UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ESTÁCIO PEREIRA DA SILVA NETO

E SE APLICAÇÕES SOUBESSEM COMO SIMULAR RESPOSTAS QUANDO NÃO HÁCONEXÃO COM O SERVIDOR?

CAMPINA GRANDE - PB

2019

ESTÁCIO PEREIRA DA SILVA NETO

E SE APLICAÇÕES SOUBESSEM COMO SIMULAR RESPOSTAS QUANDO NÃO HÁCONEXÃO COM O SERVIDOR?

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Matheus Gaudencio do Rêgo.

CAMPINA GRANDE - PB

2019

S586e Silva Neto, Estácio Pereira da.
E se aplicações soubessem como simular respostas quando não há conexão com o servidor? / Estácio Pereira da Silva Neto. – 2019.

11 f.

Orientador: Prof. Dr. Matheus Gaudencio do Rêgo
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Aplicações web. 2. Cache. 3. Compartilhamento de código. 4. Service worker. 5. Internet. 6. Módulo PWA. 7. Progressive Web Apps. I. Rêgo, Matheus Gaudencio do. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

ESTÁCIO PEREIRA DA SILVA NETO

E SE APLICAÇÕES SOUBESSEM COMO SIMULAR RESPOSTAS QUANDO NÃO HÁCONEXÃO COM O SERVIDOR?

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Matheus Gaudencio do Rêgo
Orientador – UASC/CEEI/UFCG

Professor Dr. Thiago Emmanuel Pereira da Cunha Silva
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Examinador – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro 2019.

CAMPINA GRANDE - PB

E se aplicações soubessem como simular respostas quando não há conexão com o servidor?

Trabalho de Conclusão de Curso

Estácio Pereira da Silva Neto
Universidade Federal de Campina Grande
estacio.pereira@gmail.com

Matheus Gaudencio do Rêgo
Universidade Federal de Campina Grande
matheusgr@computacao.ufcg.edu.br

RESUMO[1]

A internet está se popularizando e o mundo está cada vez mais conectado. Aplicações web, portanto, com seu modelo cliente-servidor, fazem parte do dia-a-dia de grande parte dos usuários da internet. Porém, ao perder conexão com o servidor, o usuário fica sem acesso à aplicação. Para resolver isto, a indústria evoluiu e lida com boa parte dos problemas de conexão utilizando cache, porém o usuário fica limitado à leitura de dados previamente acessados. Propomos, portanto, que o cliente saiba como lidar com determinadas requisições quando não houver conexão com o servidor e, para isto, o servidor deve compartilhar o código necessário para respondê-las. O cliente executa este código no *service worker* para não impactar no processo de renderização da interface. Avaliamos a performance da solução, em tempo de resposta, e os dados obtidos revelaram que para requisições abaixo de 5 *kilobytes* o servidor responde mais rapidamente que o *worker*, porém a diferença é imperceptível para usuários. Ademais, acima de 20 *kilobytes* o *service worker* prova-se superior. Dito isto, concluímos que a solução é viável no contexto de aplicações web, impacta positivamente no desempenho das mesmas e traz uma melhora significativa na experiência do usuário.

Palavras-chave

software, internet, web, cache, *service worker*, compartilhamento de código.

1. INTRODUÇÃO

Aplicações web evoluíram consideravelmente ao longo dos anos e grandes empresas nasceram por meio da internet. Empresas como Google e Facebook começaram com sites que mudaram a perspectiva das pessoas sobre o que era possível se fazer com o navegador em um computador conectado à rede mundial de computadores. Desde então, várias empresas têm consciência da importância de levar a internet para o máximo de pessoas possível e de melhorar cada vez mais a experiência de antigos e novos usuários da web. A Google tem tido um papel fundamental nesta transformação ao investir em projetos que objetivam empoderar os desenvolvedores e guiá-los no novo estilo de desenvolvimento web. Uma das iniciativas da dona do buscador mais popular do mundo neste sentido é o suporte cada vez mais presente a *Progressive Web Apps* (PWAs). PWAs são aplicativos web que entregam uma experiência mais parecida possível com a de aplicativos nativos.

Segundo o site principal da Google sobre PWA [1], aplicações web devem ser melhores no sentido de serem rápidas, confiáveis e cativantes. Devem ser rápidas, pois cada vez mais surgem estudos sobre a evasão de usuários em sites que demoram demais a carregar algum recurso [2]. Performance é definitivamente importante em uma aplicação. Devem ser cativantes ao oferecer ferramentas que empoderem os usuários e que os faça sentir que estão utilizando um aplicativo e não apenas acessando um site. As APIs fornecidas pelo navegador possuem um papel fundamental neste sentido, por permitir que aplicações web acessem recursos do sistema operacional, como aplicativos nativos fazem. Por fim, devem ser confiáveis não apenas no quesito de segurança, mas também devem, por exemplo, transmitir ao usuário a confiança de que mesmo que suas alterações feitas não sejam salvas, ele pode fechar o navegador e seu progresso não será perdido.

Ainda falando de confiabilidade de aplicações web, um ponto crucial na experiência dos usuários é quando há a perda de conexão do cliente com o servidor. Por estarmos tratando de um mundo cada vez mais mobile [3], tornou-se frequente o uso da internet em movimento, que pode acarretar em momentos de perda de conexão. É necessário, portanto, que os *websites* saibam como lidar com este cenário. A Google propõe várias soluções para problemas derivados da falta de conexão, inclusive fornece artigos e até uma biblioteca JavaScript, o *Workbox* [4], para auxiliar os desenvolvedores a abordar este problema da melhor forma. Porém, as soluções geralmente envolvem utilização do cache do navegador para as operações *offline*, ou prover páginas de erro mais amigáveis para mostrar que não há conexão e o usuário não pode acessar determinado recurso. Ambas são abordagens válidas, porém limitam a experiência do usuário. O cache, por exemplo, só irá conseguir responder a dados que já foram consultados anteriormente. Assim como páginas de erro

[1] Os autores retêm os direitos, ao abrigo de uma licença Creative Commons Atribuição CC BY, sobre todo o conteúdo deste artigo (incluindo todos os elementos que possam conter, tais como figuras, desenhos, tabelas), bem como sobre todos os materiais produzidos pelos autores que estejam relacionados ao trabalho relatado e que estejam referenciados no artigo (tais como códigos fonte e bases de dados). Essa licença permite que outros distribuam, adaptem e evoluam seu trabalho, mesmo comercialmente, desde que os autores sejam creditados pela criação original."

podem frustrar o usuário que, se continuar sem conexão, não pode fazer nada na aplicação.

Neste trabalho, propomos utilizar o navegador ao nosso favor para solucionar o problema do prejuízo à experiência do usuário no cenário supracitado. O *service worker* [5], que se comporta como a camada entre a aplicação web e a rede, pode ser utilizado com uma inteligência maior do que apenas responder, com cache, às falhas ao requisitar o servidor. Ao compartilhar código do servidor com o *service worker*, é possível fazer com que este último responda, ao detectar a falta de conexão, às requisições da mesma forma que seria feita no próprio servidor, já que o código é completamente ou parcialmente o mesmo. No caso do servidor não querer expor todo o algoritmo executado em um recurso, pode-se compartilhar parte dele, que será utilizado para satisfazer a necessidade imediata do usuário em obter aquela resposta. Desta forma, além de não haver duplicação de código, há uma melhora significativa na experiência de uso da aplicação. Porém, é necessário avaliar a viabilidade desta solução. Responder ao usuário nestas condições é um ganho que pode vir com problemas de performance ao executar as funções na máquina do usuário.

Para isto, decidimos avaliar a viabilidade da solução no contexto de aplicações web a fim de decidirmos se vale a pena utilizar a abordagem proposta para tolerância a falhas de conexão. Construímos uma aplicação[2] de exemplo que implementa estes conceitos com o máximo de compartilhamento de código possível e que simula cenários de computação realizada pelo servidor no que se refere a tamanho de dados retornados e complexidade do algoritmo executado. Analisamos o tempo de resposta das requisições realizadas nos diferentes estados de conexão e desta forma conseguimos verificar se há um ganho ou se há perda de performance a depender de quem executa a operação (servidor ou cliente).

Os resultados foram satisfatórios e confirmaram nossas suspeitas de que a computação realizada no lado do cliente não afetaria a performance do site de forma negativa. Por estar sendo realizada numa thread diferente da de renderização da *DOM* [6], não há impacto na visualização do site. Além disso o *service worker* mostrou-se ser bastante performático e, em algumas ocasiões, mais rápido do que o servidor em tempo de resposta.

Na Seção 2 descrevemos a arquitetura proposta. Em seguida, a metodologia do experimento é apresentada para, na Seção 4, demonstrarmos e discutirmos os resultados obtidos. Concluímos o artigo com uma discussão mais abrangente sobre as limitações do trabalho e possíveis trabalhos futuros.

2. ARQUITETURA

Por tratar-se de uma aplicação web, os dois principais módulos são, i) o servidor, em nosso exemplo escrito em Node.js, e, ii) o cliente que, neste caso, foi feito em *React.js* simplesmente pelo ambiente de desenvolvimento ser mais confortável nos momentos de alteração de código, por utilizar o *Hot Module Replacement*. Neste projeto, desenvolvemos uma biblioteca adicional que opera junto ao cliente contendo todo código necessário para que a aplicação, de forma transparente à *UI*[3], responda a requisições mesmo quando *offline*. A separação feita faz com que o código do cliente tenha sua área de atuação bem definida e, desta forma, possa ser implementado independentemente. A Figura 1 mostra essa arquitetura, onde o desenvolvedor de uma aplicação web desenvolverá seu cliente de forma genérica e irá incorporar esta biblioteca desenvolvida.

Há uma semelhança notável entre os módulos principais da arquitetura web e esta se dá por ambos possuírem bancos de dados. Enquanto o banco de dados do servidor pode ser escolhido pelo projetista de software, o do navegador não pode. Todos os principais navegadores do mercado implementam uma mesma API para armazenamento e busca em alta performance de quantidades significantes de dados chamada *IndexedDB* [7]. Ou seja, é possível armazenar vários dados *client-side* de forma elegante e performática. Além disso, há também a presença de duas entidades responsáveis por gerenciar bancos de dados. Uma para o cliente e outra para o servidor. Desta forma, o código dos serviços pode ser implementado fazendo uso de uma interface genérica de consulta e escrita de dados, fazendo-o ser independente de contexto. Esta interface genérica deve prover, no caso deste trabalho, as operações básicas de consulta: criação, atualização e remoção de dados. Ao executar o código de algum serviço, a instância do gerenciador de dados correspondente deve ser disponibilizada.

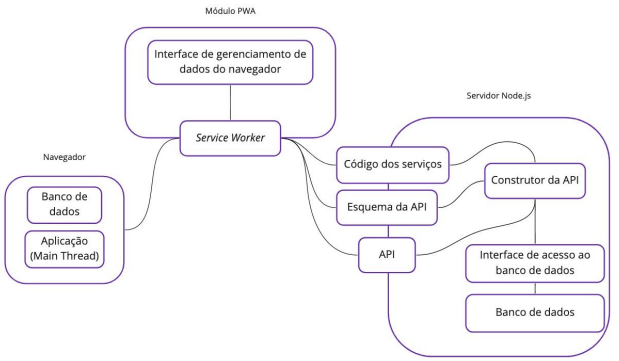

Figura 1: Arquitetura da solução *Web*

2.1 Cliente

No lado do cliente, o módulo mais simples, temos apenas o banco de dados nativo do navegador e a aplicação *React.js* sendo executada na *Main Thread*. Isso se deve ao fato de termos decidido transferir toda a lógica de manipulação de requisições para o *service worker* de modo que tudo fique transparente para a interface que interage com a aplicação cliente. Além disso, o código da solução encontrada torna-se totalmente independente de bibliotecas e *frameworks* web como *React, Angular e Vue.js*.

2.2 Módulo PWA

As bibliotecas e *frameworks* mencionados na seção anterior lidam com manipulações da *DOM*, ou seja, da *UI.* Para isso, o código JavaScript executado por estes tipos de tecnologias executa na *Main Thread*, mencionada anteriormente. Além de executar código JavaScript, a *Main Thread* realiza todas as operações relacionadas à visualização do site, portanto, cuida da organização dos elementos na tela, da estilização dos mesmos, de capturar eventos como cliques e digitação, etc. Portanto, com todo este trabalho sendo executado em uma única *thread*, era necessário uma solução para evitar que código JavaScript relacionado a lógica de negócio bloqueie as tarefas de *layout* e,

[2] https://github.com/estacioneto/TCC
[3] User Interface

assim, o site congele. Surgiram então a *Worker Thread* e os *Web Workers* [8].

A *Worker Thread* se trata de um contexto diferente da *Main Thread* que consegue executar código em paralelo sem afetar a execução de tarefas de interface gráfica. *Web Workers* são mecanismos que possibilitam tal execução e um *worker* está associado a um código JavaScript. Além disso, *workers* não podem acessar determinadas APIs do navegador diretamente, como a *DOM*, porém conseguem comunicar-se com a *Main Thread* a partir da troca de mensagens e assim conectar o processamento de dados à interface gráfica. Os *service workers*, como mencionados anteriormente, são de um tipo especial de *worker* que deve lidar especificamente com tarefas relacionadas à conexão de rede e por isso implementamos um *service worker* para responder às requisições da aplicação no *client-side*.

Criamos, portanto, um módulo responsável pelo código relacionado à solução em si, ou seja, lidar com a falta de conexão com a API. Esse módulo conta com as implementações do *service worker* e da instância para gerenciar os dados do *IndexedDB*. Para a implementação deste gerenciador, foi utilizada a biblioteca *LocalForage*[4] que facilita bastante a manipulação de dados presentes no banco de dados do navegador.

2.2.1 *Service worker*

Para servir o *service worker* corretamente, resolvendo todas as dependências importadas pelo arquivo principal, utilizamos o *Rollup* [9] e para auxiliar com as requisições realizadas pela aplicação utilizamos o *Workbox*, mencionado na introdução deste documento. O *Workbox*, em conjunto com *TypeScript* [10], faz com que o desenvolvimento de um *service worker* seja bem mais rápido, pois a biblioteca resolve muita lógica de negócio relacionada a cache, bem como fornece outras ferramentas para funcionalidades mais complexas como *push notifications* e sincronização em segundo plano.

2.2.2 *Funcionamento*

O *service worker*, ao ser ativado, primeiramente requisita ao servidor os dados que podem ser manipulados no cliente para que seja possível popular o banco de dados do navegador. Além disso, ele requisita à API um objeto de configuração que define quais recursos o cliente pode resolver por si mesmo, bem como qual serviço está associado à rota e qual função deste serviço deve ser executada para responder a requisição. Neste objeto de configuração também está presente a rota do servidor na qual o código dos serviços pode ser baixado.

```
{
  "servicesUrl": "/services_code/",
  "routes": {
    "/collections": {
      "GET": {
        "service": "collection_service",
        "handler": "getCollections",
      },
      "POST": {
        "service": "collection_service",
        "handler": "addCollection",
      }
    }
  }
}
```

Código 1: Exemplo de Configuração de Recursos

A partir do objeto *JSON*[5] no Código 1, por exemplo, é possível ter uma ideia de onde requisitar o código dos serviços, quais requisições responder e como respondê-las. Neste caso, o

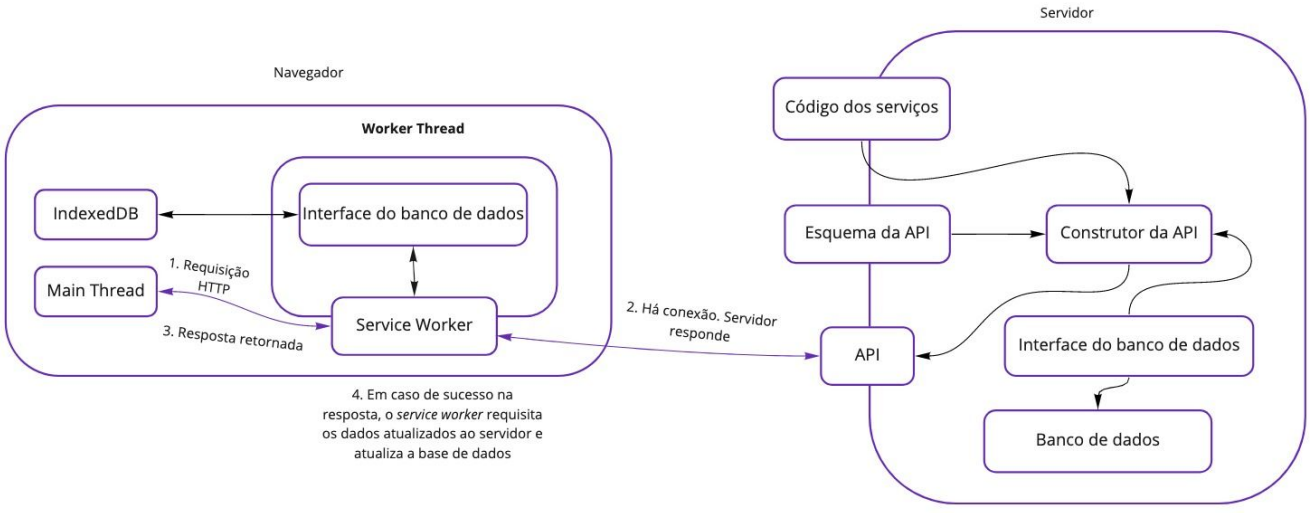

Figura 2: Fluxo de execução com conexão entre cliente e servidor

[4] https://github.com/localForage/localForage

[5] *JavaScript Object Notation* - Notação de Objetos JavaScript

serviço *collection_service* está disponível no recurso */services_code/collection_service.js* e responde a requisições do tipo GET e POST ao recurso */collections* a partir das funções *getCollections* e *addColection* respectivamente.

De posse dos recursos que devem ser respondidos em caso de falha de conexão com o servidor, o *service worker* registra um *listener* para o evento de *fetch*, que é disparado pelo navegador quando há uma requisição. Caso a requisição não seja para nenhuma das rotas definidas ou divirja no método HTTP, não há envolvimento do *worker*. Caso contrário, será feita a tentativa de requisição para o recurso da API e, em caso de sucesso, a resposta é retornada para a aplicação enquanto o *worker* requisita o servidor para atualizar sua base de dados que pode ter ficado inconsistente após a operação. Porém, se ocorrer uma falha na conexão com o servidor, o *worker* executará a função do serviço correspondente. Na chamada da função, o *worker* informa os parâmetros da requisição e também a instância do gerenciador dos dados do *IndexedDB* e desta forma o serviço consegue executar normalmente. O dado retornado pela função é redirecionado para a aplicação, porém é necessário realizar a sincronização do servidor ao detectar o retorno da conexão.

Para isto o *Workbox* provê ao desenvolvedor uma fila especial que lida com sincronização em segundo plano. Ou seja, requisições que não puderam ser completadas podem ser enfileiradas e, quando possível, enviadas novamente para sincronizar o cliente com o servidor. Utilizamos, portanto, esta estrutura de dados presente na biblioteca, porém implementamos nosso próprio método de sincronização, pois é necessário atualizar a base de dados do cliente no final do processo. Ao enviarmos ao servidor as mesmas requisições que já foram executadas no cliente, e não apenas os dados processados, a execução realizada no servidor servirá não só para atualizar o banco de dados, mas também para validar as operações processadas no cliente, evitando possíveis problemas de segurança. O resultado dessas requisições não precisa ser enviado à aplicação executando na *Main Thread*, pois o dado que o usuário requisitou já foi respondido outrora pelo *service worker*.

2.3 Servidor

O servidor é o módulo que possui mais componentes e faz com que o Módulo PWA saiba como reagir. Afinal, se alguns recursos não conseguem ser respondidos *client-side*, o *service worker* não precisa saber deles. Começando pelo banco de dados, foi utilizado o sistema de arquivos para guardar os dados com a finalidade de simplificar a implementação. Além disso, a instância que gerencia os dados foi construída com a ajuda da biblioteca *LowDB*[6], que abstrai a manipulação do arquivo de banco de dados.

Para construir a API REST, utilizamos a biblioteca *Express* [11] novamente em conjunto com *TypeScript*. Para o contexto deste trabalho, o servidor apenas realiza as operações presentes no esquema, ou seja, tudo que a API pode fazer, o *service worker* consegue fazer. Portanto, para construir os recursos presentes no esquema, foi implementado um construtor que consome os dados do arquivo de configuração e, via *Express*, define os recursos.

Como os serviços estão diretamente ligados à API, estes estão no mesmo módulo do servidor *Node.js*. Porém, para serem servidos de forma que sejam executados em diferentes contextos, os serviços precisam de alguém que resolva as dependências importadas por eles e isole-los em arquivos separados. Para isso foi utilizado novamente o *Rollup* que, com pouca configuração, consegue gerar arquivos JavaScript que podem ser executados tanto no servidor quanto no navegador. Com isso, após a transformação destes arquivos, é possível servir a API, o esquema dela e os recursos utilizados por ela.

2.4 Fluxos de execução

A Figura 2 mostra o fluxo de execução quando há conexão com o servidor. Nota-se que não é muito diferente de uma requisição comum. A única diferença é a necessidade de uma requisição extra para atualizar os dados do cliente. Neste sentido, podem ser feitas diversas otimizações para reduzir o custo no que se refere a consumo de banda do usuário. Já na Figura 3, o cenário retratado é mais complexo. É o cenário em que há uma requisição que falha por falta de conexão com o servidor. Neste caso, é necessário que o *service worker* responda à requisição e eventualmente sincronize os dados com o servidor. Como foi explicado anteriormente, ao fim da sincronização, o cliente precisa atualizar a base de dados presente no navegador.

3. METODOLOGIA

Primeiramente, como o ambiente alvo é o de aplicações web, foram desenvolvidas a base do serviço que disponibiliza o código compartilhável e as instâncias dos gerenciadores das bases de dados do navegador e do servidor. Logo após, a API foi desenvolvida juntamente com o código necessário para que o código compartilhável pudesse ser utilizado para lidar com as requisições ao servidor. Por fim, com o código do servidor pronto, foi feito o código do service worker para lidar com as requisições no caso de falhas ao requisitar o servidor, utilizando o esquema da API para saber como lidar e reutilizando o código compartilhável para conseguir responder à *Main Thread*, bem como foi desenvolvido o código da aplicação web em si. Com tais etapas finalizadas, foram feitos testes dos gerenciadores das bases de dados para garantir a corretude do código desta camada, que é a mais dependente de contexto.

Finalizada a etapa de desenvolvimento, iniciamos a coleta de métricas de performance. Para esta etapa foi decidida a forma de avaliação e esta baseia-se no tempo de resposta, que, no nosso experimento, será diretamente proporcional ao tamanho do dado a ser retornado pela função chamada pela API ou pelo *service worker*.

3.1 Experimento

Decidido isto, o primeiro passo foi a construção do cenário dos experimentos. Foi implementada, no serviço de código compartilhável, uma função que simulará a execução de diversos cenários possíveis numa aplicação web. Esta consiste tanto na geração de uma palavra aleatória de tamanho *length*, fornecido como parâmetro na url requisitada, como no incremento de um contador de gerações feitas. Ou seja, ao requisitar o recurso */api/generate_string/:id/:length*, a função irá buscar na base de dados o número de gerações feitas até o momento, gerar uma palavra de tamanho igual ao parâmetro *length*, incrementar o contador de gerações, atualizar o banco de dados utilizando o parâmetro *id* para indexação e, finalmente, retornar o dado atualizado. O dado retornado contém tanto a palavra gerada, como o valor do contador. A geração da palavra é feita em um laço de repetição que gera caracteres de 'a' até 'z' e concatena-os, a uma

[6] https://github.com/typicode/lowdb

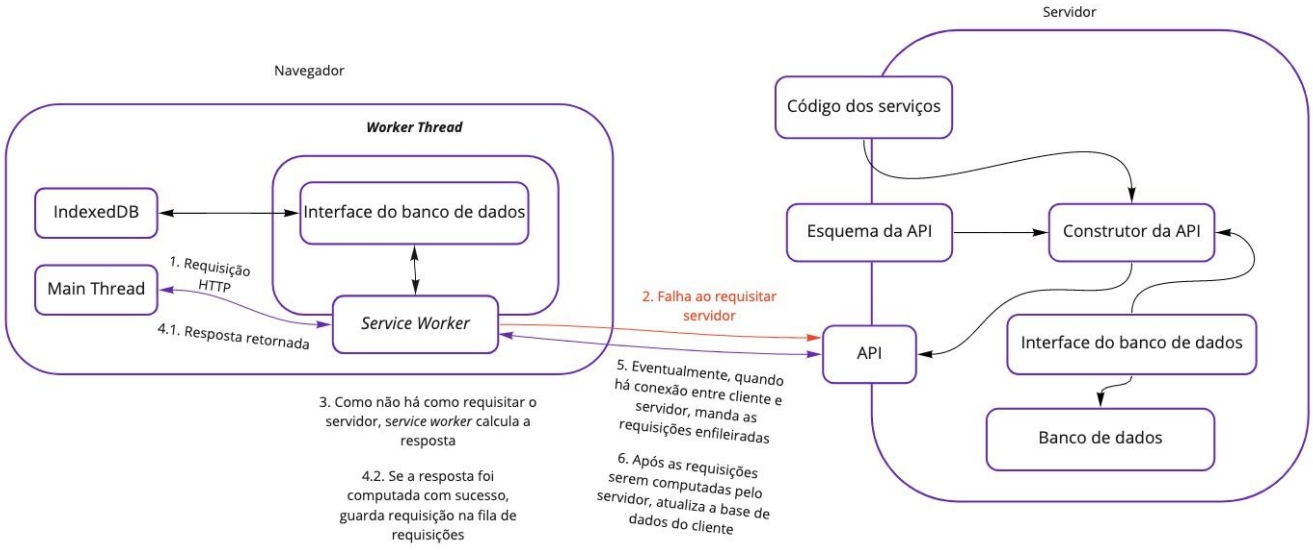

Figura 3: Fluxo de execução com falha de conexão entre cliente e servidor

miro

palavra inicialmente vazia. Portanto, a complexidade da função que lida com o recurso acima é de O(*length*^{2}), assumindo que o motor JavaScript utilizado realiza a concatenação de strings em tempo linear e a geração de um número aleatório em tempo constante.

Após a elaboração do cenário a ser avaliado, utilizamos o navegador, neste caso o Google Chrome por possuir o mesmo motor JavaScript do Node.js, para a coleta dos dados relativos à performance. A aplicação web possui um botão que dispara 100 requisições sequenciais para um recurso, com meio segundo entre elas. Além disso, pelo fato de o cliente e o servidor estarem sendo executados na mesma máquina, a latência da rede é zero. As requisições sendo feitas sequencialmente não impacta na avaliação já que nos dois ambientes o código JavaScript roda sequencialmente e a função é, por natureza, *CPU Bound*. Portanto, ao modificar o parâmetro *length* conseguimos simular desde uma computação mais simples, gerando uma palavra de tamanho quinhentos por exemplo, até uma computação bem mais intensa, gerando uma palavra de tamanho um milhão. Ademais, o tamanho da palavra retornada é bem aproximado do valor, em *bytes*, do dado retornado pela função, o que facilita a medição.

Desta forma, foram coletados dados de cem requisições de três tipos:

- API: Trata-se de requisições que passaram pelo *service worker*, mas como havia conexão com o servidor, foi possível enviar a computação para ser realizada no servidor.
- Worker: É caracterizado pela falta de conexão com o servidor, ou seja, a requisição deve ser respondida e, consequentemente computada, pelo *service worker* e guardada numa fila de requisições para sincronizar cliente e servidor quando houver conexão entre eles.
- Sync: Refere-se justamente à sincronização do cliente com o servidor, uma vez que a conexão entre os dois foi restaurada.

Percebe-se então, que a diferença entre as requisições do primeiro e do terceiro tipos refere-se ao tempo de serialização do dado que está na *Worker Thread* para responder à *Main Thread*, que está presente nas requisições do primeiro tipo, porém não encontra-se presente nas do terceiro tipo. Além disso, para realizar a coleta das requisições do segundo e do terceiro tipos foi utilizado o painel de ferramentas do desenvolvedor do navegador, mais precisamente o menu que permite simular diferentes cenários de rede. Para realizar requisições do segundo tipo foi selecionada a opção *"Offline"* e ao término das requisições, ao selecionar a opção *"Online"*, o *service worker* automaticamente faz o trabalho de sincronizar as cem requisições que foram respondidas por ele, pois a API do *Workbox* lida com este cenário.

Como mencionado anteriormente, o tamanho da palavra gerada foi a nossa variável para medir a performance da aplicação web. Variamos, então, este valor para conseguirmos dados com no mínimo 500B e no máximo 1MB e com isso conseguir identificar se há diferença no tempo de resposta médio. Com os arquivos do tipo *HTTP Archive* (*.har*) extraídos do navegador, foi feito um script para separar os dados das requisições dos três tipos supracitados e colocá-los um arquivo *.csv* para, por fim, realizar a análise de resultados.

4. RESULTADOS

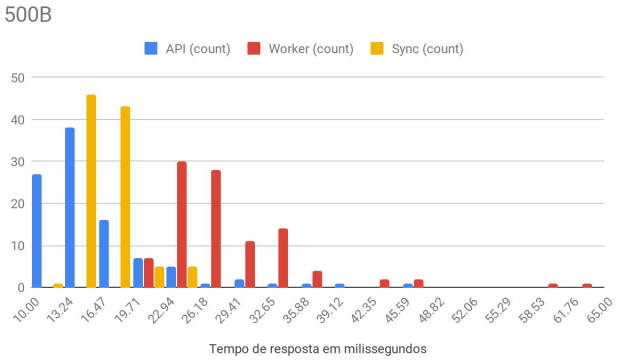

Figura 4: Gráfico de quantidade de requisições por tempo de resposta relativos a dados de tamanho 500 *bytes*.

A Figura 4 mostra o histograma de tempo de resposta no cenário em que foram geradas palavras de tamanho 500 e, consequentemente, o dado retornado possui 500 *bytes*. Como pode ser visto na figura, respostas do servidor são relativamente mais rápidas que respostas calculadas pelo *service worker* para um volume de dados relativamente pequeno. Há aproximadamente 15 milissegundos de diferença que, para o contexto de uma chamada HTTP, é imperceptível para o usuário. Nota-se, também que há uma disparidade de aproximadamente 10 milissegundos entre o tempo de resposta do servidor, com os dados trafegando do *service worker* para a *Main Thread* e o tempo de resposta das requisições de sincronização. Porém, as requisições do primeiro tipo mostram-se mais rápidas, contrariando as expectativas de que trafegar estes dados entre as duas *Threads* fosse causar um impacto negativo no tempo de resposta.

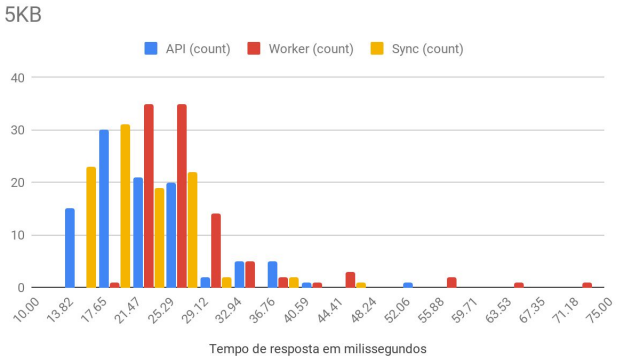

Figura 5: Gráfico de quantidade de requisições por tempo de resposta relativos a dados de tamanho 5 *kilobytes*.

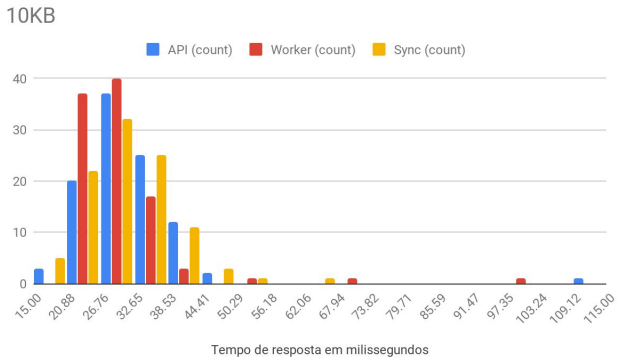

Figura 6: Gráfico de quantidade de requisições por tempo de resposta relativos a dados de tamanho 10 *kilobytes*.

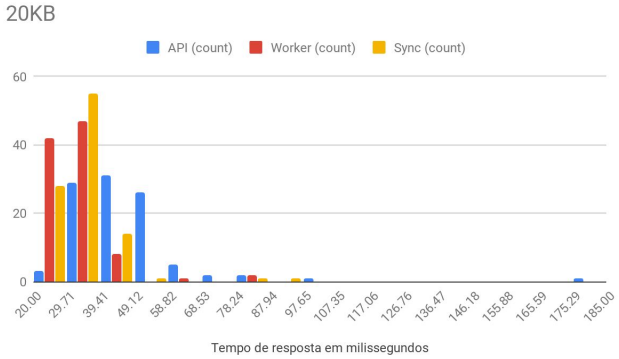

Figura 7: Gráfico de quantidade de requisições por tempo de resposta relativos a dados de tamanho 20 *kilobytes*.

Percebe-se comportamento similar, entre os tipos supracitados, nas três figuras seguintes, porém o tempo de resposta do service worker permanece na faixa dos 20 a 40 milissegundos, ao contrário do tempo de resposta do servidor, que passa de aproximadamente 14 a 40 milissegundos para 30 a 50 milissegundos. Ou seja, enquanto o tempo de resposta da API e da sincronização permanecem equivalentes, ao aumentar o volume de dados e, consequentemente, a quantidade de instruções a serem executadas pela função implementada no serviço compartilhado, a computação realizada no próprio navegador prova-se ser mais eficiente.

Para confirmar a crescente diferença, como mostra a Figura 8, realizamos um teste extremo ao tentar gerar uma palavra com um milhão de caracteres. Ao fazer isso, o dado gerado possui 1MB. O *service worker* consegue responder até quatro vezes mais rápido que o servidor, que deixaria o usuário esperando por aproximadamente um segundo até que a aplicação responda a um comando feito por ele.

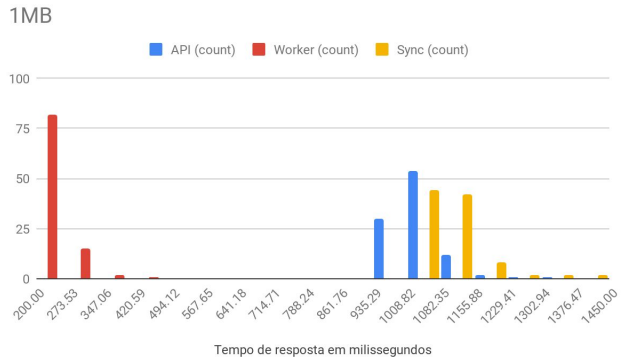

Figura 8: Gráfico de quantidade de requisições por tempo de resposta relativos a dados de tamanho 1 *megabyte*.

Por fim, fizemos uma análise de efeito para conseguir mensurar estatisticamente a diferença. Para calcular o Cohen d [12], a medida de efeito mais popular, utilizamos os dados de tempo médio de resposta, representados na Tabela 1, com 95% de confiança. Foi calculada, portanto, para os cenários apresentados nessa seção, conforme demonstrado na Tabela 2, e descobrimos que há um grande efeito negativo (≤ -1.0) do *service worker* na API para o cenário em que o volume de dados é pequeno, mostrado graficamente anteriormente. Conforme o tamanho dos dados vai aumentando, o efeito torna-se positivo e, a partir de 20 KB, fica acima de 1, ou seja, caracteriza-se um grande efeito. A diferença fica ainda mais significativa em dados de 1MB. Neste caso, o efeito é positivo e enorme tanto em relação às requisições feitas à API quanto às requisições feitas para sincronização de dados. Isto prova o potencial positivo do *worker* nestas situações no quesito performance.

Tabela 1: Média de tempo de resposta (em milisegundos).

Tamanho de dados	API	Worker	Sync
500B	16.94	29.34	17.08
5KB	23.83	29.17	21.97
10KB	32.26	29.81	31.67
20KB	47.4	32.77	34.03
1MB	1043.66	249.88	1106.2

Além disso, observa-se que o efeito dos eventos de sincronização em relação à API é baixo para dados de até 10KB. A partir daí, a divergência é grande, porém bem distante do efeito causado pelo *worker*.

Tabela 2: Cohen d entre as diferentes abordagens.

Tamanho de dados	Cohen d (Worker - API)	Cohen d (Worker - Sync)	Cohen d (Sync - API)
500B	-1.854	-2.315	-0.03
5KB	-0.683	-0.992	0.296
10KB	0.249	0.213	0.065
20KB	1.012	0.134	0.922
1MB	15.21	14.551	-0.936

5. CONCLUSÃO

Concluímos, portanto, que a solução proposta para o contexto de cliente e servidor, implementada da forma descrita neste documento, é viável, no ponto de vista de performance, tanto para pequenos quanto para grandes volumes de dados a serem computados. A diferença de performance observada nos casos em que há um pequeno volume de dados prova ter o mínimo de impacto negativo na aplicação, uma vez que o tempo de resposta não chega a cem milissegundos. Como não há latência na rede no experimento, em cenários de alta latência a estratégia do *service worker* pode ser ótima. Além disso, a solução traz benefícios na experiência do usuário que não podem ser alcançados com estratégias de cache convencionais.

Quanto a cenários com grande volume de dados e computações mais intensas, o *service worker* mostrou-se ser bem superior no tempo de resposta e, por estar executando fora da *Main Thread*, não impacta na renderização da interface. Portanto, pode-se até implementar uma estratégia na qual o *service worker* calcula a resposta mesmo quando há conexão com o servidor. Após responder à requisição, ou até mesmo ao iniciar a execução da função, o *worker* envia a requisição de sincronização para que o servidor saiba do que está acontecendo. Ademais, num cenário em que há vários clientes requisitando o mesmo servidor, dividir a computação entre eles pode ser bem mais eficiente do que sobrecarregar o servidor, prejudicando sua vazão.

Como nosso objetivo é provar um conceito, há possíveis melhorias a serem realizadas na solução implementada a fim de ser utilizada em projetos de uso real. Eficiência na recuperação e proteção dos dados que populam o banco de dados do navegador, tratamento de erros no cenário de sincronização, implementação de funcionalidades mais complexas como *middlewares*, etc. Tais pontos identificados possuem solução e só precisam de tempo para serem resolvidos. Com este trabalho, vimos que foi possível fazer uma aplicação web simples responder de forma consistente ao usuário mesmo sem conexão com o servidor. Isto foi alcançado com o mínimo de alterações no código do cliente e reutilizando código entre cliente e servidor, diminuindo a duplicação de código e aumentando a manutenabilidade de ambos.

Nós recomendamos, portanto, que os desenvolvedores de aplicações web de pequeno e médio porte analisem a adoção desta prática de reúso de código entre as duas pontas. Projetos *open source* e aplicações simples podem ser beneficiadas pela adoção desse padrão e podem influenciar outros desenvolvedores a buscarem soluções semelhantes nos diferentes âmbitos da engenharia de software, como, por exemplo, na comunicação entre microsserviços.

6. REFERÊNCIAS

[1] GOOGLE DEVELOPERS. Progressive Web Apps. Disponível em : <https://developers.google.com/web/progressive-web-apps> Acesso em: 10 nov. 2019.

[2] GOOGLE AI. User Preference and Search Engine Latency. Disponível em: <https://ai.google/research/pubs/pub34439/> Acesso em: 14 nov. 2019.

[3] GOOGLE MOBILE ADS BLOG. Mobile-friendly sites turn visitors into customers. Disponível em: <http://googlemobileads.blogspot.com/2012/09/mobile-friendly-sites-turn-visitors.html> Acesso em: 14 nov. 2019.

[4] GOOGLE DEVELOPERS. Workbox. Disponível em: <https://developers.google.com/web/tools/workbox> Acesso em: 11 out. 2019.

[5] MDN. Service Worker API - Web APIs. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API> Acesso em: 10 set. 2019.

[6] MDN. Document Object Model (DOM) - Web APIs. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model> Acesso em: 17 out. 2019.

[7] MDN. IndexedDB API - Web APIs. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API> Acesso em: 17 out. 2019.

[8] MDN. Web Workers API - Web APIs. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API> Acesso em: 17 out. 2019.

[9] ROLLUP. rollup.js. Disponível em: <https://rollupjs.org/guide/en/> Acesso em: 30 out. 2019.

[10] MICROSOFT. TypeScript - JavaScript that scales. Disponível em: <https://www.typescriptlang.org/> Acesso em: 11 nov. 2019.

[11] EXPRESS. Express - Node.js web application framework. Disponível em: <https://expressjs.com/> Acesso em: 11 nov. 2019.

[12] Lenhard, W. & Lenhard, A. (2016). Calculation of Effect Sizes. Disponível em: <https://www.psychometrica.de/effect_size.html>. Acesso em: 16 nov. 2019.