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, houve a tendência de conter componentes *JavaScript*. Inicialmente com o intuito de interagir com o *DOM* através 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 no desempenho, já que permitia obter apenas o conteúdo de interesse da componente servidora sem trazer uma pagina correspondente na integra.

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

Ao longo 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 ausência de todos os *headers/identificadores* que estão presentes durante cada pedido *HTTP*, passando praticamente (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 perceção 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 iniciação, não

[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/`

existe necessidade da passagem de todos os *headers* (que pode ser uma dimensão 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 disponível, 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 possível 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 surgem com o uso de *WebSockets*, já que ao manter uma ligação persistente sem grandes restrições em termos de pedidos e respostas não acontecem os problemas de concorrência impostos pelos *browsers* (limite de ligações activas). 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.

3 Funcionamento

Antes de demonstrar o funcionamento das *WebSockets*, há que ter em conta que o protocolo [4] deixa muito em aberto para as implementações em si, especificando apenas o seu estabelecimento, formato de fragmentos e parâmetros da "sessão". A fase de estabelecimento é conhecida como *handshake*, sendo feita sobre *HTTP*, acontecendo depois do *handshake* do *TLS* e autenticação caso sejam necessárias.

3.1 *handshake*

Para passar uma conexão normal para *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. É de notar que a conexão usada para este *handshake* será a mesma que vai ser usada para servir de *WebSocket*.

```
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-Extensions: permessage-deflate
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
Sec-WebSocket-Extensions: permessage-deflate
```

Figura 2: Resposta de um servidor

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

[4] `http://tools.ietf.org/html/rfc6455`

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*.

Ambos os campos *Sec-WebSocket-Protocol* e *Sec-WebSocket-Extensions* são parametros negociáveis, tendo o cliente de especificar quais os sub-protocolos e extensões pretende utilizar. Na figura, o cliente pretende utilizar os protocolos *chat* e *superchat* e extensão *permessage-deflate*. 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, enquanto em relação às extensões, responde no campo *Sec-WebSocket-Extensions* com as extensões que suporta. Continua a ser necessário que o cliente verifique se o protocolo enviado corresponde a algum dos que ele colocou.

Após o *handshake* terminar, o *WebSocket* estará estabelicido e o envio de mensagens poderá começar.

3.2 Formato de mensagens/fragmentos

Um conceito importante, e que por extensão deve estar presente quando se transferem dados em ambas direções, é que o recetor só tem acesso aos dados quando estes são recebidos na totalidade (os dados na totalidade são chamados de mensagens), sendo este um dos principais fatores que leva a grande parte das implementações de *WebSockets* serem *event-based*.

Estas mensagens durante transporte estão divididas em *frames* (numero ilimitado), que podem também ser fragmentadas (estes fragmentos podem mudar durante os diversos intermediários presentes na transferência). Cada *frame* tem a noção se corresponde à conclusão da mensagem, mas nunca de quantas *frames* ainda estão em falta, sendo esta a razão porque o recetor só pode ler a mensagem quando a ultimo *frame* é recebido. No protocolo *HTTP*, esta conclusão podia ser tirada a partir do cabeçalho *Content-length*.

3.3 Sub-protocolos e extensões

Devido à simplicidade das mensagens, o protocolo conta que os utilizadores definam um formato para as mensagens, já que ao tirar proveito da existência de uma conexão para diversos propósitos, é necessário encontrar um "substituto"para o par pedido//resposta do protocolo *HTTP* (novamente, nada impede que existam diversas respostas para o mesmo pedido).

Como exemplo, temos *JSON-RPC 2.0*[5] e *WAMP*[6], que usam *JSON* como formato das mensagens e identificadores para relacionar os diversos pedidos e respetivas respostas.

[5] `http://www.jsonrpc.org/specification`
[6] `http://wamp.ws/`

Em relação a extensões, podemos as considerar como formas de expandir o protocolo já existente, ao adicionar compressão (extensão *permessage-deflate*[7] ou ao adicionar múltiplos canais numa única conexão (extensão *Websocket-multiplexing*[8]).

4 Possíveis problemas

Os *Websockets* como vimos, trazem um conjunto de facilidades/melhorias, mas como é de esperar, traz também um conjunto de problemas cujos utilizadores terão de ter em conta quando desenvolvem as suas aplicações.

Devido à restrição de ser obrigado a obter os dados na sua totalidade (mensagens completas), há que ter o cuidado com o tamanho dos mesmos, devido ao *buffering* necessário, e problemas de memória que podem emergir.

4.1 Interação com terceiros

À procura de melhorias de desempenho, na *internet* estão presentes mecanismos que até agora apenas tinham em conta o protocolo *HTTP*, como *proxies* (que permitem filtrar conteúdo, facilitar *caching*, etc), *load balancers* que são importantes no âmbito de sistemas distribuídos (redirecionam os pedidos para quem está disponível para os executar) ou *firewalls*.

Tendo em conta que a única componente semelhante a *HTTP* é o *handshake*, é necessário ter em conta se estes mecanismos irão continuar a funcionar como suposto com *WebSockets* (alterar os mesmos pode não ser simples/possível). Em relação a *firewalls*, não há nenhum impedimento já que se continua a usar a porta 80 e 443, apesar do mesmo não se poder dizer de *proxies* e *load balancers*.

No caso dos *proxies*, dependendo da configuração, alguns dos cabeçalhos podem ser removidos, o que é problemático caso o cabeçalho *upgrade* durante o *handshake* não seja propagado (resultando em falha). Novamente dependendo da configuração, os *proxy's* podem não propagar o pedido se o conteúdo não for *HTTP*, este problema pode ser evitado usando *WebSocket-secure*, já que os *proxies* por norma propagam dados que estejam encriptados.

Em relação a *load balancers*, dependendo da camada onde estes actuam, podem surgir problemas. No caso de atuarem sobre a camada de transporte, continuarão a atuar corretamente. Caso atuassem sobre a camada de aplicação, modificações serão necessárias, já que é necessário assegurar que a mensagem será redirecionada sempre para o mesmo recetor.

[7] `http://tools.ietf.org/html/draft-ietf-hybi-permessage-compression-00`
[8] `http://tools.ietf.org/html/draft-ietf-hybi-websocket-multiplexing-11`