ИНТЕРНЕТ СВРЪЗКИ В РЕАЛНО ВРЕМЕ. ТЕХНОЛОГИИ, ПРИЛОЖЕНИЯ, СИГУРНОСТ.

доц. д-р Светослав Енков, докт. Емил Ангелов
Пловдивски университет „Паисий Хилендарски“, Факултет по математика и информатика, гр. Пловдив бул. "България" 236
e-mail: emil.angelov@gmail.com

Абстракт
В статията са разгледани WebSockets, socket.io и WebRTC. Анализирани са приликите и разликите помежду им и са очертани предимствата и недостатъците им при решаване на различни типове задачи. Проследени са поддръжката и развитието им и са очертани тенденции в тяхната бъдеща употреба. Акцентирано е върху проблемите сигурност/несигурност. Разгледано е реално приложение за клиент-сървърна обработка на изображения, както и за разпределена между клиенти обработка - с и без сървърно участие. Изследвана е ролята на факторът свръзка – скорости, трафик, надеждност. Изследван е вариативен подход при определянето на най-подходящата за конкретните обработки конфигурация от свръзки. Повод за изследването е ролята на свръзката и нейното отражение върху производителността и надеждността на фотообработките. На базата на многофакторни измервания, като резултат с все по-голяма сила се налага изводът, че за оптимизирането на процеса е задължително използването на променливо канализиране от всеки изброен тип свръзка.
***Ключови думи**: WebSockets, socket.io, WebRTC, JS, AJAX*

ВЪВЕДЕНИЕ

Когато изграждаме приложение в реално време, идва момент, в който трябва да изберем как да реализираме обмена на данни в реално време между клиент и сървър. WebSockets и Socket.IO са може би две от най-популярните решения за внедряване на комуникации в реално време в съвременната мрежа. Кое да изберем? Правилният избор на технология изисква добро познаване на алтернативите, техните предимства и недостатъци.

WebSockets

WebSockets е комуникационен протокол, който осигурява пълнодуплексен комуникационен канал през една TCP връзка. Той позволява взаимодействие между клиент и сървър с минимални режийни разходи, което ни позволява да създаваме приложения, които използват предимствата на комуникациите в реално време. Например, ако изграждаме приложение за чат, трябва да получаваме и изпращаме данни по най-бързия начин. Точно това прави WebSockets. Може просто да отворим една TCP връзка и да споделим данни, оставяйки я отворена, стига да имаме нужда от нея. WebSockets се появи за първи път през 2010 г. в Google Chrome 4, а първият RFC (RFC 6455) беше публикуван една година по-късно, през 2011 г.

Основно WebSockets се ползват за:

- Приложения за чат;
- Мултиплейър игри;
- Съвместно редактиране;
- Социални емисии;
- Отдалечено получаване на данни в реално време;
- Приложения, базирани на местоположението;
- Много други...

Socket.IO

Socket.IO е JavaScript библиотека, изградена върху WebSockets и други технологии. Използва WebSockets, когато е наличен, но може да се върне към други технологии, като Flash Socket, AJAX Long Polling, AJAX Multipart Stream и много други, което позволява Socket.IO да се използва в контексти, където WebSockets не се поддържат.

Разлики между WebSockets и Socket.IO

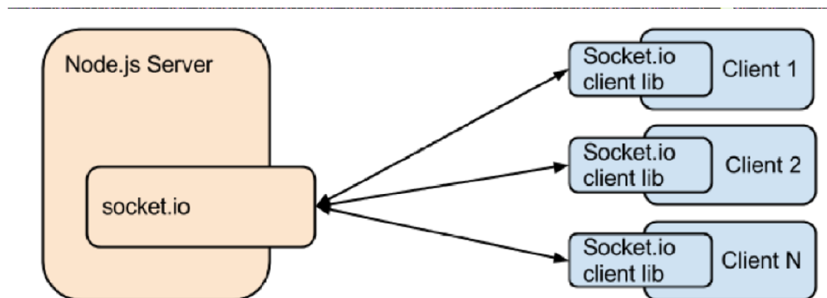
За разлика от WebSockets, Socket.IO ни позволява да излъчваме съобщение до всички свързани клиенти. Например, ако пишем приложение за чат и искаме да уведомим всички свързани клиенти, че нов потребител се е присъединил към чата, можем лесно да предаваме това съобщение - чрез едно съобщение до всички (фиг. 1). Използвайки обикновен WebSocket, ще ни е необходим списък с всички свързани клиенти и след това изпращаме съобщението директно, но един по един.

Прокси сървърите и балансиращите устройства правят WebSockets трудни за изпълнение и мащабиране. Socket.IO поддържа тези технологии без затруднения.

Както беше казано по-рано, Socket.IO може да се върне към технологии, различни от WebSockets, когато клиентът не го поддържа.

Ако по някаква причина връзката с WebSocket прекъсне, тя няма да се свърже автоматично отново, докато Socket.IO се справя с това автоматично!

API на Socket.IO са създадени, за да се работи с тях по-лесно.



Фиг. 1. Схема на работа на Socket.IO

Socket.IO може да бъде определен като „връх на комуникациите в реално време“. Разбира се, има някои добри причини да се използва ванилови WebSockets.

На първо място, всеки съвременен браузър поддържа WebSockets. Socket.IO използва много повече първоначален код и ресурси, за да го върне към други технологии. През повечето време не се нуждаете от това ниво на подкрепа. Дори по отношение на мрежовия трафик, Socket.IO е много по-скъп. Всъщност, с обикновените WebSockets браузърът може да се наложи да изпълни само две заявки:

- Заявката GET на HTML страницата, и
- Връзката UPGRADE към WebSocket.

За разлика от това, при Socket.IO се изпълняват:

- Заявката GET на HTML страницата;
- Клиентска библиотека на Socket.IO (207kb);
- Три заявки на Ajax с дълго анкетиране;
- Връзката UPGRADE към WebSocket.

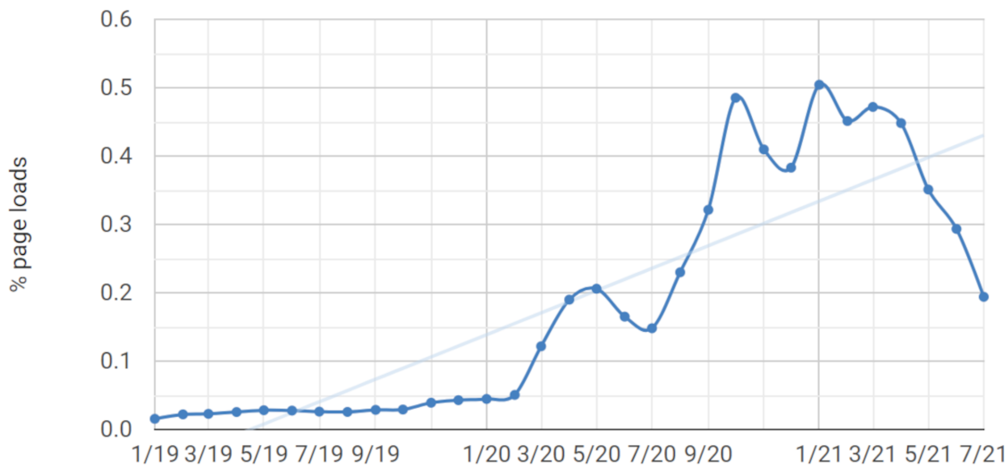
В свят, в който се използват много JavaScript код и библиотеките драстично намаляват „теглото“ си ... 207kb са много! А какво да кажем за всички тези искания? Каква загуба на мрежов трафик! Съществуват дори тестове, като например един пакет в npm, наречен websocket-vs-socket.io (https://www.npmjs.com/package/websocket-vs-socket.io), който е създаден за сравняване на мрежовия трафик на тези две технологии – разликата е огромна!

WebRTC

WebRTC е спецификация на HTML5, която можете да използваме, за да добавяме медийни комуникации в реално време директно между браузъри и/или устройства. Казано с други думи, WebRTC дава възможност за аудио и видео комуникация, работещи в уеб страници, като за да направи това не е необходимо предварително инсталиране на приставки, плъгини или каквото и да е в браузъра.

WebRTC беше промотиран официално през 2011 г. и оттогава популярността и употребата му постоянно нараства. Счита се, че само до 2016 г. е имало около 2 милиарда инсталирани браузъри, които имат възможност да работят с WebRTC. От гледна точка на трафика, са отчетени приблизително над милиард минути и 500 терабайта данни, предавани всяка седмица само от комуникациите между браузъри.

WebRTC нараства в популярността и използването по време на пандемията от COVID-19. Карантините и работата от дома направиха необходимостта от отдалечени комуникации, индоктринирайки милиарди потребители относно използването на видео разговори. Крайният резултат е скок в използването на WebRTC (фиг. 2).



Фиг. 2. Динамика на използването на WebRTC по време на пандемията COVID-19

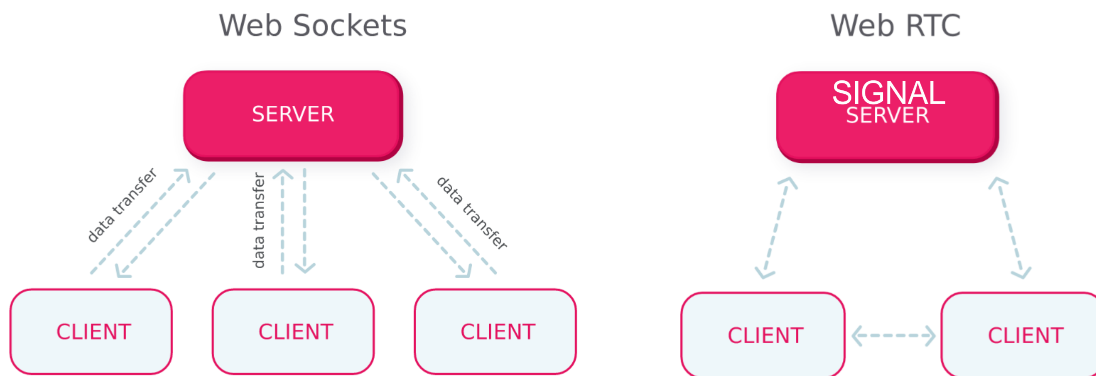
През 2021 г. WebRTC беше официално стандартизирана, което отхвърли всички съмнения относно бъдещите му перспективи. Днес WebRTC е широко популярен за видео разговори, но е способен на много повече. Важно е да отбележим, че WebRTC е напълно безплатен.

Фиг. 3. Участие на сигнален сървър в комуникацията

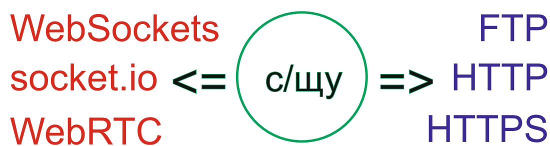
WebRTC идва като проект с отворен код, който е вграден в браузъри, но може да го вземем и ползваме за собствените си нужди. Това от своя страна създава жива и динамична екосистема около WebRTC на различни проекти и рамки с отворен код, както и търговски предложения от компании, които помагат изграждането на пресонализирани продукти. WebRTC непрекъснато се развива и подобрява, така че трябва да бъде проследяван. Да обърнем внимание на фигура 3 – тук имаме участие на т. нар. **Сигнален сървър,** чиято функция се свежда до маршрутизиране на пакетите. През него не минават данни. Пише се на node.js.

ПОСТАНОВКА НА ЕКСПЕРИМЕНТА. РЕЗУЛТАТИ.

Следва обхват на стари и нови връзки и тяхното съвместно използване в експеримент, който изследва ролята на всяка връзка, както и взаимното им допълване или изключване с цел постигане на оптимален трафик откъм скорост и надеждност.



Фиг. 4. Схема на работа на Web Sockets и Web RTC



Фиг. 5. Web Sockets и Web RTC срещу FTP и HTTP

На фиг. 5 са представени известните от по-далечното минало свързки, противопоставени на разглежданите тук – отляво в червено.

На базата на гореописаните стандарти, протоколи и базираните на тях свръзки, е проведен един практически експеримент, който да оптимизира и опише използването на различните видове канали за комуникация.

За целта е написан модул, именуван **TTS** (Traffic Test and Switch) за най-добър трафичен резултат. От една страна е тестван един стабилен **FTP** канал. От друга е използвана node.js server апликация със сървърно socket.io, като част от интегрираната JS среда, в едно с клиентско socket.io. Предимството тук е, че сървърното socket.io може да обслужи много клиенти паралелно или един клиент с голяма оптимизация на връзката при слаби физически параметри. Интересното е, че FTP скоростта пада при много малообемни файлове и се закрепя стабилно при малък брой големи файлове. TTS следи и отчита в реално време тези промени. Отлични резултати се получават, когато големите файлове минават през FTP, а множеството малки файлчета – миниатюрите през сокета.

За целта на експеримента, при който искаме да видим ролята на свръзките при разпределянето на обработки между клиент и сървър, сме избрали JS среда за фотообработки. Те сами по себе си представляват конкретен избор, но винаги може да бъдат заменени от всякакви произволни други обработки – като например намиране на криптоключове, задачи от астрономия, физика, медицинска генетика и други области. Т. е. нашето изследване може да се обобщи в посока – разпределяне на обработки между клиент и сървър, а нашите тестове – прехвърляне на обработки на мощен сървър, а съвсем абстрактната посока би могла да бъде – разпределяне на обработки в многомашинен клиентски слой. Но това е предмет на последващи изследвания.

WebRTC модулът се явява резервен канал, включен в приложението – който предимствено е за връзка между различни клиенти (p2p), но в частният случай при него едната страна може да е нашият сървър. Добрата му характеристика е, че прекарва трафика през най-бързите за всеки един момент пътища - нещо, което не е така при сокетите и FTP. Той е най-стабилен при нестабилна връзка и най-сигурен за безаварийното доставяне на файловете в двете посоки, дори при прекъсване на връзката и след възстановяването ѝ не се налагат повторни опити. При трите вида връзки се използва Хофманов контрол за интегритета (цялостност и здравина до байт) на файловете. FTP вграденият контрол води до повторение на качванията на цели файлове, което не е приемливо при големите файлове. Сокетите контролират на пакети и там повторенията им водят до по-малки загуби на време. При слаба, нестабилна или много променлива връзка, WebRTC спасява положението. В случай, че имаме сигурна връзка, може да го изключим от проверките, за да не стават много. Това също е в обхвата на TTS. Ще обобщя, че благодарение на TTS, при слаба връзка имаме средно приблизително удвояване на скоростта или двукратно съкращаване на времето за пренос на снимките, спрямо най-бързия от трите канала, използван самостоятелно. При нашите тестови данни – **качване** над 1000 Mb в 100 файла и **сваляне** над 1260 Mb в 400 файла с различни размери разделям връзките на 3 вида:

Само FTP:

- Слаб трафик > 60 минути многократни увисвания
- Среден трафик ~ 30 минути
- Силен трафик < 15 минути

Само WebSockets:

- Слаб трафик ~ 45 минути
- Среден трафик ~ 30 минути
- Силен трафик < 15 минути

Само WebRTC:

- Слаб трафик ~ 40 минути
- Среден трафик ~ 30 минути
- Силен трафик < 15 минути

Чрез включен TTS:

- Слаб трафик ~ 30 минути
- Среден трафик ~ 15 минути
- Силен трафик ~ 5 минути

ЗАКЛЮЧЕНИЕ

В заключение може да се направи следното обобщение при ползването на TTS:

При възможност трябва да се работи с всички канали, изключвайки губещите, като FTP и да се ползват паралелно сокети. При определена нестабилност, WebRTC държи трафика на линия, като спестява повторения, които иначе са на ниво пакет при сокетите, а при FTP – на ниво файл. При големи забавяния, се изключват до ново включване, ако връзката се подобри.

Благодарности: Работата е подкрепена от проект МУ21-ФМИ-004, финансиран от Фонд „Научни изследвания“ при Пловдивския университет „П. Хилендарски“.

ЛИТЕРАТУРА

1. Aurelia, [https://aurelia.io, 20.08.2021].
2. Backbone.js, [https://backbonejs.org, 20.08.2021].
3. Bootstrap, [https://getbootstrap.com/, 20.08.2021].
4. DoFactory, JavaScript Design Patterns, [https://www.dofactory.com/javascript/design-patterns, 20.08.2021].
5. DoFactory, JavaScript Introduction, [https://www.dofactory.com/javascript, 20.08.2021].
6. Ember.js, [https://emberjs.com/, 20.08.2021].
7. jQuery, [https://jquery.com/, 20.08.2021].
8. Haverbeke, M., Eloquent JavaScript, 2018, 3rd edition, [https://eloquentjavascript.net, 20.08.2021].
9. MDN Web Docs, [https://developer.mozilla.org/, 20.08.2021].
10. MDN Web Docs, Pixel manipulation with canvas, [https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas, 20.08.2021].
11. Meteor, [https://www.meteor.com/, 20.08.2021].
12. Microsoft Docs, [https://docs.microsoft.com/, 20.08.2021].
13. Mithril, [https://mithril.js.org, 20.08.2021].
14. npm, [https://www.npmjs.com/, 20.08.2021].
15. Polymer Project, [https://www.polymer-project.org, 20.08.2021].
16. React, [https://reactjs.org/, 20.08.2021].
17. LogRocket, [https://blog.logrocket.com/webrtc-over-websocket-in-node-js/, 20.08.2021]
18. WebRTC vs WebSockets, [https://requestum.com/webrtc-vs-websockets, 20.08.2021]
19. Socket.IO, [https://socket.io/get-started/chat, 20.08.2021]
20. JScrambler, [https://blog.jscrambler.com/mixed-signals-with-socket-io-and-webrtc/, 20.08.2021]