer taki może służyć do wymiany lub do zainicjowania wymiany krótkich komunikatów tekstowych pomiędzy dwiema lub więcej przeglądarkami (np. na zasadzie czatu internetowego). Można go również wykorzystać do zainicjowania komunikacji głosowej pomiędzy dwiema przeglądarkami (telefonia przez Internet) lub pomiędzy wieloma przeglądarkami (telekonferencja przez Internet). Komunikacji głosowej może towarzyszyć transmisja obrazu z lokalnych kamer internetowych (wideotelefonia i wideokonferencja przez Internet).

[2] http://nodejs.org/

[3] Rejestr NPR jest dostępny pod adresem http://npmjs.org/.

Najprostszy serwer łączy jednocześnie uczestników (dwóch lub wielu) tylko jednej sesji multimedialnej. Nie może być zatem używany w charakterze lokalnej centralki telefonicznej czy mostka konferencyjnego. Nie daje również możliwości skorzystania z "pokoi czatów" czy "pokoi rozmów".

2.1. Serwer sygnalizacyjny dla techniki WebRTC

Serwer sygnalizacyjny WebRTC jest punktem spotkań dla sygnalizacji sesyjnej przesyłanej pomiędzy systemami końcowymi WebRTC (przeglądarkami), będącymi klientami sygnalizacji. Początkowo serwer oczekuje na podłączenie się klientów sygnalizacji. Po ich podłączeniu pośredniczy w przesyłaniu informacji między nimi. Wymaga to dwukierunkowej, asynchronicznej komunikacji pomiędzy serwerem a klientem.

Bezpośrednia, dwukierunkowa komunikacja pomiędzy aplikacją internetową uruchomioną w przeglądarce a serwerem możliwa jest z wykorzystaniem klasycznego protokołu HTTP (ang. *Hypertext Transfer Protocol*). Jednakże wykorzystanie tej metody komunikacji w aplikacjach interaktywnych powoduje znaczne obciążenie serwera, komplikuje implementacje aplikacji i daje duży narzut nagłówków. Aby temu przeciwdziałać opracowana została metoda komunikacji bezpośredniej, znana jako WebSocket [11] (dosł. "gniazdo WWW"; nazwa nawiązuje do interfejsu gniazd, ang. socket). WebSocket umożliwia bezpośrednią, dwukierunkową komunikację klient-serwer przy wykorzystaniu pojedynczego połączenia TCP. WebSocket nie korzysta bezpośrednio z protokołu TCP lecz ze stosu protokołowego HTTP/TCP/IP, wykorzystując protokół HTTP jako warstwę transportową. Pozwala to na korzystanie z infrastruktury opracowanej dla protokołu HTTP (m.in. serwerów proxy). Dzięki zastosowaniu interfejsu WebSocket możliwa jest w pełni symetryczna komunikacja klient-serwer. Nie jest ona możliwa przy korzystaniu bezpośrednio z HTTP, gdyż protokół ten nie był projektowany do w pełni dwukierunkowej komunikacji.

Zdefiniowane dla protokołu WebSocket API [12] jest zaimplementowane we wszystkich współczesnych przeglądarkach internetowych. Aby zwiększyć możliwości serwera sygnalizacyjnego dla WebRTC, do jego budowy wykorzystano bibliotekę socket.io[4]. Biblioteka ta wykorzystuje protokół WebSocket dodając pewne dodatkowe metadane do pakietów. Pozwala to m.in. na rozszerzenie komunikacji o dodatkową multipleksację, która umożliwia budowanie złożonych połączeń sygnalizacyjnych dla wielu, jednocześnie podłączonych do serwera klientów.

2.2. Przykład prostego serwera sygnalizacyjnego

Kod przykładowego serwera (Rys. 2) składa się z części zawierającej deklaracje wstępne (Rys. 2a) i podstawowej części serwera (Rys. 2b). W deklaracjach wstępnych występują żądania załadowania wykorzystywanych modułów (metoda `require`). Aby zbudować serwer sygnalizacyjny dla techniki WebRTC konieczna jest instalacja pakietów bibliotek JavaScript: `node-static` [13] i `http`. W kolejnym kroku skrypt pobiera (podany w linii wywołań) adres IP, na którym będzie nasłuchiwał serwer sygnalizacyjny (gdy brak takiego adresu wskazywany jest adres interfejsu loopback – 127.0.0.1). Wykorzystywany adres, w celach kontrolnych, wypisywany jest także na konsoli node.js.

Podstawowa część kodu serwera (Rys. 2b) rozpoczyna się od utworzenia nowej instancja (tu: `ap1`) obiektu `http`. Podczas tworzenia tej instancji należy podać adres IP interfejsu serwera i numer portu, na którym serwer będzie oczekiwał na klientów.

Następnie żądana jest biblioteka socket.io. W podanym przykładzie, wraz z żądaniem załadowania biblioteki socket.io tworzona jest nowa instancja serwera sygnalizacyjnego, który będzie wykorzystywał, zdefiniowaną wcześniej, instancję `ap1` serwera HTTP.

a)

```
// Żądanie biblioteki JavaScript node-static
var static = require('node-static');
// Żądanie biblioteki JavaScript http
var http = require('http');

// funkcja pomocnicza odczytująca parametry
// podane w linii wywołania
var argv = [];
process.argv.forEach(function (val, index,
array) {
   argv[index] = val;
});
// adres IP, na którym będzie nasłuch
// kontrola, czy istnieje parametr wywołania
// jeśli nie: nasłuch na interfejsie loopback
var adres = '127.0.0.1'
if (argv.length > 2) {
  adres = argv[2]
}
// wypisanie adresu serwera na konsoli node.js
console.log('adres serwera: '+adres);
```

b)

```
// Utworzenie instancji serwera z podaniem, na
// którym porcie i adresie serwer nasłuchuje
var ap1 = http.createServer(function (req,
res) { }).listen(5000,adres);

// Żądanie biblioteki JavaScript socket.io
var io = require('socket.io').listen(ap1);

// Zarządzanie połączeniem:
// 'connection' jest to predefiniowany
// komunikat metody io.sockets.on
io.sockets.on('connection', function (socket){
// Obsluga komunikatu 'komunikat'
   socket.on('komunikat', function (wiad) {
// kanał pracuje w trybie rozgłoszeniowym
      socket.broadcast.emit('komunikat',
                            wiad);
      console.log('odebrane: '+wiad);
   });
});
```

Rys. 2. Przykładowy skrypt serwera sygnalizacyjnego: a) deklaracje wstępne, b) podstawowy kod serwera.

W kolejnym etapie oprogramowywane jest zarządzanie połączeniem. Wywołanie metody `io.sockets.on` [14] uruchamia serwer, który oczekuje na określone zdarzenie. Przed uruchomieniem serwera wykonywany jest kod zdefiniowany w wyrażeniu funkcyjnym, który określa reakcję serwera na zdarzenia, które będą miały miejsce podczas wymiany komunikatów. W przykładzie serwer sygnalizacyjny oczekuje na nowe połączenia od klientów (oczekuje na predefiniowany komunikat „connection"). Gdy klient połączy się z serwerem sygnalizacyjnym (odebrany zostanie komunikat „connection") jego połączenie obsługiwane będzie w sposób określony przez metodę `socket.on` obiektu `io`.

W niniejszej implementacji serwera, komunikaty sygnalizacyjne będą posiadały identyfikator 'komunikat'. Ponieważ w komunikacji jest wykorzystywany jeden komunikat, definiowany jest wpis tylko dla tego jednego zdarzenia.

W serwerze sygnalizacyjnym, reakcja na otrzymany komunikat sprowadza się do przesłania go do pozostałych klientów. Zastosowano tu metodę `socket.broadcast.emit`, która wysyła komunikaty w trybie rozgłoszeniowym do wszystkich dołączonych klientów. Komunikat jest również kontrolnie wyświetlany na konsoli serwera sygnalizacyjnego.

[4] https://socket.io/

2.3. Uruchomienie serwera w trybie konsoli

Skrypt serwera może zostać uruchomiony w trybie konsoli lub w trybie wsadowym. Z oczywistych względów częściej stosowana jest ta druga metoda, jednak w przypadku tak nieskomplikowanego skryptu, jak zamieszczony na rysunku 2, użycie trybu konsoli nie będzie stanowiło problemu. Warto zauważyć, że tryb konsoli, choć mniej wygodny, daje większe możliwości, np. debugowania.

Node.js wymaga do pracy określonego środowiska, definiowanego m.in. zmiennymi środowiskowymi systemu. Zmienne środowiskowe ustawiane są automatycznie na etapie instalacji node.js - po zainstalowaniu samego node.js, w danym systemie komputerowym tworzone są skrypty startowe przygotowujące środowisko pracy. Z tego względu należy uruchamiać środowisko node.js używając zainstalowanych skryptów startowych. Przykładowo, w systemie operacyjnym Windows tworzony jest skrót, który uruchamia wiersz poleceń (konsolę) systemu operacyjnego z właściwie ustawionymi zmiennymi środowiskowymi (Rys. 3a). Z tego wiersza poleceń można uruchomić środowisko node.js zarówno w trybie konsoli (Rys. 3a), jak i wsadowym.

a)

```
Your environment has been set up for using Node.js 0.10.32 (x64) and npm.

C:\Users\robert>node
>
```

b)

```
C:\Users\robert>node
> var static = require('node-static');
Error: Cannot find module 'node-static'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.require (module.js:364:17)
    at require (module.js:380:17)
    at repl:1:14
    at REPLServer.self.eval (repl.js:110:21)
    at repl.js:249:20
    at REPLServer.self.eval (repl.js:122:7)
    at Interface.<anonymous> (repl.js:239:12)
    at Interface.emit (events.js:95:17)
> var http = require('http');
undefined
>
```

c)

```
> var static = require('node-static');
undefined
> var http = require('http');
undefined
> var argv = [];
undefined
> process.argv.forEach(function (val, index, array) {
... argv[index] = val;
... });
undefined
> var adres = '127.0.0.1'
undefined
> if (argv.length > 2) {
... adres = argv[2]
... }
undefined
> console.log('adres serwera: '+adres);
adres serwera: 127.0.0.1
undefined
```

d)

```
> var api = http.createServer(function (req,
... res) { }).listen(5000,adres);
undefined
> var io = require('socket.io').listen(api);
undefined
> io.sockets.on('connection', function (socket){
... socket.on('komunikat', function (wiad) {
..... socket.broadcast.emit('komunikat',
..... wiad);
..... console.log('odebrane: '+wiad);
..... });
... });
```

Rys. 3. Uruchamianie przykładowego skryptu serwera sygnalizacyjnego: a) uruchomienie środowiska w trybie konsolowym, b) żądanie biblioteki 'node-static', która nie jest zainstalowana- reakcja na błąd (informacje debugera), c) uruchomienie kodu z rysunku 2a, d) uruchomienie kodu z rysunku 2b.

Po uruchomieniu środowiska node.js w trybie konsoli, system node.js wyświetla znak zachęty (tu: znak większości - por. Rys. 3a). Od tego momentu można podawać kolejne komendy (kod języka JavaScript). Podanie komendy, która ma złą składnię albo odwołuje się do nieistniejących elementów (np. nieistniejącej biblioteki) skutkuje pojawieniem się informacji debugera o błędzie i kontekście wystąpienia tego błędu. Przykładowo, instrukcja "`var static = require('node-static');`" spowodowała błąd (Rys. 3b). Informacja o błędzie pojawia się od razu w następnej linii po instrukcji języka, która spowodowała błąd - napis "`Error: Cannot find module 'node-static'`" wskazuje, że nie został zainstalowany wymagany moduł 'node-static'. Niżej podawany jest kontekst tego błędu.

Poprawne podanie komendy skutkuje wyświetleniem wyniku przeprowadzonej operacji. Paradygmat programowania strukturalnego wymaga, by funkcja zawsze zwracała wartość. Jeżeli dana funkcja języka JavaScript nie zwraca żadnego wyniku (w myśl programowania strukturalnego, jest to procedura), zostaje jej przypisana domyślna wartość 'undefined' - por. instrukcja '`var http = require('http');`' na Rys. 3b.

W przypadku deklarowania złożonych struktur (zajmujących więcej niż jedną linię np. funkcja czy instrukcja warunkowa) pojawia się dodatkowa informacja w postaci kropek na początku linii (Rys. 3c). Przy bardziej złożonych strukturach następuje zmiana liczby kropek wiodących (Rys. 3d). Zależy ona od poziomu zagnieżdżenia danej instrukcji.

```
{ name: '/',
  server:
   { nsps: { '/': [Circular] },
     _path: '/socket.io',
     _serveClient: true,
     _adapter: [Function: Adapter],
     _origins: '*:*',
     sockets: [Circular],
     eio:
      { clients: {},
        clientsCount: 0,
        pingTimeout: 60000,
        pingInterval: 25000,
        upgradeTimeout: 10000,
        maxHttpBufferSize: 100000000,
        transports: [Object],
        allowUpgrades: true,
        allowRequest: [Function],
        cookie: 'io',
        ws: [Object],
        _events: [Object] },
     httpServer:
      { domain: null,
        _events: [Object],
        _maxListeners: 10,
        _connections: 0,
        connections: [Getter/Setter],
        _handle: [Object],
        _usingSlaves: false,
        _slaves: [],
        allowHalfOpen: true,
        httpAllowHalfOpen: false,
        timeout: 120000,
        _connectionKey: '4:127.0.0.1:5000' },
     engine:
      { clients: {},
        clientsCount: 0,
        pingTimeout: 60000,
        pingInterval: 25000,
        upgradeTimeout: 10000,
        maxHttpBufferSize: 100000000,
        transports: [Object],
        allowUpgrades: true,
        allowRequest: [Function],
        cookie: 'io',
        ws: [Object],
        _events: [Object] } },
  sockets: [],
  connected: {},
  fns: [],
  ids: 0,
  acks: {},
  adapter: { nsp: [Circular], rooms: {}, sids: {}, encoder: {} },
  _events: { connection: [Function] } }
>
```

Rys. 4. Informacja debugera o utworzonym obiekcie.

Obiekt `io` jest obiektem bazowym dla serwera sygnalizacyjnego. Aby uzyskać żądaną funkcjonalność serwera sygnalizacyjnego, w obiekcie tym można dodawać nowe i modyfikować istniejące metody. Po utworzeniu obiektu `io`, debuger wbudowany w środowisko node.js wyświetla pełne informacje o nim (Rys. 4).

Aby aplikacja serwera sygnalizacyjnego mogła pełnić funkcję serwera wobec klientów sygnalizacji, zmienna: `_serveClient` musi być ustawiona na wartość `true`. W serwerze można ustawić filtr na połączenia od klientów – można określić, jakie adresy muszą posiadać klienci i/lub na jakich portach mogą nadawać. W uruchomionym serwerze (Rys. 4) akceptowane są połączenia od dowolne-

go klienta – o dowolnym adresie IP i dowolnym numerze portu (zmienna: `_origins: '*:*'`).

W serwerze ustawione są również pewne wartości kontrolno-zabezpieczające. Serwer sprawdza aktywność klientów poprzez cykliczne przesyłanie komunikatów *ping*. Gdy klient nie odpowiada przez pewien czas od wysłania skierowanego do niego komunikatu (tu: czas określony przez zamienną `pingTimeout` na wartość 60000 ms), serwer kończy połączenie. W celu ochrony przed atakiem typu DoS (ang. *Denial-of-Service*) wprowadzony został parametr definiujący ilość danych, jaka może zostać odebrana w ramach sesji. Określa to zmienna `maxHttpBufferSize` (jej wartość domyślna wynosi 100.000.000 bajtów). Po przekroczeniu wartości `maxHttpBufferSize` sesja z danym klientem jest zamykana, aby ochronić serwer przed potencjalnym atakiem.

Rys. 5. Informacja o obiekcie `io` po podłączeniu dwóch klientów.

W wyniku uruchomienia skryptów z rysunku 2 powstał obiekt `io` (Rys. 4), pełniący funkcje serwera dla klientów sygnalizacji. W trakcie pracy serwera sygnalizacyjnego możliwe jest uzyskanie informacji o aktualnie podłączonych klientach. W tym celu należy wyświetlić informacje o obiekcie `io` używając polecenia `console.log(io)`. Wśród otrzymanych informacji są, m.in. (Rys. 5), informacje o identyfikatorach klientów i liczbie klientów (zmienna `clientsCount`, tu równa 2).

Podsumowanie

Technika WebRTC umożliwia transmisję informacji multimedialnej w czasie rzeczywistym pomiędzy przeglądarkami WWW. Strona WWW napisana w języku HTML5 stanowi interfejs użytkownika, a protokoły komunikacyjne warstw od siódmej do górnej podwarstwy warstwy czwartej implementowane są z wykorzystaniem języka JavaScript. Chociaż same strumienie mediów przesyłane są bezpośrednio pomiędzy przeglądarkami, na etapie zestawiania sesji niezbędny jest serwer sygnalizacyjny. Stanowi on swoisty punkt spotkań dwóch lub więcej nieznających się nawzajem systemów końcowych, wykorzystujących technikę WebRTC.

W artykule przedstawiono zasady budowy serwera sygnalizacyjnego z użyciem skryptów języka JavaScript, uruchamianych w środowisku node.js. Pokazana została prosta aplikacja serwera, której działanie zostało zilustrowane przykładem uruchomienia w środowisku node.js w trybie konsoli.

Opisywany serwer sygnalizacyjny, pomimo swej prostoty, pozwala na łączenie się dwóch przeglądarek WWW zlokalizowanych w różnych sieciach (również wtedy, gdy jedna z nich znajduje się za serwerem NAT) i realizację usługi (wideo)telefonii przez Internet wykorzystującej stronę WWW jako interfejs użytkownika.

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 R. R, Chodorek A., Rzym G., Wajda K., Badania wideokonferencji wykorzystującej WebRTC z mostkiem konferencyjnym w środowisku chmury, „Przegląd Telekomunikacyjny, Wiadomości Telekomunikacyjne" 2017, nr 6.
4. Bernier P., "How IoT and WebRTC Can Change the World", http://www.realtimecommunicationsworld.com/topics/realtimecommunicationsworld/articles/400358-how-iot-webrtc-change-world.htm
5. Chodorek R. R, Rzym G., Wajda K., Chodorek A., Analiza współpracy terminali mobilnych wykorzystujących technikę WebRTC z telefonią VoIP stosującą protokół SIP, „Przegląd Telekomunikacyjny, Wiadomości Telekomunikacyjne" 2016, nr 6.
6. Loreto S., Romano S. P., Real-Time Communications in the Web. Issues, Achievements, and Ongoing Standardization Efforts, „Internet Computing" 2012.
7. Amirante A., Castaldi T., Miniero L., Romano S. P., On the seamless interaction between webRTC browsers and SIP-based conferencing systems, „Communications Magazine" 2013, nr 4.
8. Johnston A., Yoakum J., Singh K., Taking on webRTC in an enterprise, IEEE Communications Magazine, 2013, nr 4.
9. Chodorek A., Chodorek R. R, Udostępnianie mediów lokalnych przeglądarce internetowej z wykorzystaniem techniki WebRTC, „Autobusy: technika, eksploatacja, systemy transportowe" 2016, nr 6.
10. Silberschatz A., Galvin P. B., Gagne G., Operating System Concepts, 9th Edition, John Wiley & Sons, Inc., 2013.
11. Fette I., Melnikov A., The WebSocket Protocol, RFC 6455, 2011.
12. The WebSocket API, http://www.w3.org/TR/websockets/
13. node-static, https://www.npmjs.com/package/node-static
14. socket.io, https://www.npmjs.com/package/socket.io

Signalling servers for WebRTC technology with the use of node.js run-time environment

Web Real-Time Communications (WebRTC) technology is an enabler of native transmission of multimedia information between two and more Web browsers. WebRTC is based on HTML version 5 (HTML5) and JavaScript languages. Although both real-time media streams (audio, video) and non-real-time non-media flows (also present in WebRTC architecture) are transmitted directly between browsers, to enable transmission of information necessary for session management (and more precisely: session establishment) purposes, a signalling server is needed. Such server may be regarded as a rendez-vous point of end systems that use WebRTC technology. In this paper, node.js run-time environment and principle of building of Webrtc signalling server with the use of scripts written in JavaScript and run in node.js are presented. Example of script of a simple signalling server also is shown. The exemplary server connects users of one and only session, although this session may serve not only one-to-one, but also many-to-many connections. Despite its simplicity, this is a fully functional signalling server, able to serve signalling messages for purposes of message exchange, or for purposes of initialization of WebRTC-based Internet telephony or simple conferencing systems.

Keywords: client-server, session establishment, node.js, signalling server, 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