Instituto Superior de Engenharia de Lisboa

Sistemas Distribuidos

Relatório do trabalho de investigação

Websockets

Autoria:
33724 David Raposo
32632 Pedro Pedroso
33404 Ricardo Mata

Em coordenação com:
Engº Luís Falcão
Engº José Simão

28 de Junho de 2014

Conteúdo

1 Introdução

Com os avanços tecnológicos que têm havido nos últimos anos, é cada vez mais fácil perder a noção do que está a acontecer dentro dos computadores. Isto faz com que se tenha algum desleixo perante os recursos que se usam. No entanto, com os dispositivos móveis em constante crescimento[1], volta a tornar-se importante a otimização dos recursos usados.

A compatibilidade entre diferentes plataformas é garantida pela utilização de protocolos que faz com que cada plataforma saiba comunicar entre si. Um exemplo desses protocolos é o protocolo HTTP[2], que surgiu como necessidade de transferir conteúdo estático (páginas de hipertexto). Desde a sua implementação, o protocolo foi beneficiando de revisões que expandiram o seu uso original.

Contudo, devido à necessidade de páginas mais interativas, a criação de páginas web, tomou a tendência de conter componentes *JavaScript*. Inicialmente com o intuito de interagir com o *DOM* atravéz de eventos, mas cada vez indo mais longe, até ao ponto de conter grande parte da lógica necessária. O que tornou esta evolução possível foi o aparecimento de *XMLHttpRequest* [3], que trouxe um grande aumento de perfomance, já que permitia obter apenas o conteudo de interesse da componente servidora, sem trazer uma paginá correspondente na integra.

As *WebSockets* surgem como um novo passo nesta procura de aumento de perfomance, que tal como o nome subentende, tenta trazer a utilização básica de sockets (tal como *HTTP*, funcionando sobre *TCP*) para a interação entre *web-browsers* e *web-servers*. Uma *WebSocket* pode também ser iniciada em modo seguro, sendos ambos protocolos conheciddos como "*ws*"(*WebSocket*) e "*wss*"(*WebSocket Secure*).

Durante este documento, serão feitas diversas comparações com o protocolo *HTTP*, já que é sobre este, que *WebSockets* surge como alternativa.

2 Visão global

O protocolo *WebSockets*, tal como o protocolo *HTTP*, pertence à camada de aplicação e funciona sobre *TCP*. Tem como objetivo trazer a possibilidade de trocar dados entre componentes cliente e servidor, de forma "semelhante"à comunicação entre duas aplicações (através de *send* e *recv*, ou abstrações que recorrem a estes mecanismos). Com comunicação semelhante, referimo-nos à ausencia de todos os *headers/identificadores* que estão presentes durante cada pedido *HTTP*, passando práticamente (já que continua a ser necessário dados para identificar os diversos pacotes) a enviar apenas o que estamos habituados a ver no corpo de pedido/resposta *HTTP*.

De forma a facilitar a percepção das vantagens que surgem do uso de *WebSockets*, iremos mostrar como o protocolo *HTTP* lida com a concorrência de pedidos e a recolha constante de nova informação da componente servidor. Tenhamos em conta, que tal como foi referido previamente, além da fase de ini-

[1] Fonte: `http://www.digitalbuzzblog.com/infographic-2013-mobile-growth-statistics/`

[2] A sigla HTTP vem de *HyperText Transfer Protocol*, que significa Protocolo de Transmissão de Hipertexto.

[3] `http://www.w3.org/TR/XMLHttpRequest/`

ciação, não existe necessidade da passagem de todos os *headers* (que pode ser uma dimensao relevante se o tamanho da mensagem a enviar for pequeno).

2.1 Antes de *WebSockets*

O protocolo *HTTP* tem como base a ideia de par pedido - resposta (sejam estes enviados na integra ou em diversos fragmentos) que necessita sempre que o cliente inicie esta "conversa". Mesmo tirando proveito da persistência de conexões, um segundo pedido teria de esperar que o primeiro acabasse. Implicando que em relação a concorrência, o protocolo *HTTP* necessita de uma conexão nova para cada pedido concorrente, o que torna importante não esquecer o facto dos *browsers* terem os seus próprios limites de conexões concorrentes para cada *host*.

Em relação a obter informação da componente servidora no protocolo *HTTP*, serão agora enumeradas as diferentes estratégias, juntamente com os problemas existentes em cada:

1. *Polling*: Consiste em efetuar periodicamente pedidos a questionar o servidor se existem novos dados a obter. Isto trás um custo elevado, pois ao serem feitos constantemente pedidos pode ser necessário estar a criar novas conexões (por não haver nenhuma disponivel, por a anterior estar ocupada... ou por ter sido fechada). Muitos destes pedidos provavelmente poderão obter resposta que diz não haver informação nova.

2. *Long-Polling:* Semelhante ao *polling*, mas o servidor prende a ligação até que haja informação a enviar. Assim que haja informação a enviar, o servidor responde com os dados de interesse. É mais vantajoso que o *polling*, já que não se faz pedidos desnecessários (a não ser que haja algum mecanismo de timeout que liberte o pedido antes de haver dados). Tal como polling, implica que novos pedidos sejam feitos para cada obter novos dados.

3. *Pushing:* É feito um pedido de dados ao servidor. O servidor mantém a ligação aberta e vai enviando dados para o cliente sem nunca fechar a ligação (tira partido de ser possivel ler partes da resposta no *XMLHttpRequest*). É mais vantajoso face a *Long-Polling* na medida que o cliente recebe novos dados sem ter de iniciar uma nova ligação. No entanto, pedidos para intuitos diferentes terão de ter a sua própria conexão.

2.2 Após *WebSocket*

Os problemas referidos previamente são algo que não surge com o uso de *WebSockets*, já que ao manter uma ligação persistente sem grandes restrições em termos de pedidos-respostas (pode-se enviar e receber o que for necessário), não acontece os problemas de concorrência impostos pelos *browsers*. Esta liberdade de enviar/receber por iniciativa de ambos cliente e servidor, evita a necessidade das estratégias mencionadas em cima para o servidor enviar o que for necessário.

Naturalmente, existem cuidados, e possivelmente algum *overhead* que é necessário adicionar intencionalmente, de forma a possibilitar o uso desta tecnologia. Estas condicionantes irão ser expostas mais à frente, no local apropriado.

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

Figura 1: Pedido de um cliente

```
HTTP/1.1 101 Switching Protocols
Upgrade:  websocket
Connection:  Upgrade
Sec-WebSocket-Accept:  s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol:  chat
```

Figura 2: Resposta de um servidor

3 Funcionamento

Para iniciar uma *WebSocket*, é necessário executar uma "fase de negociação" (conhecida como *handshake*). Esta será feita sobre *HTTP*, acontecendo depois do *handshake* do *TLS* e autenticação caso sejam necessárias.

Para passar uma coneção normal para *WebSocket*(necessário notar que a conexão usada para este *handshake* será a mesma que vai user usada para servir de *WebSocket*) é necessário fazer um pedido inicial através de HTTP, cujo cabeçalho indicará ao servidor que se pretende fazer a comunicação através de *websockets*. O servidor depois responde com sucesso ou insucesso, dependendo se suporta ou não o protocolo.

Nas figuras 1 e 2 podemos ver um exemplo de cabeçalhos *HTTP* usados no *handshake* inicial entre um cliente e um servidor. No fundo, são cabeçalhos HTTP perfeitamente normais, com pares de campos//valor. Podemos ver que o cliente pretende efetuar a comunicação através de um canal *websocket* através do campo *Connection*. O campo *Connection*, quando contem o valor *Upgrade* indica que a comunicação deve passar a ser feita por *websocket*. O campo *Sec-WebSocket-Key* contem um valor codificado em base 64 que é processado pelo servidor, cujo resultado do processamento é enviado para o cliente no campo *Sec-WebSocket-Accept* da resposta. O servidor ao receber esta string concatena um valor constante, volta a converter para base 64 e é interpretado pelo cliente para saber se o cabeçalho da resposta equivale realmente a um cabeçalho de sucesso para *websocket*. O campo *Sec-WebSocket-Protocol* indica quais os sub protocolos é que o cliente pretende utilizar. Na figura, o cliente pretende utilizar os protocolos *chat* e superchat. O servidor, entre todos os sub protocolos que

conhece, escolhe apenas um daqueles que vem no pedido do cliente (desde que o conheça), e coloca o protocolo escolhido no campo *Sec-WebSocket-Protocol* da resposta. Apesar desta verificação, ao receber a resposta do servidor o cliente vai verificar se o protocolo enviado corresponde a algum dos que ele colocou.

Após o *handshake* terminar, a comunicação entre o servidor e o cliente são feitas através de um *websocket*. A partir deste ponto,

4 Vantagens

Como já foi referido, *websockets* permitem que se faça comunicação entre aplicações e servidores web sem se estar preso ás limitações do protocolo HTTP. Isto porque no final do *handshake* os intervenientes da ligação comunicam diretamente sobre o canal TCP aberto, e é nisto em que consistem os *websockets*.

Os *websockets* têm o conceito de mensagens. Nativamente, o protocolo TCP funciona com *streams* de *bytes*. A implementação de *websockets* permite trabalhar num nível acima de *streams*, para abstrair o cliente da necessidade de cuidar da transferência de dados. Fazendo um paralelo com *sockets*: Numa aplicação que utilize *sockets*, uma chamada ao método *recv* iria retornar um conjunto de bytes que podem pertencer a mais que uma mensagem. Com *websockets* temos a garantia de que uma chamada a *recv* retorna não só os bytes de apenas uma mensagem, como retorna todos os bytes da mensagem.

Referências

[1] I. Fette, A. Melnikov (December 2011) *The WebSocket Protocol PROPOSED STANDARD*

http://tools.ietf.org/html/rfc6455