Методологични проблеми при създаването на уебприложения, работещи в реално време

Павел Петров

Methodological problems in the creation of web applications running in real time

Pavel Petrov

Abstract

The study focuses on methodological problems in the creation of web applications running in real time. In this paper, the term web applications running in real time we understand the distributed client-server application that uses techniques to reduce the time interval from the time of occurrence of an event on the web server side to its interpretation (mostly visual) by the web client This paper covers: classical approaches based on a technique "pull" for realization of "thin" and "fat" ("thick") clients; approaches based on techniques "long pooling", "push" and AJAX; approaches based on the latest web standards, issued by the IETF and W3C. The paper also discusses some practical problems in the exchange of data in real-time distributed client-server web applications.

Key words: web applications, real time, information systems, AJAX

Понятието уебприложение в реално време започва да се среща все по-често в литературата, но обхвата му все още не е достатъчно добре дефиниран и варира в широки граници в зависимост от контекста на изложението. В тази публикация, под понятието уебприложение в реално време ще разбираме разпределено клиент-сървър приложение, което използва техники за намаляване на интервала от време от момента на възникване на някакво събитие от страна на уебсървъра, до неговото интерпретиране (най-често визуализиране) от страна на уебклиента (най-често уеббраузър)[1]. Интервалът от време (често наричан още лаг или времезабавяне) трябва да е толкова малък, че да създава субективното усещане у потребителя за незабавно изпълнение, т.е. той не трябва да усеща забавяне между генерираните от него събития и ответната реакция. В началото сървърът най-често е класически уебсървър, а в процеса на еволюция навлизат в употреба и модифицирани специализирани сървъри като комет-сървър, APE-сървър или WebSocket-сървър.

Класически подходи, базиран на техника "издърпване" за реализация на "тънки" и "дебели" клиенти.

Исторически най-старият подход за работа в реално време, е зареждането в уеббраузъра на страница, която се обновява постоянно, през определен интервал от време. Това е т.нар. техника "издърпване" - "pull", която се реализира чрез HTML тага meta или чрез използване на таймер в JavaScript. При първия вариант не се налага програмиране от страна на клиента, а във втория се използва език за програмиране, като в годините до 2005 г. освен JavaScript се е използвал и VBScript главно за Internet Explorer. При този подход цялата или основната част от логиката е изнесена от страна на сървъра и натоварването му нараства значително, при намаляване на времето за обновяване на уебстраницата и при нарастване на броя на уебклиентите. Допълнителен проблем е, че при неуспешно зареждане на поредната страница, се налага потребителят собственоръчно да презареди страницата, тъй като изведеното съобщение за грешка прекъсва цикъла на презареждане.

Определянето на интервала от време за презареждане се осъществява чрез съобразяване с две взаимно противоречащи си условия: висока степен на актуалност на данните (минимална латентност) при ниско натоварване на уебсървъра. По-дългият интервал може да доведе до пропускане на точния момент на възникване на някое събитие при сървъра, а по-късият интервал води до по-голям трафик и натоварване на уебсървъра. В идеалния случай интервалът на презареждане трябва да съвпада с интервала на възникване на събития при сървъра.

[1] Под събитие имаме впредвид ситуация на промяна на данните в сървърната част на разпределеното уебприложение, която клиентската част своевременно трябва да инерпретира по подходящ начин.

Еволюция на гореописания подход е използването на HTML фреймове - чрез тагове frameset или iframe, като в родителския фрейм се постави програмния код за обновяване на видим или невидим подчинен фрейм чрез таймер. Тъй като родителския фрейм се зарежда еднократно в началото и в последствие таймерът работи постоянно, се избягва проблемът при неуспешно зареждане на поредна страница във фрейма. При такава ситуация, дори и в подчинения фрейм да възникне грешка, следващото задействане на таймера в родителския фрейм ще направи опит за ново зареждане и цикъла на презареждане няма да се прекъсне. Тук отново цялата или по-голяма част от логиката е от страна на сървъра, т.е. това е т.нар. архитектура "тънък клиент".

Следващ етап от еволюцията на уебприложенията работещи в реално време е намаляване или премахване на изпращането на едни и същи данни многократно. Обикновено при уебприложения в реално време структурата на уебстраницата се запазва в течение на времето, а се променят само показваните данни. Ако шаблонът на страницата, указващ положението и оформлението на отделните данни, се изпрати еднократно, а в отговор на всяка заявка се изпращат само променящите се данни, то трафикът между сървъра и клиента може да спадне значително, което влияе положително и върху намаляване на времевия интервал от изпращане на HTTP заявката до получаване на отговора. Допълнителен положителен ефект има при уебсървъра, тъй като отпада операцията вмъкване на данни между HTML тагове, което на практика представлява елементарно слепване на символни низове или поредица от операции търсене и заместване при използване на шаблони. И при двата варианта се използва значително количество оперативна памет и честото й динамично заемане и освобождаване намалява производителността на сървъра като цяло. В тежки случаи може да се стигне и до използване на тази част от виртуалната памет, която е разположена на периферно устройство, което още по-рязко намалява производителността. Изпращането само на данните, и то само на тези, които са се променили, прехвърля значителна част от обработката на данните от сървъра към клиента и се отива към т.нар. архитектура "дебел клиент". В идеалния случай от страна на уебсървъра само се извличат данните (от файлове, чрез заявка до сървър на база от данни или чрез NoSQL) и веднага се връща отговор на клиента без някаква допълнителна обработка.

В практиката този подход се реализира чрез използване на поне три фрейма - родителски, видим и невидим. В родителския фрейм се съхраняват шаблоните на уебстраниците и чрез таймер се реализира постоянно презареждане на невидимия фрейм. В него, освен данните, има и кратък програмен код, който се изпълнява веднага след зареждане на страницата и най-често извиква функция от родителския фрейм. Тя, от своя страна, поставя данните между HTML тагове от шаблоните и, или чрез DOM променя само отделни части от вече изобразена уебстраница, или визуализира изцяло генерирана нова уебстраница.

С цел по-нататъшно съкращаване на времевия интервал от момента на настъпване на някакво събитие от страна на сървъра до получаването му от клиента, в практиката бяха реализирани редица подходи, използващи различни особености при реализацията на протокола HTTP, както при уебсървърите, така и при уебклиентите. Подходите, описани по-горе, при които периодично чрез таймер се отправят заявки, в литературата са известни като "издърпващи" - "pull" техники, тъй като клиента периодично "издърпва" нови данни от сървъра и така се създава илюзията за непрекъснат обмен на данни. Както се вижда, те са базирани почти изцяло на HTML и до известна степен на JavaScript.

Следващото стъпало в еволюцията на "pull" техниката се базира на особеност на протокола HTTP по отношение на времето за прекъсване на мрежовата връзка при липса на обмен на данни (timeout). В спецификацията на HTTP/1.0 няма предвидени ограничения за това време[2], т.е. теоретически мрежовата връзката между клиента и сървъра може да остане отворена неограничен период от време. На практика обаче масово при реализацията на клиента и сървъра се предвижда такова ограничение, като до преди години по подразбиране то беше

[2] Berners-Lee T., Fielding R., Frystyk H. Hypertext Transfer Protocol - HTTP/1.0, 1996 <http://www.ietf.org/rfc/rfc1945.txt>

най-често 300 секунди (5 минути), а в последните години има тенденция да намалява.

Подходи, базирани на техники "дълго запитване", "избутване" и AJAX.

Тези особености по отношението на времето за прекъсване се използват за реализация на техника, известна в литературата[3] като "дълго запитване" - "long polling" . При нея отново периодично клиентът изпраща заявки, но сървърът не връща веднага отговор, а изчаква да настъпи някакво събитие и тогава предава данните за него. В случай, че наближава времето за прекъсване на мрежовата връзка при липса на трафик, а събитието все още не е настъпило, се изпраща формален отговор, че няма данни, след което клиентът повтаря процедурата. По този начин мрежовата връзка изкуствено се поддържа отворена продължителен период от време, но за сметка на това веднага при настъпване на събитие сървърното приложение изпраща отговор до клиента. Като допълнителен ефект може да се получи значително намаляване на трафика в сравнение с обикновената "pull" техника, при която сървърът веднага връща отговор, независимо от това има ли нови данни или няма такива. Очевидно е, че намаляване на трафика ще имаме само когато интервала от време, през който клиентът изпраща заявките, е по-малък от интервала от време, през който възникват събитията при сървъра.

При "long polling" техниката, се предполага, че сървърът не разполага с нужните данни към момента на получаване на заявката и той задържа по-дълго време от нормалното връщането на отговор, докато те не се получат или не настъпи някакво събитие. Когато данните се получат, сървърът отговаря и, или веднага затваря връзката (класически "long polling"), или продължава да я държи отворена и да изпраща данни към клиента като добавя тагове в един "безкраен" iframe (преминаване към "push" техника).

Развитие на "long polling" техниката е т.нар. "push" - избутваща техника (в литературата се среща още и като "web page streaming"), при която по отворената мрежова връзка сървърът дълго време изпраща отделни порции физически завършени HTML тагове. Повечето уебклиенти са така реализирани, че с цел по-бързо визуализиране на уебстраницата, започват интерпретирането на HTML таговете преди да се е получил целия отговор, т.е. преди да се е получила двойката за край на документа </body></html>. Отделните порции физически завършени HTML тагове всъщност представляват програмен код, ограден в тагове <script></script>, в който са както данни, така и програмния код (най-често извикване на функции от родителския фрейм), опресняващ съдържанието на видимата уебстраница, фрейм или отделен DOM елемент.

Основен недостатък на тези техники е, че се налага използването на специализирани уебсървъри (подобни на Node.js), тъй като повечето универсални уебсървъри (като Apache и IIS) не са предназначени да поддържат голямо количество едновременно отворени мрежови връзки дълги периоди от време. Налага се употребата на уебсървъри с по-малко богатство от възможности, но по-добре използващи оперативната памет за всяка една мрежова връзка. Допълнителен проблем може да възникне, ако между клиента и сървъра има прокси сървър, чиито настройки да не позволяват мрежовата връзка между тях да остава отворена дълги периоди от време. Възможно е и с цел антивирусна защита прокси сървърът да очаква предаването на цялата заявка или отговор, и след това да я изпрати на другата страна. Заради такива случаи се предпочита използването на HTTPS, тъй като криптираното съдържание обезсмисля неговия междинен анализ за целите на антивирусната защита.

След масовото въвеждане на вградения обект XMLHttpRequest в уеббраузърите[4], който дава възможност за асинхронно отправяне на заявки и получаване на отговор, техниките описани по-горе еволюират в направление на отпадане използването на скрити фреймове за сметка на използването на XMLHttpRequest обекти - т.нар техника AJAX. Съществуват различни приложения, включващи уебсървър и библиотека с функции на JavaScript, които да

[3] Виж Loreto, S., Saint-Andre, P., Salsano, S., G. Wilkins, RFC 6202: "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", 2011, <http://www.ietf.org/rfc/rfc6202.txt >.

[4] Често в литературата този обект съкратено се нарича XHR.

улеснят разработването на двете части на разпределеното уебприложение чрез AJAX. За съжаление, в тези приложения не е решен общия проблем, наличен и при двете техники "push" и "long polling" - поддържането на голям брой едновременно отворени мрежови връзки от страна на сървъра. Техниката "long polling" се поддържа без изменения, докато техниката "push" е значително видоизменена, тъй като отговора по AJAX не се обработва директно от уебклиента, а от потребителска функция.

Както видяхме от гореизложените подходи, основен нов момент в развитието на уебтехнологиите е усъвършенстване на техниките, използвани при изграждане на уебприложения, работещи в реално време. Подобни приложения стават все по-популярни и се различават съществено от класическите разбирания за уебсайт, доближавайки се като функционалност до десктоп приложенията. Тези подходи преди 2004 г. са се е реализирал със скрити фреймове в уебстраница, а след 2005 г. масово започва да се използва вграден обект XMLHttpRequest , като популярността му нараства с широкото използване в уебуслугите на Google - Mail и Maps. От тогава техниката с използване на новия за времето си обект започва да се нарича AJAX.

AJAX е съкращение на Asynchronous JavaScript and XML, но това име според нас не отразява точно същността на технологията, тъй като заявките може да се изпращат и синхронно, а не само асинхронно; може да се използва и друг език за програмиране, например VBScript за IE, а не само JavaScript; данните може да се предават и в други формати - TXT, HTML, JSON, а не само в XML.

Технологията AJAX е съвременен начин за създаване на интерактивни разпределени уебприложения. При класическия модел уеббраузърът зарежда цяла уебстраница, изчаква да се случи определено събитие и зарежда друга уебстраница. При използване на AJAX, докато една страница е заредена, на заден план (най-често асинхронно) се изпращат HTTP заявки, без да се презарежда цялата страница. Обработката на HTTP отговора също се извършва на заден план.

Както е известно, действията които извършва потребителя с мишката или клавиатурата докато работи с една уебстраница, генерират събития. Тези събития могат да се прихващат и да се изпълнява определен програмен код. Прихващането става чрез т.нар. манипулатори, на които се присвоява като низ програмния код, който трябва да се изпълни. Обикновено в низа не се пишат много програмни редове, а се извиква само една функция. Тази функция специално се предвижда да се изпълнява след точно определено действие, като например кликване върху някой елемент, и този начин на програмиране е известен като събитийно програмиране, както вече казахме по-рано.

Манипулаторите, които прихващат събития се записват като параметри на HTML таговете. Различните тагове поддържат различно множество от събития и този събитиен модел е стандартизиран от W3C[5] под формата на технически доклади. Основните събития от страна на уебклиента са следните:

- генерирани от мишката - onclick, ondblclick, onmousedown, onmousemove, onmouseover, onmouseout, onmouseup;

- генерирани от клавиатурата - onkeydown, onkeypress, onkeyup;

- генерирани от прозорци, фреймове и други обекти - onabort, onerror - генерира се при грешка в скрипта, onload - генерира се при завършване на зареждане на страницата, onresize, onscroll, onunload;

- генерирани от формуляри и полета в тях - onblur - генерира се при напускане на обект, onchange - генерира се при промяна съдържанието на обекта, onfocus - генерира се при получаване на фокуса от обекта (т.е. ще получава данни от клавиатурата), onreset, onselect - генерира се при маркиране на текст, onsubmit - генерира се при изпращане на формуляр. Преди да се изпратят, формулярите най-често се проверяват дали са попълнени всички задължителните полета, дали въведените данни са валидни - валиден имейл адрес, въведен ли е текст в цифрови полета, дати и т.н.

[5] Document Object Model Events, <http://www.w3.org/TR/DOM-Level-2-Events/events.html>

С AJAX сравнително лесно може да се реализират техниките "издърпване" - "pull" и "дълго запитване" - "long polling", но техниката "избутване" - "push" трудно се реализира, като най-често се използват няколко AJAX заявки работещи в комбинация по техниката "дълго запитване" - "long polling". Популярно средство, с лиценз свободен софтуер, което значително улеснява разработката на уебприложения в реално време, използващи "push" техниката, е Ajax Push Engine. За съжаление сървърната част изисква операционни система UNIX/Linux/MacOSX, тъй като използваните техники за оптимално поддържане на голям брой мрежови връзки не работят под Windows[6]. Друго популярно средство, специално за разработчици на Java, е CometD , също с лиценз свободен софтуер[7].

Подходи, базирани на нови уеб стандарти на IETF и W3C.

Като заместител на все по-усложняващите се AJAX техники за реализация на "push" техниките, W3 Consortium предложи протокола Server-Sent Events (SSE) , който към есента на 2015 г. се поддържа от съвременните версии на Firefox, Chrome, Opera, Safari и др., но не се поддържа от Internet Explorer 11 и от браузъра на Windows 10 - Edge[8].

При SSE, уебприложение чрез изпращане на заявка се "абонира" към поток от събития, генерирани от уебсървър. Връзката остава отворена неограничен период от време и когато ново събитие се появи при сървъра, към клиента се изпраща съобщение по нея. Връзката е основно еднопосочна - от уебсървъра към уебклиента и SSE на практика представлява стандартизиран и усъвършенстван вариант на "long polling" и "push" техниките.

Предполагаме, че този протокол не получава подкрепа от Microsoft, тъй като протоколи като WebSocket предлагат повече възможности - двупосочна асинхронна връзка. Въпреки че използването на двупосочен канал за връзка дава повече възможности пред еднопосочния (при създаването на игри, приложения за съобщения и прочие), то съществуват различни ситуации, при които не се налага активно да се изпращат данни от клиента към сървъра. Например, следене на обновяване на потребителски статуси, новини, борсови котировки и т.н. В случай, че се налага изпращането на данни от уебклиента до уебсървъра, може да се направи заявка чрез AJAX или по друг начин.

При класическите заявки по протокол HTTP 1.0, уебклиентът изпраща заявка до уебсървъра, получава отговор и мрежовата връзка между двете страни се прекъсва. Версия 1.1 на HTTP даде възможност в рамките на една мрежова връзка, уебклиентът синхронно да изпраща на уебсървъра множество заявки, като по този начин се избягва т.нар. "трипасово ръкостискане" характерно за TCP/IP при отваряне на една нова мрежова връзка (сокет), както и разходите от време, свързани със затварянето й. В резултат, мрежовата връзка между клиент и сървър стои отворена много по-дълго време, спрямо връзките по протокол HTTP 1.0. Това наложи еволюцията на уебсървърите да се развие и в посока поддържане на по-голям брой едновременно отворени мрежови връзки, с цел да няма отказ от обслужване при голям брой HTTP заявки за единица време. В резултат се появиха и наложиха уебсървъри, способни да поддържат десетки хиляди едновременно отворени връзки при сравнително малък обем заета оперативна памет, като например nginx и lighttpd. За сравнение, при най-популярните уебсървъри - Apache и IIS, обемът памет, заеман за всяка мрежова връзка е много по-голям, което налага и по-големи изисквания към хардуера по отношение на инсталираната оперативна памет.

От друга страна, реализацията на техниките "издърпване" - "pull", "дълго запитване" - "long polling" и "избутване" - "push" в повечето случаи изисква използването на средства, които по принцип отговарят на стандартите, но не са изрично предвидени в тях. Затова през последните години за създаването на приложения в реално време се предпочитаха други подходи, като Java аплети (все по-рядко), Flash приложения (най-често) или Silverlight

[6] Ajax Push Engine, <http://ape-project.org/>
[7] CometD Bayeux Ajax Push, <http://cometd.org/>
[8] Server-Sent Events, Editor's Draft 14 May 2014, <http://dev.w3.org/html5/eventsource/>

приложения (много рядко). Използването на подобни приложения затруднява потребителите, тъй като изисква инсталирането на допълнителни модули (плъгини) към браузърите и това намалява тяхната приложимост.

В документния обектен модел на уебстраниците (DOM) през последните години бяха предложени за добавяне няколко глобални обекти, специално предназначени за разработката на уебприложения в реално време. Такъв обект, освен EventSource за протокола Server-Sent Events, който вече разгледахме, е обектът WebSocket, стандартизиран от W3 Consortium[9]. Също така, специално за WebSocket има и специално разработен протокол от IETF със същото име , поддържащ се от най-популярните браузъри към настоящия момент[10]. Това осигурява широкото разпространение на WebSocket и той е особено перспективен, тъй като решава всички въпроси, свързани с двупосочната асинхронна връзка между клиента и сървъра. Той опростява значително създаването на уебприложения в реално време, като отпадат проблемите, налични при гореописаните подходи по отношение на множеството от допълнителни програмни библиотеки. Като допълнение се появи идеята за ревизиране на HTTP 1.1 и дори за замяната му с друг протокол, който по-добре да обслужва уебприложенията в реално време. Например протоколът SPDY на Google (залегнал в основата на бъдещия HTTP 2.0), който все още не се поддържа широко от уебсървърите, за сметка на браузърите. Според някои изследвания, към октомври 2015, около 6,3% от уебсайтовете използват SPDY[11], което е ръст с около 5 процентни пункта за една година, т.е. спрямо октомври 2014. В браузъра Google Chrome справка за използваемостта на този протокол може да се получи от URL chrome://net-internals/ , раздел SPDY.

Редица големи доставчици на облачни услуги поддържат WebSocket , което дава възможност за прилагането му и без пълен административен контрол върху хардуерния сървър при хостинг на такива приложения. Възможността по ефективен начин (по отношение на обем трафик и натоварване на сървъра) да се предават данни на малки части, с висока честота и асинхронно, създава предпоставки за разработването на приложения, които работят също толкова бързо и интерактивно, както и традиционните десктоп приложения. Подходите, базирани изцяло на особеностите на протокола HTTP, не са подходящи за тези цели, тъй като при предаването на заявката и на отговора, към данните се добавят и заглавни части, които съдържат различна служебна информация, като тази за типа на данните, кодировка, бисквитки, идентификация на клиента или сървъра, дата, контролни суми и др. Тези служебни данни варират значително по обем и съпоставени с размера на една уебстраница или изображение са сравнително малки, но когато данните се изпращат на малки порции и през малък интервал от време, обема на служебните данни започва да става съпоставим и може в пъти да надхвърли обема на "полезните" данни, които всъщност се предават.

За разлика от сесийния синхронен характер на HTTP (заявка - отговор), при WebSocket след като веднъж се осъществи връзката, двете страни са равноправни и използват постоянна мрежова връзка. Обменът на данни е асинхронен, което означава, че двете страни на мрежовата връзка не е нужно да се изчакват, за да изпращат данните и това може да става едновременно и от двете страни. Използването на събития от страна на клиента дава възможност без да се прекъсва изпълнението на програмен код при него, след пристигане на данни от сървъра те да бъдат обработени.

Едно от основните предимства на WebSocket е максимално изчистения и опростен приложен програмен интерфейс, който в сравнение с голямото количество библиотеки, предназначени да улеснят създаването на уебприложения в реално време, значително улеснява разработката от страна на уебклиента.

[9] The WebSocket API, W3C Candidate Recommendation, 2012, <http://www.w3.org/TR/websockets/>
[10] Fette I., A. Melnikov, RFC6455 The WebSocket Protocol, 2011, <http://tools.ietf.org/rfc/rfc6455.txt>
[11] Usage of SPDY for websites, <http://w3techs.com/technologies/details/ce-spdy/all/all>

Работата от страна на сървъра е по-сложна и изисква повече познания за стандартите на WebSocket. Затова при програмиране на Node.js с използване на WebSocket и за клиентската, и за сървърната част се препоръчва използването на програмната библиотека socket.io.

Приложни проблеми при обмена на данни в реално време при разпределените клиент-сървър уебприложения.

При разработката на разпределени клиент-сървър уебприложения, освен техниката за реализация на работа в реално време и как ще се съхраняват данните от страна на сървъра и клиента, трябва да се избере и формата на данните, който ще се използва за комуникация между двете страни. Съществуват голям брой отворени стандарти и подходът за избор на един или друг формат за обмен на данни обикновено зависи от редица фактори. Най-често се срещат 3 варианта:

1) При новоразработени клиентска и сървърна част на разпределеното уебприложение обикновено изборът се прави или на база изрични изисквания в заданието, или на база модерна към момента технология (следване на "модните тенденции"), или на база субективния опит на разработчиците.

2) При ново разработване само на клиентската или само на сървърната част, обикновено изборът се прави на база какви формати за обмен на данни поддържа другата страна. Тук по-разпространен е вариантът, при който вече съществува сървърно приложение предлагано като уебуслуга, а тепърва се разработва клиентската част. Много често се предлага и API на различни езици за програмиране, при което няма възможност за избор на стандарт за обмен на данни.

3) При съществуващи клиент-сървър приложения, които вече имат определена широко използвана функционалност, очевидно става въпрос за надграждане, при вече използван формат за обмен на данни. В този случай следва да се използва съществуващия формат, тъй като смяната му може да доведе до пренаписване на голяма част от програмния код и на клиентското и на сървърното приложение, което е свързано със значителни разходи.

Вижда се, че изборът на формат за обмен на данни е решение, което се отразява на редица бъдещи действия по развитието на едно разпределено приложение и изборът трябва да е добре мотивиран. Към настоящия момент широко се използват два текстови отворени формата за обмен и съхраняване на данни при уебприложенията в реално време - форматите XML[12] и JSON[13].

Форматът XML има универсален характер и съответно е "по-тромав" (т.е. по-обемен и по-бавен за обработка) спрямо JSON. Трябва да се отчита, че при приложенията в реално време асинхронните заявки към уебсървъра от страна на клиента трябва да са малки по обем, бързо да се обработват и обикновено се извършват от програма написана на JavaScript. Ако от страна на сървъра и на клиента имаме приложения на JavaScript, по-голямо удобство на работа дава JSON.

В **заключение**, процесът на еволюция при създаването на уебприложения в реално време не приключва с гореописаните техники, но можем да твърдим, че той е към своя край, като очакваме в скоро време да има качествено нови промени, т.е. да има по-скоро "революция", отколкото плавно еволюционно изменение в подходите. Основание за това предположение ни дава натиска, оказван от фирми с големи пазарни дялове в тази ниша (главно Google) за приемането и налагането на нови стандарти, специално предназначени за този вид приложения. Имаме в предвид протоколът SPDY на Google (залегнал в основата на новоналагащия се HTTP 2.0), обектите EventSource и WebSocket специфицирани от W3 Consortium.

[12] Extensible Markup Language (XML) 1.1 (Second Edition) <http://www.w3.org/TR/xml11/>
[13] The JSON Data Interchange Format, ECMA-404 October 2013, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

Използвана литература:

1. Berners-Lee T., Fielding R., Frystyk H. Hypertext Transfer Protocol - HTTP/1.0, 1996 <http://www.ietf.org/rfc/rfc1945.txt>
2. Loreto, S., Saint-Andre, P., Salsano, S., G. Wilkins, RFC 6202: "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", 2011, <http://www.ietf.org/rfc/rfc6202.txt >
3. Document Object Model Events, <http://www.w3.org/TR/DOM-Level-2-Events/events.html>
4. Ajax Push Engine, <http://ape-project.org/>
5. CometD Bayeux Ajax Push, <http://cometd.org/>
6. Server-Sent Events, Editor's Draft 14 May 2014, <http://dev.w3.org/html5/eventsource/>
7. The WebSocket API, W3C Candidate Recommendation, 2012, <http://www.w3.org/TR/websockets/>
8. Fette I., A. Melnikov, RFC6455 The WebSocket Protocol, 2011, <http://tools.ietf.org/rfc/rfc6455.txt>
9. Usage of SPDY for websites, <http://w3techs.com/technologies/details/ce-spdy/all/all>
10. Extensible Markup Language (XML) 1.1 (Second Edition) <http://www.w3.org/TR/xml11/>
11. The JSON Data Interchange Format, ECMA-404 October 2013, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

За контакти:
Доц. д-р Павел Петров
Икономически университет - Варна
E-mail: petrov@ue-varna.bg