Enriquecimento de plataformas web colaborativas com comunicação browser-a-browser

Albert Linde, João Leitão e Nuno Preguiça

NOVA LINCS & DI-FCT-Universidade Nova de Lisboa

Resumo A popularidade das aplicações colaborativas na Web tem crescido significativamente nos últimos anos. Estas incluem ferramentas de edição como o Google Docs, Microsoft Office 365, assim como outros tipos de plataformas Web como as redes sociais. Apesar de todas estas aplicações serem centradas nos utilizadores, estas continuam a recorrer a um modelo de interação mediado por uma componente centralizada, que, para além de ser um ponto de contenção do qual depende todas as interações entre os clientes, também leva a que a propagação de conteúdos entre os utilizadores seja afetada por grandes latências de comunicação. Para ultrapassar estas limitações, propomos enriquecer as arquiteturas atuais com suporte para comunicação direta browser-a-browser. Para tal, apresentamos o desenho de um sistema que permite a aplicações a executarem em múltiplos browsers manterem réplicas de um conjunto de objetos, os quais podem modificar concorrentemente. Os browsers propagam as modificações efetuadas diretamente usando WebRTC, recorrendo a um servidor como intermediário quando tal não é possível. A convergência das réplicas é alcançada usando uma solução baseada em CRDTs. Para demonstrar os benefícios da solução proposta, implementámos um sistema que expõe uma API semelhante ao Google Drive Realtime, e apresentamos resultados experimentais que suportam a nossa proposta.

1 Introdução

Nos últimos anos temos assistido à proliferação e ao aumento da popularidade de aplicações web. Muitas destas aplicações são centradas em utilizadores, e muitas apresentam aspetos colaborativos ou proporcionam suporte para a interação directa entre os seus utilizadores. Exemplos relevantes destas aplicações são as redes sociais, aplicações de troca de mensagens escritas, ou ferramentas de produtividade colaborativas como o GoogleDocs ou o Mircrosoft Office 365.

A complexidade destas aplicações tem vindo a aumentar, usufruindo das novas funcionalidades e capacidades dos *browsers*. De facto, e graças a popularidade crescente de linguagens como Javascript, muitas destas aplicações executam a maior parte do seu código diretamente no browser do utilizador.

No entanto, e apesar do aumento significativo das capacidades dos browsers e da natureza centrada no utilizador de uma parte significativa destas aplicações, estas continuam assentes sobre o modelo cliente-servidor, no qual toda a interação entre os utilizadores é mediada através de um servidor centralizado. As limitações do modelo cliente-servidor são conhecidas e incluem: i) o facto de incorrer em latências significativas entre utilizadores, especialmente aqueles que se encontram geograficamente próximos (ou até numa única rede local de uma empresa ou outra instituição); ii) o

servidor central é um potencial ponto de contenção que pode limitar a escalabilidade do sistema em termos de número de utilizadores que podem ser servidos devido ao limite de recursos na componente centralizada; e iii) o servidor constituí um ponto único de falha que impede qualquer interação entre os utilizadores no caso em que o servidor deixa de estar disponível.

Factualmente, o advento da computação em nuvem veio mitigar de forma significativa algumas destas limitações, nomeadamente no que diz respeito aos limites de escalabilidade e de tolerância a falhas devido à maior facilidade em recorrer a técnicas de replicação e de elasticidade (*i.e*, a capacidade de aumentar o número de servidores que suportam uma aplicação de acordo com a carga, utilizando técnicas de particionamento). No entanto, notamos que o recurso a infraestruturas na nuvem incorre em custos monetários para os operadores das aplicações que são indesejáveis para estes, ou até mesmo no caso de pequenas e médias empresas que se tentam lançar no mercado já de si bastante competitivo, incomportáveis.

Neste trabalho, quebramos com o modelo comum para o desenho de aplicações web, e propomos uma técnica complementar ao uso de infraestruturas na nuvem para ultrapassar as limitações do modelo cliente-servidor no desenho destas aplicações, com impacto direto sobre a escalabilidade e disponibilidade destes sistemas, a latência sentida por utilizadores geograficamente próximos, e ainda o custo operacional das componentes centralizadas. A ideia principal que guia o nosso trabalho parte da observação de que recentemente foram introduzidas um conjunto de novas tecnologias nos browsers que permitem a comunicação directa e transparente entre os mesmos, seguindo uma metodologia entre pares (do inglês *peer-to-peer* [17,12]) a que no contexto deste trabalho chamamos comunicação *browser-a-browser*. Estas tecnologias incluem o WebRTC [18] (Web Real Time Communication), e a proposta de protocolos de suporte ao contorno de firewall e NATs como o STUN [7] e TURN [8]. Estas novas tecnologias combinadas com novos avanços propostos no HTML5, abrem as portas ao desenvolvimento de frameworks, que permitem o enriquecimento de aplicações web com comunicação direta entre os dispositivos dos utilizadores.

Assim, para ultrapassar as limitações das arquiteturas centralizadas, neste artigo propomos enriquecer as arquiteturas atuais com suporte para comunicação direta browser-a-browser usando as novas tecnologias disponíveis nos browsers. Para tal, apresentamos o desenho de um sistema que permite a aplicações a executarem em múltiplos browsers manterem réplicas de um conjunto de objetos, os quais podem modificar concorrentemente. Este sistema é composto pelos seguintes componentes principais: i) mecanismos para a gestão de filiação que permite a instâncias de uma aplicação web, de acordo com uma política específica a essa aplicação, estabelecer ligações diretas entre os browsers dos utilizadores que interagem mais frequentemente; ii) um conjunto de primitivas de comunicação descentralizadas que permitem às aplicações trocarem informação entre si, e coordenarem-se por forma a propagar a informação para a componente centralizada para alcançar durabilidade, e permitir o acesso aos dados por utilizadores que façam uso de browsers sem suporte para WebRTC; iii) uma biblioteca de CRDTs [16,14] escrita em javascript para simplificar o desenho de aplicações web descentralizadas que executam no browser garantido que o estado da aplicação converge eventualmente.

O sistema que apresentamos neste artigo expõe para as aplicações uma API semelhante ao Google Drive Realtime, a qual é usada para a criação de aplicações colaborati-

vas na Web nas quais um conjunto de utilizadores colaboram partilhando um conjunto de objetos que podem ser atualizados concorrentemente. Além desta solução permitir que se possam usar aplicações já existentes de forma mais simples, permitiu-nos ainda comparar a solução proposta neste artigo com uma solução que apenas usa a componente centralizada. Os resultados obtidos mostram que a solução proposta neste artigo apresenta uma latência bastante inferior.

O artigo está organizado da seguinte forma: a secção 2 apresenta uma visão geral da nossa proposta de arquitetura; na secção 3 discutimos com maior detalhe os componentes principais da arquitetura proposta e interação entre os mesmos; a secção 4 discute detalhes de implementação do nosso protótipo; a secção 5 apresenta a nossa avaliação experimental e uma discussão dos resultados; o trabalho relacionado é abordado na secção 6 e finalmente, a secção 7 conclui o artigo e discute brevemente direções futuras para o trabalho apresentado.

2 Visão Geral

Nesta secção apresentamos uma perspetiva geral sobre a arquitetura proposta para enriquecer a operação de aplicações web colaborativas com comunicação direta browser-a-browser. Note-se que nesta arquitetura a componente centralizada continua a ser utilizada, nomeadamente para garantir a persistência dos dados da aplicação, mas também para suportar clientes que executem a aplicação num browser que não suporte comunicação direta entre browsers, ou para clientes que devido a políticas restritivas de firewalls ou Network Address Translations (NAT), sejam incapazes de receber ligações de outros clientes. Adicionalmente, e como ficará mais claro no resto do artigo, a componente centralizada é também ela alavancada para permitir aos clientes estabelecerem as ligações entre si quando se juntam ao sistema. No entanto começamos por discutir o modelo de interação considerado neste trabalho.

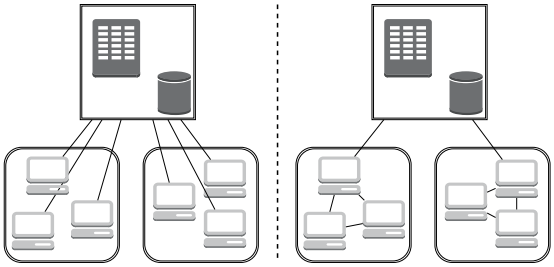


Figura 1: Modelos de comunicação (cliente-servidor) | (browser-a-browser)

2.1 Modelo de Interação

A Figura 1 captura um paralelo entre o modelo de interação comum das aplicações web (esquerda) e o modelo que propomos neste artigo (direita). Como referido anteriormente,

no modelo clássico todas as interações entre clientes são mediadas através de uma componente centralizada, que é também responsável por servir todos os objetos de dados da aplicação e processar as operações que mudem o estado desses objetos. Na nossa proposta estendemos este modelo tradicional oferecendo a hipótese dos clientes (principalmente aqueles que se encontram próximos, e.g, numa mesma rede local) servirem réplicas dos objetos partilhados diretamente, e também de operarem sobre esses objetos localmente, propagando entre eles as atualizações. Esta aproximação permite reduzir substancialmente a dependência da componente centralizada.

Em maior detalhe, o modelo de interação que consideramos neste trabalho é o seguinte. Em primeiro lugar um utilizador recorre ao seu browser para aceder a uma aplicação web, a qual está acessível num servidor web. Ao obter a aplicação web, o browser do utilizador obtém também, de forma transparente, a nossa framework, que é distribuída como código JavaScript, e que é incluído na aplicação web. Após carregar estes recursos, o browser do cliente gera um evento (*onLoad*), que é utilizado pela aplicação para inicializar a nossa framework.

Ao ser inicializada, a nossa framework estabelece uma ligação através de uma WebSocket [9] para um processo servidor, que designamos de *B2BServ*. Este servidor, escrito em NodeJS, é responsável por auxiliar os vários clientes a estabelecerem ligações entre si. A lógica que determina se dois clientes devem ou não estabelecer uma ligação direta browser-a-browser é dependente da aplicação. No entanto, e como discutimos mais tarde, imaginamos um conjunto reduzido de políticas que podem ser benéficas a vários tipos de aplicações. O servidor ao receber o registo de um novo cliente, propaga o pedido de ligação a clientes que já estão ligados aos servidor, mediando assim a ligação inicial entre os clientes.

O novo cliente é então responsável, utilizando o servidor como mediador, por tentar estabelecer as ligações diretas com os vizinhos. Para este fim recorremos aos mecanismos oferecidos pelo Web Real Time Communication (WebRTC) [18], que por sua vez pode recorrer a servidores STUN e TURN como suporte para ultrapassarem firewalls e NATs que possam dificultar o estabelecimento de ligações. Desde o momento da ligação inicial ao servidor a aplicação pode começar a solicitar à framework que esta obtenha cópias de recursos associados à aplicação – por exemplo, no caso de uma aplicação de chat, um recurso pode ser uma lista de mensagens trocadas em cada sala de chat a que o utilizador queira aceder; alternativamente no contexto de uma aplicação de edição colaborativa um recurso pode ser o conteúdo atual de um documento partilhado. Quando um cliente consegue efetuar ligações a outros clientes a comunicação é redirecionada, de forma transparente ao utilizador, para um modelo de interação direta entre clientes.

Os dados são partilhados e replicados entre os clientes usando os CRDTs apropriados para esse tipo de dados. Os CRDTs permitem minimizar a coordenação entre os clientes quando executam operações sobre as suas réplicas locais. Para além disso, permitem garantir de forma simples para o programador, que os objetos replicados entre os clientes convergem para um estado final comum, e também que os clientes têm a oportunidade de agregar várias operações de vários vizinhos numa única mensagem para ser propagada para o repositório central, criando a oportunidade de minimizar a carga imposta sobre esta componente.

Desta forma conseguimos reduzir a latência entre os clientes que estão próximos e reduzir a carga no servidor quando grupos de clientes colaboram entre si e agregam

pedidos. Também torna-se possível evitar o ponto central de falha. Nas arquitecturas actuais, o servidor é o único ponto de comunicação e quando este fica indisponível a interação para. Quando deixamos os clientes interagir diretamente uns com os outros, a interação pode continuar mesmo com falhas temporárias do servidor.

De seguida, e dado este modelo de interação, explicamos em maior detalhe a arquitetura proposta e as responsabilidades de cada uma das suas componentes.

2.2 Arquitectura

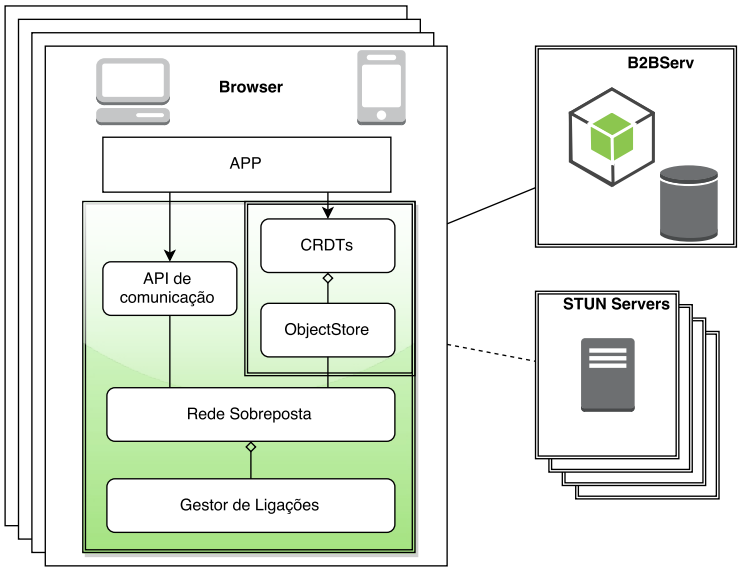


Figura 2: Arquitectura Proposta

Dada a visão geral de interação descrita acima, a Figura 2 esquematiza a arquitetura do sistema que propomos neste artigo, a qual pode ser dividida em dois grandes subcomponentes: o repositório de objetos e o sub-sistema de comunicações. O repositório de objetos é responsável por gerir os objetos a que a aplicação acede, utilizando o subsistema de comunicação para propagar e receber as atualizações efetuadas. O subsistema de comunicações é responsável por efetuar todas as comunicações com outros browsers e com a componente centralizada do nosso sistema.

Uma aplicação a executar no browser, tipicamente escrita em JavaScript, é responsável por interagir com a camada de apresentação (a página web) e por fazer uso da API oferecida pela nossa framework. Esta API permite comunicar diretamente com outros clientes e manter réplicas sincronizadas de um conjunto de objetos partilhados. Para tal, a aplicação deve inicializar a nossa framework quando esta inicia.

Subsistema de comunicação: O subsistema de comunicação é composto pelos seguintes módulos a executarem no cliente:

API de Comunicação: Oferece um conjunto de primitivas de comunicação utilizadas para comunicar com outros clientes (recorrendo à informação mantida pelo módulo *Redes Sobreposta*). A comunicação é feita através de trocas de mensagens codificadas no formato JSON.

Módulo de Gestão da Rede Sobreposta (Rede Sobreposta): Esta componente mantém informação sobre redes sobrepostas usadas para propagar informação entre clientes. Para tal, usa-se a abstração de *grupos* de clientes, em que cada grupo tem associado um conjunto de objetos replicados. Esta componente mantém informação sobre os vizinhos lógicos (i.e, outros clientes) do cliente para cada grupo e implementa algoritmos de disseminação eficientes de mensagens para um grupo, recorrendo às ligações geridas pelo Gestor de Ligações. As operações de disseminação de mensagens para um grupo são utilizadas para garantir a propagação eventual de operações realizadas sobre os objetos replicados em cada grupo.

Gestor de Ligações: Esta componente é responsável por criar e gerir as ligações entre os vários clientes e também com a componente centralizada. A gestão da criação de ligações por esta componente permite evitar o uso de ligações redundantes quer entre clientes quer com a componente centralizada. Para manter ligações entre clientes, esta componente recorre à API oferecida pelo WebRTC. Para manter ligações à componente centralizada usam-se WebSockets. Esta componente colabora ativamente com o módulo de gestão da rede sobreposta por forma a permitir a um cliente juntar-se a um ou vários grupos, de acordo com as necessidade da aplicação e das ações do utilizador.

Para suportar os serviços disponibilizados nos clientes e permitir a comunicação browser-a-browser, o nosso sistema inclui os seguintes módulos a executar na componentes centralizada, potencialmente num ambiente de computação em nuvem:

Componente Centralizada (B2BServ): A componente centralizada é materializada através de um processo escrito em NodeJS. Este servidor recebe ligações produzidas pelos módulos de gestão dos vários clientes (através de WebSockets) e atua como um ponto de entrada dos vários clientes à nossa framework. Em particular esta componente atua como mediador durante o estabelecimento de ligações diretas entre os clientes. Para além disso esta componente é responsável por mediar o acesso dos clientes à camada de persistência centralizada do sistema, isto inclui providenciar o acesso a objetos que não se encontram ainda disponíveis nos clientes, e receber atualizações desses objetos para garantir o seu correto armazenamento na camada (centralizada) de persistência. Esta componente é também responsável por servir os clientes que não tem a capacidade de estabelecer ligações browser-a-browser com outros clientes.

Servidores STUN: Estes servidores encontram-se fora do controle da nossa solução mas são usados para permitir a clientes que se encontrem atrás de firewalls ou NATs o estabelecimento de ligações browser-a-browser diretas. Apesar de que o

operador da aplicação pode correr os seus próprios servidores STUN, no contexto deste trabalho recorremos aos servidores publicamente disponíveis da Google[1].

Repositório de objetos: As aplicações a executarem em diferentes browsers partilham estado recorrendo a um repositório de objetos partilhados. Este repositório de objetos, presente em cada browser, é composto pelos seguintes módulos principais:

Biblioteca de CRDTs (CRDTs): Este módulo oferece um conjunto de tipos de dados baseados nos princípios dos CRDTs codificados em JavaScript. Esta biblioteca é usada simultaneamente pela camada da aplicação para aceder aos dados relevantes da aplicação, e também pelo módulo de gestão de réplicas que materializa as réplicas locais sob a forma de CRDTs. Esta biblioteca pode ser estendida pelo próprio programador por forma a definir novos tipos de CRDTs (por exemplo, que combinem vários dos tipos oferecidos pela framework) que melhor se ajustem às necessidades das suas aplicações.

Módulo de gestão de réplicas: Este módulo é responsável por utilizar os serviços disponibilizados pelo subsistema de comunicação para obter o estado inicial dos objetos e propagar e receber as atualizações efetuadas nos objetos. Adicionalmente, utiliza mecanismos de armazenamento persistente do HTML5 para armazenar de forma persistente as cópias dos CRDT.

3 Suporte B2B

Nesta secção detalham-se os dois subsistemas que compõem a nossa solução: o repositório de objetos e o subsistema de comunicações.

3.1 Repositório de objetos

O repositório de objetos permite, a uma aplicação a executar num browser, a partilha de objetos com outras aplicações a executarem remotamente também em browsers - tipicamente, instâncias da mesma aplicação. Estes objetos partilhados encapsulam os dados manipulados pela aplicação, permitindo aos utilizadores cooperar através da modificação concorrente desses mesmos objetos.

Biblioteca de CRDTs A versão atual do nosso sistema permite partilhar os mesmos objetos que estão disponíveis no Google Drive Realtime [6]: String, Lista e Mapa. Uma *String* mantém uma sequência de caracteres, a qual pode ser modificada através da inserção ou remoção de caracteres. Uma *Lista* mantém uma sequência de valores primitivos ou objetos JavaScript constantes e (referências para) objetos partilhados (Lista, String ou Mapa). Um *Mapa* mantém uma relação entre uma chave, representada por uma string constante, e um valor primitivo ou objeto JavaScript constantes ou (uma

[1] Existe também a hipótese de recorrer a servidores TURN, que reencaminham tráfego entre nós quando, devido a configurações locais de Firewall ou NAT, estes não conseguem estabelecer ligações diretas entre si. No entanto no contexto deste trabalho optámos por não fazer uso destes servidores, o que implica que clientes que não consigam estabelecer ligações entre si, recorrem unicamente aos mecanismos disponibilizados pela componente centralizada.

referências para) um objeto partilhado (Lista, String ou Mapa). Na implementação destes objetos procurou-se fornecer a mesma interação para as aplicações que é fornecida pelo sistema Google Drive Realtime, de forma a simplificar o processo de conversão deste tipo de aplicação para o uso da solução proposta neste artigo.

Ao contrário da solução usada pelo Google Drive Realtime, que recorre a um algoritmo de transformação de operações com servidor central [13], a solução proposta neste artigo usa CRDTs baseados na propagação do estado [16]. Esta solução permite que quaisquer dois clientes possam sincronizar diretamente o estado das suas réplicas em qualquer momento. Assim, os clientes deixam de estar dependentes da componente centralizada e podem, num passo de sincronização, propagar um estado que reflete um conjunto de modificações.

Módulo de gestão de réplicas O módulo de gestão de réplicas tem duas responsabilidades: gerir a persistência dos objetos partilhados usados pelas aplicações e manter esses mesmos objetos atualizados. Para manter os objetos de forma persistente entre sucessivas utilizações da mesma aplicação Web, o sistema recorre aos mecanismos de persistência disponíveis no HTML 5, que permitem a persistência local de estado (ainda que o browser seja terminado).

Para manter os objetos atualizados, o sistema recorre às primitivas de comunicação disponibilizadas pelo subsistema de comunição. Assim, em cada grupo podem-se manter vários objectos partilhados onde a rede sobreposta é usada para propagar as novas versões dos objectos. Na versão atual, usa-se uma solução de propagação lenta de novas versões para permitir a acumulação de múltiplas operações de escrita a cada passo de propagação entre clientes. Essas escritas são encapsuladas numa nova versão do objecto, que é propagada para os elementos do grupo de forma colaborativa como abordado de seguida.

3.2 Subsistema de comunicação

Descrevemos agora alguns detalhes relativos ao desenho das componentes da nossa framework que lidam com os aspetos relativos à comunicação browser-a-browser, e discutimos também o papel da componente centralizada na gestão destas ligações.

Módulo de Gestão das ligações A comunicação direta browser-a-browser é alcançada através do uso da WebRTC API. O WebRTC tem suporte nativo nos browsers mais usados atualmente, nomeadamente o Chrome e o Firefox, sendo possível recorrer a plugins para suportar WebRTC noutros browsers. Um dos motivos principais que motiva o uso de WebRTC é exatamente o suporte nativo nestes browsers, visto que isso permite o uso da nossa framework sem qualquer interação ou esforço (i.e, instalar software ou configurar firewall e NAT) por parte do utilizador final. O WebRTC recorre sempre a conexões seguras, em que todos os dados transmitidos entre dois browsers são cifrados, minimizando assim a exposição dos utilizadores a violações de privacidade.

A existência de Firewalls e NATs sempre constituíram um problema no uso de sistemas entre pares no passado, uma vez que dificultam (ou impedem) a criação de ligações e comunicação direta entre nós que se encontram atrás de um destes componentes. Para contornar este problema, o WebRTC recorre a serviços especialmente desenhados para facilitarem o estabelecimento de ligações nestes cenários. Existem dois tipos de

serviços que podem ser usados. O STUN (que é reflexivo) e permite a ambos os clientes determinarem os seus IPs públicos (no caso de NAT) e instalar estado nas firewalls ou NATs para permitir comunicação direta. A segunda alternativa passa pelo uso do serviço TURN, em que os servidores que providenciam este serviço apenas redirecionam o tráfego entre os dois nós.

No trabalho apresentado neste artigo, o uso do serviço TURN foi inibido, visto que nos cenários em que nos focamos, é mais eficiente para os clientes comunicarem indiretamente através da componente centralizada do sistema do que terem toda a sua comunicação direta redirecionada para um servidor que a aplicação não controla. Nestes casos, assim como no caso em que o browser não suporta WebRTC, a framework utiliza o modelo de interação típico das aplicações Web, em que todas as ações dos utilizadores são mediadas pela componente centralizada.

De forma a estabelecer uma ligação direta browser-a-browser, o WebRTC estabelece um protocolo de *Signalling* que requer a ambos os clientes a troca de informação inicial entre si sob a forma de *offers* (em formato textual). Qualquer mecanismos que permita a troca de texto pode suportar a execução do protocolo de *Signalling*. Na nossa framework, optámos por recorrer à componente centralizada, utilizando WebSockets, para efetuar a troca de informação necessária ao estabelecimento destas ligações. Esta decisão prende-se com o facto de que a maioria de serviços Web requer um processo de autenticação por parte do utilizador, o que requer necessariamente o envio de informação para a componente centralizada no início da sessão.

Componente da Rede Sobreposta Como discutido anteriormente, os vários clientes da aplicação organizam-se no contexto de grupos. Para tal a componente de filiação da nossa framework oferece uma operação de Join que requer o identificador desse grupo. Para permitir a um cliente juntar-se a um grupo com identificador $\mathcal{G}$, a nossa framework envia um pedido à componente centralizada (através da ligação WebSocket) com esse identificador e solicita a filiação corrente do grupo. Este pedido carrega em *piggyback* a mensagem desse cliente para a execução do protocolo de *Signalling* do WebRTC.

Ao receber este pedido a componente centralizada, que mantém informação (parcial) relativa à filiação dos grupos, envia ao cliente o subconjunto de membros do grupo $\mathcal{G}$, e notifica também os restantes elementos do grupo sobre a chegada do novo cliente. As mensagens enviadas para cada cliente pela componente centralizada serve também para enviar a informação (originalmente enviada por cada cliente) para a execução do protocolo de *Signalling*. Ao receberem estas notificações, cada cliente recorre então à componente de gestão da nossa framework para materializar as novas ligações.

O resultado final deste processo é que os clientes, para cada grupo, estabelecem uma rede-sobreposta similar a um grafo aleatório e com ligações bi-direcionais, com propriedades similares às redes sobrepostas mantidas por algoritmos utilizados no contexto dos sistemas entre pares como o Cyclon [17] ou o HyParView [12].

Para evitar sobrecarregar a componente centralizada com atualizações excessivas relativas às operações realizadas pelos clientes sobre os objetos que replicam localmente, apenas um pequeno subconjunto dos clientes mantém a ligação à componente centralizada. Para decidir que clientes são responsáveis por esta tarefa recorremos a um algoritmo de Bully para eleição de líderes locais [2]. Neste algoritmo cada cliente começa num estado inicial *ativo* e periodicamente emite para todos os seus vizinhos uma mensagem contendo o seu identificador. Sempre que um cliente recebe uma mensagem

de outro cliente cujo identificador seja menor que o seu, muda o seu estado para *inerte*, pára de transmitir esta mensagem periódica e termina a sua ligação com a componente centralizada. Um nó *inerte* que não receba mensagens destas por um período suficientemente grande, muda o seu estado para *ativo* e restabelece a ligação para a componente centralizada. Apenas os nós ativos são responsáveis por interagir com a componente centralizada.

Esta componente é também responsável por suportar os mecanismos de comunicação disponibilizados pela API de comunicação. Para tal existem mecanismos para comunicação ponto-a-ponto, e também mecanismos de difusão para um grupo. Estes últimos recorrem a mecanismos de difusão epidémica bem conhecidos para propagaram informação entre todos os elementos do grupo [1].

API de comunicação A API de comunicação oferece mecanismos que permitem à aplicação (e às restantes componentes da framework) comunicarem com outros clientes no contexto de um grupo. Esta API oferece mecanismos para o envio de mensagens ponto-a-ponto e multicast, sempre no contexto de um grupo a que o cliente pertença. As mensagens são sempre propagadas usando uma política de melhor esforço.

Componente Centralizada Como discutido anteriormente, a componente centralizada é materializada por um servidor aplicacional escrito em NodeJS. Este é responsável por manter informação sobre os grupos de comunicação ativos em cada momento, conhecendo em cada grupo um pequeno conjunto de elementos que se encontrem ativos (os nós que se mantém num estado *ativo* após a execução do algoritmo de *bully* ao nível da componente de filiação).

Esta componente é também responsável por receber as alterações efetuadas pelos clientes sobre os dados da aplicação e garantir a persistência dessas alterações na componente centralizada. Para além disso o servidor serve de ponto de contacto para facilitar o estabelecimento de ligações browser-a-browser e opera como ponto de acesso à aplicação para os clientes que não consigam estabelecer ligações diretas browser-a-browser.

4 Detalhes de Implementação

De forma a demonstrar a exequibilidade da nossa proposta desenvolvemos um protótipo que usaremos de seguida no nosso trabalho experimental. O nosso protótipo, incluindo código da componente centralizada, foi implementado em 3.863 linhas de código JavaScript. O código JavaScript utilizado para materializar a componente da framework que executa no cliente, quando compactado de forma automática[2], fica com um tamanho total de 39 KB.

As páginas web e o código JavaScript das aplicações e da framework pode ser servidas por qualquer servidor Web. A autenticação perante o serviço Web é da responsabilidade do programador. Nas nossas experiências efetuamos a autenticação ao nível do B2BServ na componente centralizada.

A nossa implementação do B2BServ suporta particionamento desta componente em vários processos para garantir a escalabilidade do sistema para grandes números de

[2] Recorrendo à ferramenta disponível em `https://github.com/mishoo/UglifyJS2`.

clientes e/ou objetos a serem acedidos. A lógica de particionamento é guiada através dos identificadores de grupos.

Simplificamos a implementação do nosso componente de gestão de filiação, e ao invés de recorrer a um grafo aleatório para manter as ligações entre os vários clientes de cada grupo, mantemos todos os clientes de um grupo ligados entre si num grafo totalmente conexo. Esta simplificação foi feita para minimizar o tempo de desenvolvimento do protótipo e será endereçada em trabalho futuro.

O módulo de gestão de objetos foi desenhado especificamente para lidar com objetos do tipo CRDT. Foram implementadas funções para obter, criar e apagar objetos. A implementação desta componente assume que todos os objetos devem ter definidas funções *compare* e *merge*, que são típicas às interfaces de CRDTs. A biblioteca de CRDTs fornecida atualmente com a framework providencia implementações de Strings, Listas e Mapas descritas em [16].

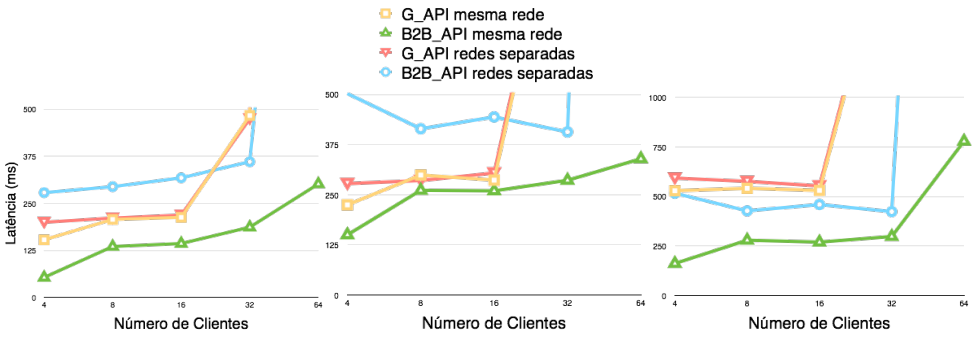
A camada de rede sobreposta suporta dois mecanismos de disseminação, um primeiro baseado em inundação em que todos os clientes enviam as mensagens para todos os outros clientes, e um segundo modo em que os clientes trocam informação através dos nós cujo estado permanece como *ativo* devido à execução do protocolo de *bully* anteriormente descrito. Este último protocolo pode ser parametrizado com o tempo entre transmissões dos identificadores dos nós em estado *ativo*. A sincronização do estado das réplicas dos clientes com a componente centralizada é também configurável, sendo tipicamente de vários segundos para permitir aos clientes responsáveis por esta operação a possibilidade de agregarem operações de vários clientes em cada passo de sincronização com o servidor.

De forma a testar o protótipo do sistema foram desenvolvidas várias aplicações de demonstração. Estas aplicações permitiram também aferir a facilidade do uso da nossa framework. Sem contabilizar o número de linhas de código para implementar a interação com o ambiente web (i.e, JavaScript para manipular o conteúdo da página HTML) as aplicações desenvolvidas incluem os seguintes exemplos: uma aplicação de chat com salas, que foi escrita em quatro linhas de código; uma lista que pode ser editada colaborativamente, a qual foi desenvolvida em seis linhas de código, entre outros exemplos. Omitimos os detalhes relativos às aplicações de exemplo devido a limites de espaço.

5 Avaliação

Para avaliar o nosso sistema comparamos o seu desempenho com o desempenho obtido utilizando uma aplicação que recorre ao Google Realtime API. Relembramos que a Google Realtime API (G_{API}) suporta os mesmos objetos, os quais podem ser partilhados por múltiplos clientes. Todos os nossos testes foram realizados utilizando instâncias *m3.xlarge* da Amazon Web Services EC2.

Em todos os testes colocámos um único servidor na zona us-west-1 (Califórnia) e dividimos os clientes de forma igual entre 16 máquinas, 8 na zona us-west-2 (Oregon) e 8 na zona us-east-1 (Virgínia). Desta forma dividimos os clientes em dois grupos de igual tamanho (cujo tamanho variamos nas nossas experiências), e apenas permitimos ligações diretas browser-a-browser entre máquinas localizadas na mesma região (i.e,



(a) Média (b) Percentual 95 (c) Máxima

Figura 3: Latência entre clientes.

na mesma rede local). Esta restrição não existe na nossa framework, no entanto esta configuração promove ligações entre clientes de baixa latência.

Em todas as experiência recorremos ao servidor na Califórnia para atuar como servidor Web e executar o processo B2BServ. Todos os clientes executam o cliente no Chromium, a base do Google Chrome de código aberto, em modo headless que registas as suas actividades através do Xvfb, um servidor de écran virtual, em memória.

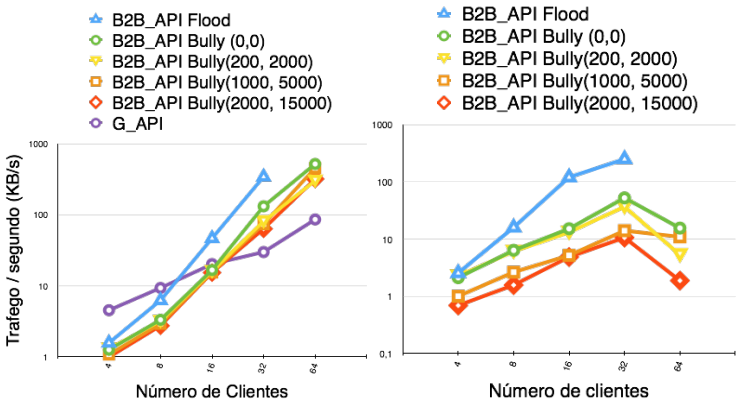
Os objetivos principais do nosso trabalho experimental são os seguintes: i) avaliar a latência observada pelos vários clientes em ambos os sistemas (secção 5.1); ii) medir a quantidade de dados trocados entre o servidor e os clientes em ambas as soluções e entre os clientes na nossa solução (secção 5.2); e iii) avaliar o suporte da nossa framework à operação desconetada do servidor (secção 5.3).

5.1 Latência entre clientes

Neste teste usamos um cliente que efetua escritas (localizado na Virgínia) e medimos o tempo até as escritas serem propagadas a todos os outros clientes. O cliente que efetua escritas escreve um carácter numa *string* em cada uma das suas operações.

Apresentamos os resultados de latência entre clientes numa mesma rede local (i.e, que recorrem a comunicação direta browser-a-browser) e clientes em redes separadas (i.e, atualizações têm que passar pela componente centralizada). Para facilitar a compreensão destes resultados indicamos de seguida os valores de latência entre as várias máquinas usadas neste teste, medida com a ferramenta `ping`: i) entre Oregon e o servidor na Califórnia: 20 ms; ii) entre Virgínia e o servidor na Califórnia: 80 ms; iii) entre Oregon e o servidor da Google: 13 ms; iv) entre Virgínia e o servidor da Google: 12 ms; e v) entre Oregon e Virgínia: 70 ms.

A Figura 3(a) apresenta a latência média, que se apresenta, como esperado, substancialmente mais baixa no caso em que os clientes comunicam diretamente através de ligações browser-a-browser. As figuras 3(b) e 3(c) apresentam os valores de latência para o percentil 95 e o valor máximo respetivamente. Estes resultados corroboram os resultados anteriores e mostram que no caso da comunicação mediada pela componente



(a) Trafego no servidor (b) Trafego por cliente (média)

Figura 4: Trafego

centralizada, os valores de latência são substancialmente mais elevados. Mais ainda, estes resultados mostram que a latência sentida pelos clientes quando recorrem à nossa framework não sofre variações significativas com o aumento de número de clientes. Os resultados mostram igualmente que o sistema baseado na Google Realtime API é muito sensível ao número de clientes, sendo que a latência aumenta substancialmente com o número total de clientes.

5.2 Tráfego

Para medir o tráfego gerado por cada alternativa, efetuamos duas escritas por segundo em todos os clientes durante um curto período de tempo, variando o número de clientes. Para além disso variamos também o protocolo de disseminação de mensagens entre os clientes recorrendo à disseminação por inundação (*Flood*) e recorrendo a comunicação baseada nos nós ativos após a execução do algoritmo de *Bully*. Neste último caso variamos também o tempo de propagação de mensagens dos nós ativos para a componente centralizada.

A Figura 4(a) apresenta a média do tráfego acumulado por segundo no servidor. Os resultados mostram que a solução de disseminação baseada em inundação leva o sistema a ficar facilmente sobrecarregada, tal não acontece com o mecanismo de disseminação alternativo, que, tal como esperado, mostram uma menor carga imposta sobre o servidor conforme o tempo de propagação de atualizações para o servidor aumenta. O leitor deve notar que estes resultados mostram o acumulado obtido na totalidade da experiência que inclui a transmissão de dados necessária para descarregar o código HTML e JavaScript do servidor assim como o estabelecimento de ligações entre os clientes. Visto que a duração da experiência é relativamente baixa, estes custos seriam amortizados em execuções mais longas.

Na Figura 4(b) apresentamos o tráfego médio por segundo em cada cliente. É notório que este tráfego varia com o número de clientes, ou seja, quando o número

de utilizadores no sistema aumenta, aumenta também a quantidade de tráfego gerado. Isto contece devido a um aumento do número de ligações a fazer entre os clientes e a interação directa entre os mesmos.

5.3 Suporte a desconexão e outros ambientes de execução

Suporte a desconexão: Efetuamos experiências onde desligámos o acesso à componente centralizada durante a execução das experiências onde os vários clientes efetuam escritas, ou seja, desconexão física à rede exterior (Internet). Esta experiência revelou que os clientes que tinham ligações diretas entre si continuavam a cooperar ativamente. Estes resultados (cujos detalhes omitimos devido a restrições de espaço) validam este aspeto do desenho da nossa framework.

Outros dispositivos: Verificámos a capacidade de usar a nossa framework num conjunto variado de dispositivos, que incluíram portáteis, telemóveis, e tablets ligados através de uma rede local. Verificámos que estes dispositivos conseguiam criar ligações entre si, permitindo a sua comunicação sem necessidade de contactar a componente centralizada.

6 Trabalho Relacionado

Serviços colaborativos baseados em Web tipicamente armazenam os dados dos clientes em servidores, os quais podem ser replicados geograficamente para garantir uma mais baixa latência e elevada disponibilidade. O uso de consistência forte para este tipo de aplicações tende a não ser adequado devido a impor restrições ou atrasos à execução de operações pelos utilizadores. Assim, estes sistemas usam tipicamente soluções de consistência eventual (ou causal), as quais incluem técnicas de resolução de conflitos para unificar as modificações executadas concorrentemente quando estes (inevitavelmente) aparecem.

Vários sistemas suportam a edição colaborativa entre clientes em tempo real recorrendo a solução de transformação de operações (OT) [13]. Nestas soluções, as operações são transformadas antes de serem executadas na réplica de cada cliente. Apesar de terem sido propostos algoritmos em que esta transformação ocorre de forma distribuída em cada cliente, os sistemas normalmente usam algoritmos que recorrem a uma componente centralizada que efetua a transformação das operações. O Etherpad [3] e o ShareJS [5] são sistemas de processamento de texto baseado na Web que usam servidores próprios que transformam as operações dos clientes. A Google Realtime API[6] requer o uso dos servidores e autenticação da própria Google, mas tem suporte a mais tipos de dados como Listas e Mapas.

As soluções de OT foram extensivamente estudadas na literatura, especialmente pela comunidade de edição colaborativa, e muitos algoritmos de OT foram apresentados. Contudo foi demonstrado que a maioria dos algoritmos de OT para sistemas descentralizados são incorretos [10]. Os CRDTs, Convergent (ou Commutative) Replicated Data Types, surgiram inicialmente no contexto da edição colaborativa [14], como uma aproximação alternativa ao OT que permite uma solução de sincronização entre pares ao mesmo tempo simples e correta. Os CRDTs [16] são tipos de dados replicados que

garantem a convergência por desenho, permitindo a execução imediata das operações num cliente sem necessidade de coordenação.

Os CRDTs têm encontrado sucesso em várias aplicações distribuídas, por exemplo, a base de dados NoSQL Riak[11] é um exemplo de uma key-value store distribuída com alta disponibilidade que recorre internamente a CRDTs. O SwiftCloud [15] é outro sistema de armazenamento de dados eventualmente consistente que usa CRDTs para manter caches do lado do cliente. O nosso sistema difere destes sistemas ao permitir aos clientes sincronizarem diretamente as réplicas dos objetos partilhados entre os clientes.

A solução que propomos neste artigo recorre à comunicação direta entre os clientes, seguindo um modelo entre-pares (do inglês *peer-to-peer*). Os sistemas entre-pares foram já amplamente estudados, sendo que existem soluções propostas na literatura para a gestão descentralizada de redes sobrepostas [12,17,4] e também para a disseminação eficiente e tolerante a faltas de informação [12,1]. Neste trabalho inspiramo-nos em algumas destas soluções para construir o subsistema de comunicação, em particular o módulo de gestão de redes sobreposta.

Um trabalho recente, Priv.io [19] recorre ao uso de WebRTC para suportar uma rede social com aspetos descentralizados. Este sistema pretende resolver problemas de privacidade nas redes sociais atuais, garantindo que toda a componente centralizada destes sistemas seja garantida por um conjunto de recursos federados (na nuvem) controlados (e pagos) pelos utilizadores. Para permitir a interação entre os utilizadores, o Priv.io recorre a uma aplicação JavaScript suportada por WebRTC para permitir aos clientes a troca direta de chaves criptográficas, entre outras interações em tempo real (e.g, conversação). No entanto, o foco principal deste trabalho passa pela privacidade dos dados dos utilizadores, evitando o uso de uma componente centralizada que serve de repositório de dados.

7 Conclusões e Trabalho Futuro

Neste artigo apresentamos uma arquitetura alternativa para o suporte de aplicações Web colaborativas com comunicação direta e transparente browser-a-browser. Implementámos esta arquitetura sob a forma de uma framework que recorre a WebRTC para materializar a comunicação direta entre browsers, e recorre a CRDTs para materializar réplicas locais nos clientes, que podem ser propagas diretamente entre estes, atualizadas localmente e reconciliadas entre os vários clientes, para além de serem propagadas para uma componente centralizada para garantir persistência e compatibilidade com clientes que não suportem WebRTC. Com base nesta framework implementámos um conjunto de aplicações simples, e uma aplicação de edição colaborativa que comparámos experimentalmente com uma aplicação similar sobre a Google Drive Realtime API.

Como trabalho futuro pretendemos melhorar o desenho de várias das componentes da nossa framework, em particular o desenho da componente de rede sobreposta e de comunicação. Pretendemos também estudar no contexto deste sistema o *trade-off* existente entre várias alternativas de desenho de CRDTs, nomeadamente entre as alternativas baseadas em operações e estado.

Agradecimentos: Este trabalho foi parcialmente suportado pelos projectos FCT/MEC NOVA LINCS PEst UID/CEC/04516/2013 e pelo projecto Europeu FP7 SyncFree (grant agreement 609551).

Referências

1. Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Transactions on Computer Systems (TOCS) 17(2), 41–88 (1999)
2. Coulouris, G.F., Dollimore, J., Kindberg, T.: Distributed systems: concepts and design. pearson education (2005)
3. EtherpadFoundation: Etherpad, http://etherpad.org
4. Ganesh, A.J., Kermarrec, A.M., Massoulié, L.: Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In: Networked Group Communication, pp. 44–55. Springer (2001)
5. Gentle, J.: Sharejs api, https://github.com/share/ShareJS#client-api
6. Google: Realtime api, https://developers.google.com/drive/realtime
7. IETF: Stun, https://tools.ietf.org/html/rfc5389
8. IETF: Turn, https://tools.ietf.org/html/rfc5928
9. IETF: Websocket, https://www.websocket.org
10. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. Theor. Comput. Sci. 351(2), 167–183 (Feb 2006), http://dx.doi.org/10.1016/j.tcs.2005.09.066
11. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming. p. 14. ACM (2010)
12. Leitao, J., Pereira, J., Rodrigues, L.: Hyparview: A membership protocol for reliable gossip-based broadcast. In: Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on. pp. 419–429. IEEE (2007)
13. Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology. pp. 111–120. UIST '95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/215585.215706
14. Preguica, N., Marques, J.M., Shapiro, M., Letia, M.: A commutative replicated data type for cooperative editing. In: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems. pp. 395–403. ICDCS '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ICDCS.2009.20
15. Preguiça, N., Zawirski, M., Bieniusa, A., Duarte, S., Balegas, V., Baquero, C., Shapiro, M.: Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In: Reliable Distributed Systems Workshops (SRDSW), 2014 IEEE 33rd International Symposium on. pp. 30–33. IEEE (2014)
16. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. pp. 386–400. SSS'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2050613.2050642
17. Voulgaris, S., Gavidia, D., Van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. Journal of Network and Systems Management 13(2), 197–217 (2005)
18. W3Cs: Webrtc, http://w3c.github.io/webrtc-pc/
19. Zhang, L., Mislove, A.: Building confederated web-based services with priv.io. In: Proceedings of the First ACM Conference on Online Social Networks. pp. 189–200. COSN '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2512938.2512943