UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO SISTEMAS DE INFORMAÇÃO

Mateus Carlos de Souza

COMUNICAÇÃO EM TEMPO REAL PARA JOGO EDUCATIVO BEER GAME

Florianópolis
2023

Mateus Carlos de Souza

COMUNICAÇÃO EM TEMPO REAL PARA JOGO EDUCATIVO BEER GAME

Trabalho Conclusão de Curso submetido ao curso de Sistemas de Informação, Campus Florianópolis, da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Mario Antonio Ribeiro Dantas
Coorientador: Prof. Julibio David Ardigo

Florianópolis

2023

Mateus Carlos de Souza

COMUNICAÇÃO EM TEMPO REAL PARA JOGO EDUCATIVO BEER GAME

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Sistemas de Informação e aprovado em sua forma final pelo Curso Sistemas de Informação

Florianópolis, 04 de novembro de 2023.

_____
Prof. Álvaro Junio Pereira Franco, Dr.
Coordenador do Curso

Banca Examinadora:

_____
Prof.(a) Mario Antonio Ribeiro Dantas, Dr.(a)
Orientador(a)
Instituição UFSC

_____
Prof.(a) Julibio David Ardigo, Dr.(a)
Avaliador(a)
Instituição UDESC

_____
Prof.(a) Frank Augusto Siqueira, Dr.(a)
Avaliador(a)
Instituição UFSC

Este trabalho é dedicado aos meus pais.

RESUMO

Com a modernização dos meios de comunicação, cada vez mais as pessoas demandam que as informações sejam rápidas, que eventos aconteçam de forma atômica e em tempo real. Em um ambiente de sala de aula isso é relevante para manter a dinâmica do aprendizado, sobretudo quando envolve atividades interativas. Este trabalho objetivou analisar as características e ferramentas de um ambiente capaz de simular um jogo online com comunicação em tempo real, tomando como referência o jogo Beer Game, que simula uma cadeia de suprimentos e ilustra o Efeito Chicote. A partir da identificação das necessidades de um jogo como o proposto, foram elencadas e analisadas as principais abordagens de comunicação existentes atualmente, como o XHR Polling, o Long Polling, WebSockets e a solução Socket.IO. Com base nas características de cada uma das abordagens estudadas, foi proposta a execução de experimentos práticos utilizando cada forma de comunicação, com foco na simulação de uma aplicação de jogo educativo, controlando o ambiente, o número de clientes simultâneos e o tempo de execução. Os experimentos focaram no acompanhamento de três métricas: a vazão de dados, indicando o número de mensagens que podem ser enviadas; a latência, representando o tempo médio de atraso para que as atualizações cheguem ao cliente; e o uso de recursos, que reflete o poder computacional utilizado em cada cenário. Os experimentos indicaram que, para menos de 10 usuários simultâneos, a escolha da tecnologia teve pouco impacto. No entanto, com um aumento de usuários, tecnologias tradicionais como polling apresentam limitações na taxa de transmissão e latência, contrastando com a eficiência constante do WebSocket. Uma avaliação detalhada comparou o WebSocket e Socket.IO indicando que, apesar das diferenças de performance serem pequenas para jogos como o proposto, o Socket.IO traz outros benefícios como a otimização no uso de recursos computacionais, melhoria na segurança, desenvolvimento facilitado e recursos que se propõem melhorar a usabilidade de jogadores em cenários de quedas ou lentidão na conexão. Os resultados indicam que a tecnologia Socket.IO pode ser a mais indicada para a aplicação em questão. Sugere-se, para trabalhos futuros, o estudo de cenários com quedas de conexão utilizando as abordagens estudadas, validando os pontos de destaque do Socket.IO, visando otimizar a experiência do usuário.

Palavras-chave: Beer Game, Comunicação cliente-servidor, Polling, Long Polling, WebSockets, Socket.IO.

ABSTRACT

With the modernization of communication methods, people increasingly demand that information be rapid, events occur instantaneously, and everything happens in real-time. In a classroom setting, this is crucial for maintaining the dynamics of learning, especially when it involves interactive activities. This work aimed to analyze the characteristics and tools of an environment capable of simulating an online game with real-time communication, using the Beer Game as a reference, which simulates a supply chain and demonstrates the Bullwhip Effect. Upon identifying the needs of a game like the proposed one, the main existing communication approaches were listed and analyzed, such as XHR Polling, Long Polling, WebSockets, and the Socket.IO solution. Based on the characteristics of each studied approach, the implementation of practical experiments was proposed, using each communication method and focusing on the simulation of an educational game application, controlling the environment, the number of simultaneous clients, and the execution time. The experiments focused on tracking three metrics: data throughput, indicating the number of messages that can be sent; latency, representing the average delay time for updates to reach the client; and resource usage, reflecting the computational power used in each scenario. The experiments showed that for less than 10 simultaneous users, the choice of technology had little impact. However, with more users, traditional technologies like polling show limitations in transmission rate and latency, in contrast to the constant efficiency of WebSockets. A detailed evaluation compared WebSockets and Socket.IO, indicating that despite the small performance differences for games like the proposed one, Socket.IO offers additional benefits such as optimization in the use of computational resources, improved security, facilitated development, and features that aim to improve the usability for players in scenarios of connection drops or slow down. The results suggest that Socket.IO technology may be the most suitable for the application in question. For future work, it is suggested to study scenarios with connection drops using the studied approaches, validating the highlights of Socket.IO, aiming to optimize the user experience.

Keywords: Beer Game, Client-server communication, Polling, Long Polling, WebSockets, Socket.IO.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Nas diversas áreas de ensino, há sempre conceitos que podem ser mais bem compreendidos por meio de dinâmicas, o que faz delas uma estratégia simples e viável de consolidação da aprendizagem. Uma forma dinâmica de ensino são os jogos, que cada vez mais aparecem para apoiar o aprendizado em configurações educacionais tradicionais e não-tradicionais.

Um exemplo é o jogo da cerveja (Beer Game), desenvolvido no início da década de 1960, pelo Sloan's System Dynamics Group do Massachusetts Institute of Technology (MIT). O jogo simula o efeito chicote na cadeia de distribuição, que é uma distorção da percepção da procura ao longo da cadeia de abastecimento.

Para o fluxo natural de um jogo que tem por objetivo representar o dinamismo da cadeia de abastecimento do mercado, um ponto crucial é a execução em tempo real das jogadas, que faz com que qualquer ação executada no servidor seja replicada para outros jogadores imediatamente. Para que a comunicação ocorra de forma natural, é possível utilizar de ferramentas que permitam uma comunicação em tempo real entre Cliente/Servidor, um exemplo de ferramenta disponível no mercado é o WebSocket.

1.1 OBJETIVOS

Nas subseções abaixo estão descritos o objetivo geral e os objetivos específicos deste TCC.

1.1.1 Objetivo Geral

Este trabalho tem por objetivo realizar uma análise do jogo Beer Game da perspectiva de comunicação em tempo real entre cliente/servidor utilizando WebSockets, e propor uma solução para que o jogo seja tolerante a falhas do ponto de vista dos jogadores.

1.1.2 Objetivos Específicos

Para atingir o objetivo geral, os seguintes objetivos específicos deverão ser alcançados:

- Efetuar o levantamento de tecnologias existentes no ambiente e comunicação na web.
- Analisar o cenário de aplicações web utilizando como referência o jogo Beer Game do ponto de vista das tecnologias e protocolos de comunicação apresentados.
- Avaliar a aplicação do jogo através de simulações práticas, utilizando a ferramenta apresentada para comunicação e tolerância a falhas e comparando com outras abordagens atuais.

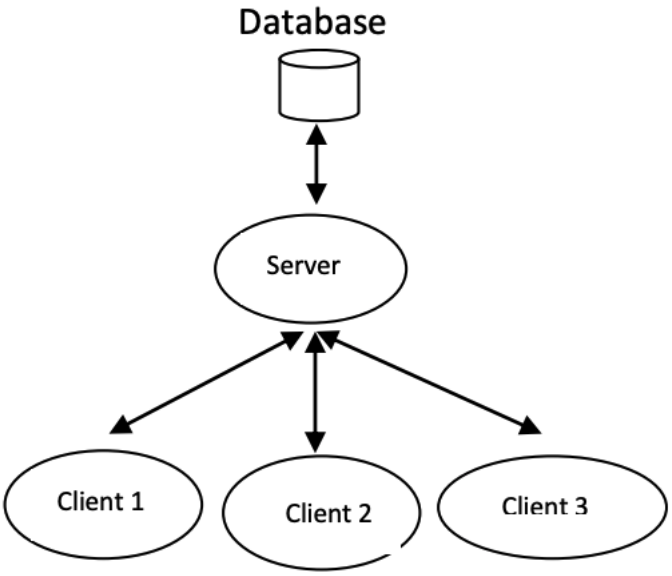
2 APLICAÇÕES WEB

Uma aplicação web é um programa de computador que usa um navegador da web para executar uma função específica. As aplicações web estão presentes em muitos sites. Um exemplo é um formulário de contato em um site. (Riverbed, 2017)

No modelo cliente-servidor, o servidor processa solicitações e armazena dados, como em um banco de dados para registros de usuário. Tecnologias como AJAX permitem uma experiência interativa, enquanto a arquitetura suporta múltiplos usuários simultâneos, assegurando escalabilidade e disponibilidade.

O sistema cliente-servidor pode ser definido como uma arquitetura de software composto pelo cliente e pelo servidor, em que os clientes sempre enviam solicitações enquanto o servidor responde as solicitações enviadas, conforme Figura 1. (Oluwatosin, 2014)

Figura 1 - Cliente / Servidor



Fonte: (Oluwatosin, 2014)

2.1 HTTP

O protocolo de transferência de hipertexto (HTTP) é um protocolo de nível de aplicação para informações distribuídas. O HTTP tem sido usado pela iniciativa global World Wide Web desde 1990. (RFC 2068).

2.1.1 Versão 0.9

A primeira versão do HTTP, referido como HTTP / 0.9, era um protocolo simples para transferência de dados brutos em toda a Internet. No (Gamage, 2017) início não possuía uma versão definida, até que foi atribuído o “0.9” para diferenciar das versões mais atuais.

A versão inicial do HTTP possuía apenas um método de chamada “GET”, conforme Figura 2 e não possuía cabeçalhos, o que limitava o protocolo à transmissão de arquivos HTML, como representado na Figura 3. Outra limitação dessa versão eram os erros, não existiam códigos de Erro, e caso ocorresse, era retornado uma página HTML específica.

Figura 2 - Request HTTP/0.9

Fonte: adaptado de (Gamage, 2017)

Figura 3 - Response HTTP/0.9

Fonte: adaptado de (Gamage, 2017)

2.1.2 Versão 1.0

Já o HTTP/1.0, conforme definido pela RFC 1945, melhorou o protocolo, permitindo que as mensagens tenham metainformações sobre os dados transferidos e modificadores na semântica de solicitação / resposta.

Exemplos de melhoria são os códigos de retorno, informando ao cliente (Normalmente o navegador) se houve erro, e o tipo do erro. Adicionalmente, foram incluídos os “Headers”, como o “Content-type”, que informa ao navegador qual o tipo

de informação está sendo enviada pelo servidor, e vice-versa. Com a atualização o protocolo se tornou mais dinâmico, permitindo o envio não só de páginas HTML, mas outros formatos, como imagens e arquivos em geral.

Os métodos aceitos na versão 1.0 são: **GET**, **HEAD**, **POST.** Pode-se analisar um exemplo de *request* e *response* utilizando o HTTP/1.0 na Figura 4 e Figura 5, respectivamente.

Figura 4 - Request HTTP/1.0

```
GET /mypage.html HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)
```

Fonte: adaptado de (Gamage, 2017)

Figura 5 - Response HTTP/1.0

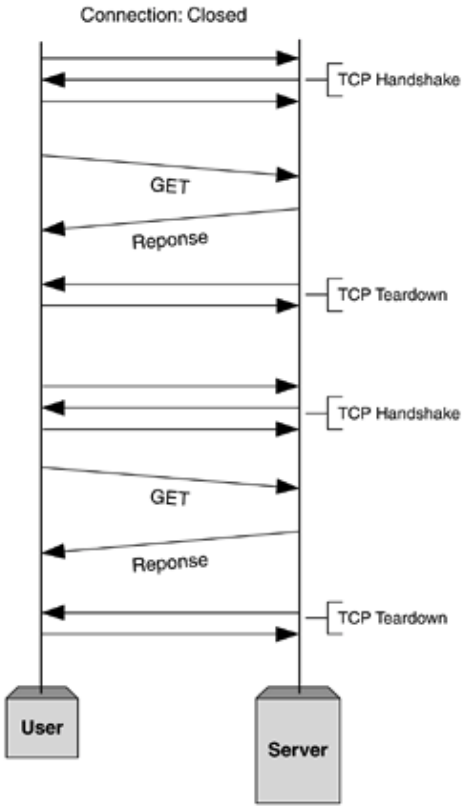
```
200 OK
Date: Tue, 15 Nov 1994 08:12:31 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/html
<HTML>
A page with an image
  <IMG SRC="/myimage.gif">
</HTML>
```

Fonte: adaptado de (Gamage, 2017)

Ao analisar as características do HTTP 1.0, percebem-se diversas falhas que seriam impeditivos para uma adoção do protocolo em escala. De acordo com Baker, Lai e Cheng (1999), o principal problema dessa versão é que para cada objeto em uma página, é aberta uma conexão separada. Por exemplo, cada imagem em uma página é um objeto, e por consequência, no HTTP/1.0, é aberta uma conexão para cada imagem na mesma página.

Na Figura 6 - Multiple Requests HTTP/1.0.Figura 6, perceber-se que para cada pedido do cliente ao servidor, existe um início de comunicação “Handshake”, o request, e então uma chamada para o “TCP Teardown”, que é o encerramento da comunicação pelo canal.

Figura 6 - Multiple Requests HTTP/1.0.



Fonte: adaptado de (InformIT, 2004)

Baker, Lai e Cheng (1999) também citam um segundo problema, que é o formato de pedidos e respostas estrito que a versão do protocolo utiliza, onde um segundo *request* só será feito após o retorno do primeiro. O que é um grande problema em servidores que possuem latência alta.

2.1.3 Versão 1.1

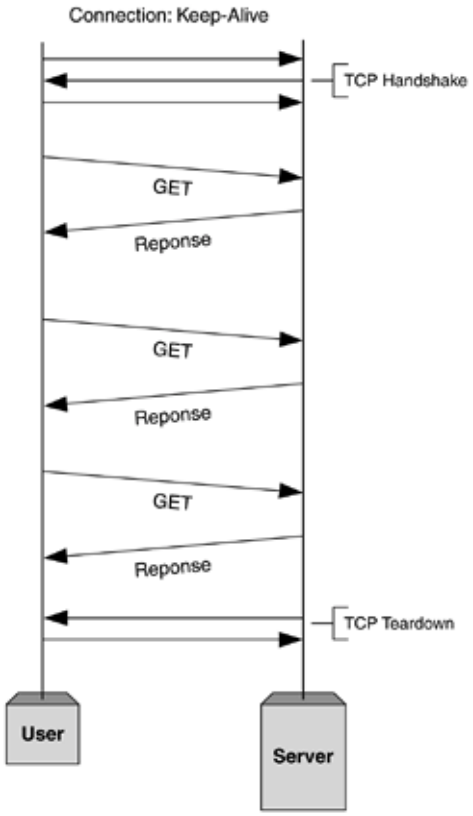
A primeira versão padronizada do protocolo foi a 1.1, que foi publicada em 1997, e é a versão comumente utilizada atualmente pelos servidores e clientes HTTP. (Gamage, 2017)

Os métodos aceitos na versão 1.1 são: **GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS.**

A versão foi criada para resolver os problemas existentes no protocolo na versão 1.0, apontados no item 2.1.2. Na atualização o cliente HTTP cria uma única

conexão persistente com o servidor, que fica aberta e aguardando outras requisições, como se observa na Figura 7. (Cheng, Lai e Baker, 1999).

Figura 7 - Requests/Responses HTTP/1.1

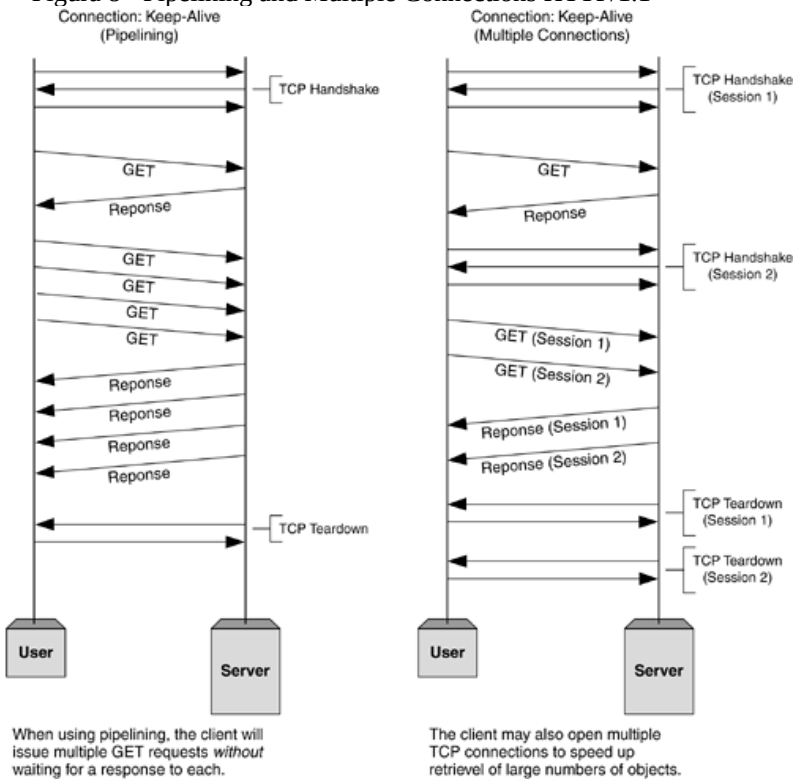


Fonte: adaptado de (InformIT, 2004)

Outras funcionalidades relevantes adicionadas na versão são: Pipeline e Múltiplas conexões. Essas funcionalidades têm, respectivamente, a função de permitir que sejam realizadas mais chamadas na mesma conexão sem que seja necessário o retorno da chamada anterior, e que se possam abrir mais de uma conexão com o mesmo servidor, visando aumentar o desempenho e carregar os objetos de forma assíncrona.

Na Figura 8 há um exemplo das duas funcionalidades.

Figura 8 - Pipelining and Multiple Connections HTTP/1.1



Fonte: (InformIT, 2004)

2.1.4 Versão 2.0

O HTTP/2 é a versão padronizada mais atual do HTTP, foi lançada em 2015, e em 2016 já era utilizada por quase 10% dos sites. De acordo com Varvello (2016), o HTTP/2 “promete tornar a web mais rápida e eficiente compactando cabeçalhos, introduzindo push de servidor, corrigindo o problema de bloqueio de cabeçalho de linha, e carregando elementos da página em paralelo em uma única conexão TCP”.

2.1.5 Upgrade

O HTTP Upgrade é um mecanismo utilizado para realizar uma atualização de protocolo em uma conexão já estabelecida. (Mozilla, 2021). O funcionamento do

mecanismo consiste no envio de um cabeçalho “Upgrade” pelo cliente com os protocolos de Upgrade reconhecidos por ele, o servidor identifica os protocolos propostos e caso aceite a proposta, retorna para o cliente com um status “101 – *Switching Protocols*”, caso nenhum dos protocolos seja aceito, o servidor responde com um status padrão “200 - *OK*).

Um dos principais casos de uso dessa ferramenta é o WebSocket, que inicia com uma conexão HTTP/1.1, e só então faz um Upgrade para o protocolo, inicializando o Handshake. (Mozilla, 2021)

2.2 REST

REST (*REpresentational State Transfer*), inicialmente apresentado em 2000 por Roy Fielding, é uma definição de estilo de arquitetura para aplicativos em rede. Ele existe como uma série de restrições aplicadas à implementação de componentes de rede, permitindo a semântica de interface uniforme, em vez de implementações e sintaxe específicas do aplicativo. (Paessler, 2021)

O REST não é um protocolo, ou um padrão. De acordo com Tilkov (2010), o REST pode ser entendido como uma lista de princípios que definem como os padrões da WEB (HTTP, URIs) devem ser utilizados. A proposta é que uma aplicação que os utilize, aproveite melhor a arquitetura WEB.

Para que a aplicação seja considerada RESTful, ela deve seguir todos estes princípios que serão abordados a seguir.

2.2.1 Interface Uniforme

De acordo com Fielding (2000), “a principal característica que difere o REST de outros estilos e princípios de definição de rede, é a ênfase em uma interface uniforme entre os componentes”.

- A interface de cada recurso deve ser definida previamente, e seguida religiosamente;
- Cada entidade do sistema deve ter apenas uma URL lógica;
- Devem ser utilizados padrões de nomenclaturas, e formatos de dados (JSON/XML/etc.);

- Recursos devem ser acessíveis utilizando uma rota (Endpoint) padrão, um método GET, e alterado da mesma forma, utilizando o método mais apropriado;

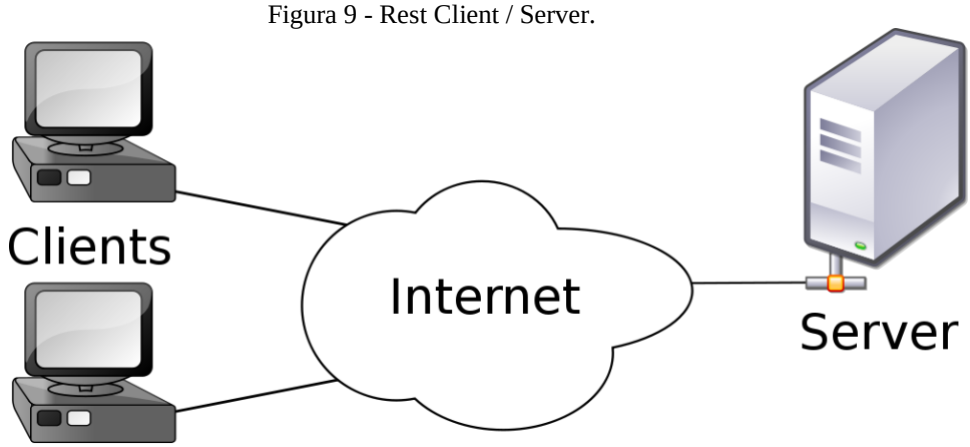
Fielding (2000) aponta também em sua dissertação que o preço dessa generalização, e facilidade de manipulação é a eficiência, visto que em casos que utilizam formatos mais específicos, estes são criados e adaptados para a aplicação em questão, sendo mais performáticos.

2.2.2 Cliente – Servidor

O principio de cliente-servidor é definido pela separação por completo entre o cliente, interface de uma aplicação web por exemplo, e o servidor, onde ocorrem interações com a base de dados e processamentos da lógica de negócio, como é possível visualizar na Figura 9.

Fielding (2000) aponta que realizando essa separação, há uma portabilidade maior nas plataformas utilizadas na interface de usuário, assim como permite uma escalabilidade maior na parte do servidor, por simplificar os componentes utilizados. Essa separação, permite que os dois lados evoluam sem dependências, se adequando as necessidades de escala do mercado.

Figura 9 - Rest Client / Server.



Fonte: (Planski, 2014)

2.2.3 Stateless

Seguindo a separação entre as arquiteturas, o principio stateless indica que toda comunicação efetuada entre cliente e servidor deve ser feita sem o armazenamento do estado do cliente no servidor. Cada chamada deve ser tratada de forma individual pelo servidor, sem considerar que houve uma chamada realizada previamente.

Cada uma das chamadas realizadas pelo cliente deve ser completa, contendo toda a informação necessária para que o servidor possa processá-la sem levar em conta nenhum contexto e informações mantidas por ele. (Fielding, 2000)

De acordo com Fielding (2000) essa limitação afeta positivamente a visibilidade, confiabilidade e escalabilidade da aplicação, o preço pago é o "*overhead*" criado, pois para cada *request* é necessário reenviar dados e metadados que poderiam ser mantidos pelo servidor e reutilizados a cada chamada.

Tilkov (2010) exemplifica o benefício da restrição Stateless em uma situação em que a comunicação entre o cliente e o servidor é interrompida por um período. Nesse caso o cliente se comunica com outros servidores da mesma aplicação sem notar que houve uma queda no servidor que se comunicava inicialmente. Essa dinâmica é possível pois não havia nenhuma informação relevante no servidor que passou por problemas.

2.2.4 Cacheable

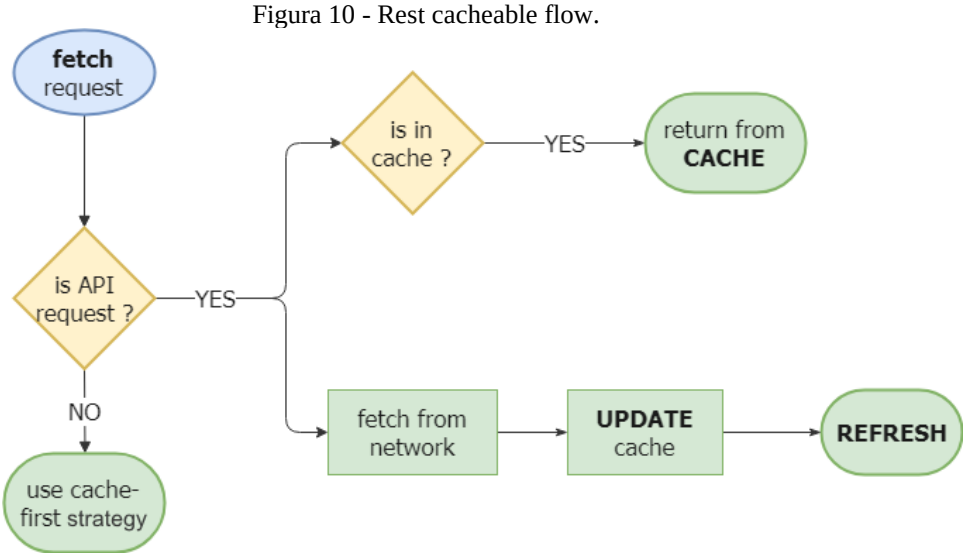
A restrição de cache define que dados de uma resposta feita pelo servidor devem ser marcados como "cacheable" ou "non-cacheable", informando se aquele pedido pode ou não ser armazenado em um servidor de cache, ou, por exemplo, no próprio navegador para ser utilizado posteriormente em uma chamada com os mesmos dados.

"A vantagem de adicionar restrições de cache é que eles têm o potencial de eliminar parcial ou completamente algumas interações, melhorando a eficiência, escalabilidade e desempenho percebido pelo usuário, reduzindo a latência média de uma série de interações" (Fielding, 2000)

Fielding (2000) aponta também que o preço pago para essa restrição do REST é a diminuição na confiabilidade do dado buscado, visto que o dado pode ter

alterações relevantes no servidor, e utilizando um retorno em cache, é possível que haja uma informação desatualizada ou incompleta. Na Figura 10 é possível visualizar o fluxo de utilização de cache.

Figura 10 - Rest cacheable flow.



Fonte: (womakerscode, 2019)

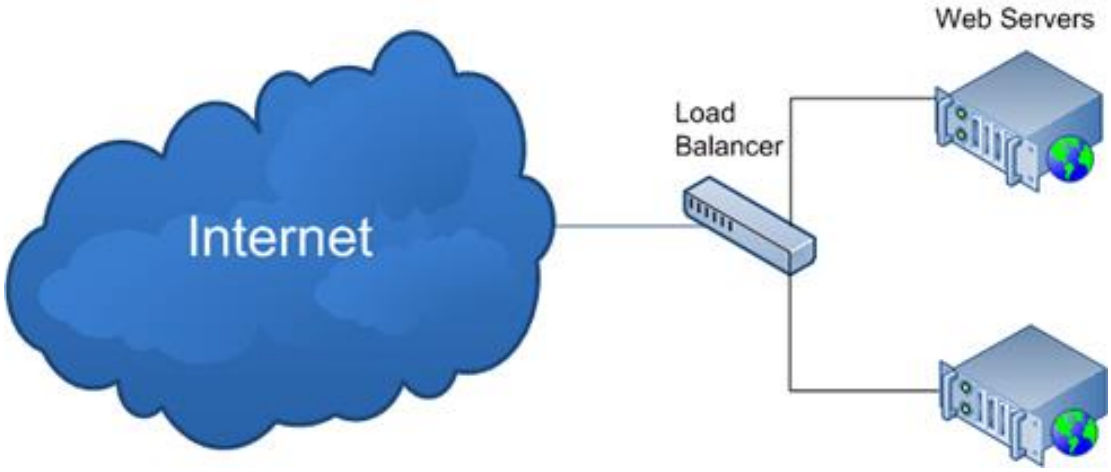
2.2.5 Layered System

A restrição de camadas permite que a aplicação tenha diversas camadas diferentes, porém, cada *layer* dessa aplicação conhece apenas a *layer* com que está se comunicando diretamente, é transparente ao cliente interagindo com um servidor HTTP por exemplo, quantas camadas existem entre o servidor e o banco de dados.

Fielding (2000) cita como adições dessa restrição, a possibilidade de criar Load balancers para tratar os pedidos, melhorando a performance da aplicação, sem o conhecimento do cliente por exemplo.

Na Figura 11 é possível identificar que o cliente se comunicaria diretamente com o *Load Balacer*, sem saber quantas ou quais aplicações ele está interagindo.

Figura 11 - Rest Layering.



Fonte: (Planski 2014)

2.3 POLLING

Os protocolos e modelos de comunicação discutidos até agora concentram-se na dinâmica entre um cliente, que faz solicitações, e um servidor, que as processa e responde. Cada um dos pedidos é feito através de um canal, iniciado pelo cliente e encerrado após a conclusão das chamadas e operações, dependendo da versão do protocolo e o modelo utilizado.

Entretanto, uma característica comum a todos os métodos é que o servidor é um agente passivo, atualizações feitas no servidor só serão vistas, recebidas por um cliente, caso o cliente faça um novo pedido por estas informações.

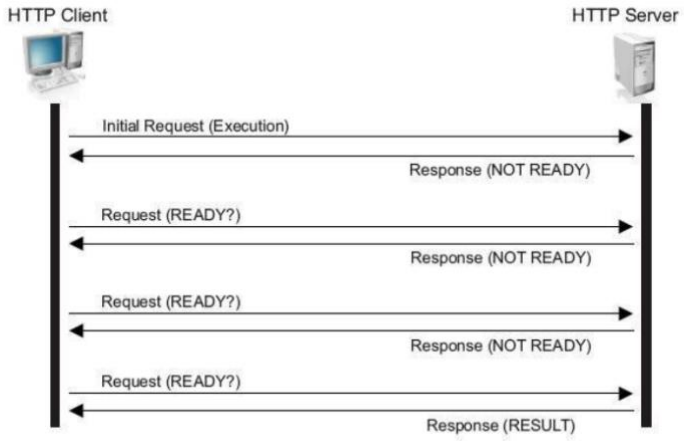
Quando se avalia em um cenário mais estático de aplicações WEB, essas soluções são suficientes, o número de atualizações realizados em uma página HTML que contém um blog, ou um currículo, é muito baixo. Porém, ao trazer um cenário atual para o contexto, as páginas estão se tornando cada vez mais dinâmicas, o que traz à tona uma necessidade por atualizações quase imediatas. As atualizações visam trazer ao usuário/cliente uma experiência mais real, onde as informações se atualizam, sem que seja necessária uma ação por parte do “ser humano”.

Uma das técnicas utilizadas para que essas atualizações sejam recebidas “em tempo real”, é o **polling.**

Gonzales e Lucas (2011), descrevem o funcionamento da técnica de Polling: “o cliente faz solicitações periodicamente ao servidor para obter informações sobre seu pedido original. Cada chamada é separada por um período. Nesta técnica, a

primeira chamada do cliente deve, na verdade, fazer a solicitação original de execução do algoritmo ou programa e as chamadas subsequentes são feitas a fim de verificar se a solicitação original concluiu sua execução". Note-se que embora as solicitações periódicas sejam feitas pelo *client*, elas não demandam a intervenção do usuário. Na Figura 12 é possível visualizar um exemplo da técnica, onde a cada período, denominado por Wikipedia (2020) de *Polling Cycle,* o cliente vai até o servidor buscando informações sobre uma determinada informação, ou pedido feito anteriormente.

Figura 12 - Polling



Fonte: Gonzales, Lucas (2011)

Apesar de conseguir resolver o problema de trazer informações de forma mais ágil e dinâmica do servidor, a técnica de polling apresenta um custo alto em termos de recursos computacionais. Gonzales e Lucas (2011), afirmam que cada um dos *requests* realizados em um ciclo utiliza recursos tanto do cliente, quanto do servidor. Deve-se considerar também a banda necessária para enviar cada uma das informações. Os autores apontam que a chave para ter relativo sucesso com essa técnica é balancear a média de tempo utilizado nos ciclos com a qualidade de experiência do cliente, baseando-se na necessidade da aplicação desenvolvida.

2.4 LONG POLLING

O Long Polling é uma técnica derivada do Polling que consiste em utilizar uma chamada feita pelo cliente por um determinado tempo, antes de responder ao cliente

informando que não tem atualização. O funcionamento da técnica é assim descrito por Gularte (2014):

1. A aplicação (cliente) abre uma conexão com o servidor
2. O servidor segura essa conexão por um tempo determinado (X segundos)
3. Se o tempo passou e nada aconteceu no servidor, é retornado para o cliente com nenhuma informação.
4. Se dentro do tempo alguma atualização aconteceu, o servidor responde imediatamente para o cliente.

De acordo com Appelqvist, Örnmyr (2017), essa técnica é muito similar ao Polling. A diferença é que não é necessária uma série de pedidos que serão retornados ao cliente sem informações relevantes. No Long Polling, o retorno só acontece caso exista um *timeout* ou uma atualização para ser enviada. Os autores apontam também que caso a frequência de atualizações seja igual ao *Polling Cycle*, as duas técnicas tendem a ter a mesma performance.

2.5 WEBSOCKET

Apresentado inicialmente por Michael Carter e Ian Hickson em 2008, e implementado pelo Google Chrome em 2010, o WebSocket é “uma fina camada de transporte construída sobre a pilha TCP / IP de um dispositivo”. (Kilbride-Singh, 2020)

A tecnologia introduzida junto ao HTML5, tem como objetivo fornecer uma comunicação bidirecional entre cliente e servidor, utilizando apenas um socket TCP/IP aberto inicialmente pela aplicação chamada de “cliente”.

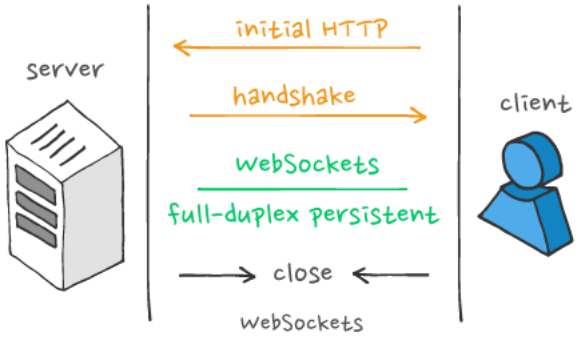
Em um artigo sobre comunicações em tempo real na Web, Liu e Sun (2012) descrevem o protocolo como “uma tecnologia que permite que páginas da web usem o protocolo WebSocket para comunicação full-duplex com um host remoto”. Uma das características mais relevantes no WebSocket é a redução na banda utilizada e na latência quando se compara o protocolo a soluções utilizadas anteriormente, como o Polling e Long Polling.

Carter (2008), no artigo de divulgação do WebSocket define 5 características tidas como essenciais para o protocolo:

- Atravesse firewalls e roteadores sem restrições;
- Permita comunicação entre domínios, se devidamente autorizada;
- Aceite a autenticação baseada em cookies;
- Integre com balanceadores de carga HTTP existentes;
- Seja compatível com dados binários;

Para utilizar o protocolo, o início da conexão requer um *handshake*, que consiste no envio de um pedido por parte do cliente de abertura de conexão, e um aceite do servidor. Na Figura 13 é possível visualizar um fluxo onde a conexão é aberta e o canal "full-duplex" é utilizado por ambos os lados para enviar informações.

Figura 13 - WebSocket handshake



Fonte: adaptado de (Kilbride-Singh, 2020)

A API (*Application Program Interface*) disponibilizada para trabalhar com o protocolo utiliza uma URI (*Uniform Resource Identifier*) que pode ser comparada com uma URI de um *request* HTTP, alterando apenas o prefixo para "ws://", conforme pode-se ver na Figura 14.

Figura 14 - WS API initialize

```
var conn = new WebSocket("ws://www.example.com/livedemo")
```

Fonte: adaptado de (Carter, 2008)

Após essa conexão ser criada é feita uma subscrição aos eventos que podem ser emitidos pelo protocolo, como: “Conexão iniciada”, “Conexão encerrada” e “Mensagens”, conforme pode-se observar na Figura 15Figura 15. (Carter, 2008)

Figura 15 - WS attach events

```
conn.onopen = function(evt) { alert("Conn opened"); }
conn.onread = function(evt) { alert("Read: " + evt.data); }
conn.onclose = function(evt) { alert("Conn closed"); }
```

Fonte: adaptado de (Carter, 2008)

Após a conexão realizada, é possível enviar mensagens utilizando o canal criado, conforme Figura 16.

Figura 16 - WS sending messages

```
conn.send("Hello World")
```

Fonte: adaptado de (Carter, 2008)

Kilbride-Singh (2020), aponta que o protocolo WebSocket padrão tem falhas, como não se recuperar automaticamente de falhas de comunicação, realizando uma reconexão, por exemplo, e dificuldade em trabalhar com navegadores antigos, versões anteriores a 2011.

Atualmente existem diversas aplicações de *clients* que se baseiam na implementação do WebSocket padrão e criam funcionalidades e atalhos para facilitar o trabalho de desenvolvedores.

2.6 SOCKET.IO

Socket.IO é uma biblioteca de código aberto criado por Guillermo Rauch. É construído com Engine.IO, que é uma abstração de baixo nível em cima da tecnologia WebSocket. Socket.IO é usado para comunicação bidirecional entre o lado do servidor e cliente em uma sintaxe que parece que você está apenas disparando e ouvindo eventos. (Cadenhead, 2015)

Segundo (Socket.IO, 2020), o principal motivos de adesão à biblioteca quando foi lançada em 2014 foi o “HTTP long-polling fallback”, recurso que possibilita que um

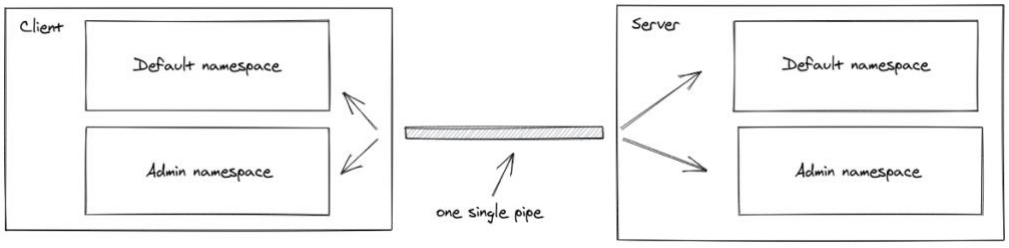
navegador que ainda não seja compatível com WebSocket ainda tenha a experiência de uma comunicação em “tempo real”, utilizando a abordagem long-polling.

Segundo a documentação do Socket.IO (2020), aproximadamente 97% dos navegadores já dão suporte a WebSocket.

Entre as principais características da biblioteca, quando comparado com uma implementação do WebSocket padrão, estão (Socket.IO, 2020):

- Compatibilidade: utiliza a técnica de fallback, caso uma conexão WebSocket não possa ser iniciada com sucesso.
- Reconexão automática: caso a conexão criada inicialmente seja perdida, oferece a opção de reconexão transparente, sem precisar de ação do desenvolvedor.
- Buffer de pacote: Caso a conexão seja perdida, os eventos que seriam enviados ficam em um buffer aguardando a reconexão para tentar novamente o envio.
- Reconhecimento (Callback): Possibilidade de trabalhar com uma função de call-back no cliente para reconhecer que o servidor tratou um *request.*
- Namespace: Separação de mensagens para grupos específicos, também chamados de “Salas”. Exemplo na Figura 17**Error! Reference source not found.**.

Figura 17 - Socket.IO Namespace



Fonte: (Socket.IO - namespace, 2020)

3 BEER GAME

Em 1956 alguns líderes da empresa General Eletric, um conglomerado multinacional de Nova York sediado em Boston, notaram uma grande oscilação nos níveis de produção em uma de suas fábricas, muito maiores do que as oscilações no consumo. Algum tempo depois, professores do MIT, motivados por discussões com estes líderes iniciaram o desenvolvimento do *Beer Game.* (Snyder, 2018)

Snyder (2018), afirma que o jogo foi criado inicialmente pelo professor Jay Forrester, do MIT. Inspirado pela General Eletric, ele desenvolveu uma simulação de um sistema de produção e distribuição, que inicialmente foi feito em uma tabela com papel e caneta. Forrester em seu artigo de 1958 já apontava que uma pequena alteração nos pedidos de consumidores pode afetar os pedidos das outras etapas da cadeia, como varejo, distribuição e fabricação voláteis, o que cria o chamado "*Bullwhip Effect*".

Em 1960, em uma aula de verão, a simulação criada por Forrester foi transformada e utilizada como um jogo pela primeira vez, utilizando um tabuleiro físico e cartas. (Snyder, 2018)

O nome *Beer Game* só foi adotado uma década mais tarde pelo professor do MIT J. Miller, que explicou a escolha do produto pela necessidade que as empresas de cerveja têm de ter sempre o produto disponível para o cliente, e ao mesmo tempo manter um estoque com menor tamanho possível. (Snyder, 2018)

3.1 EFEITO CHICOTE

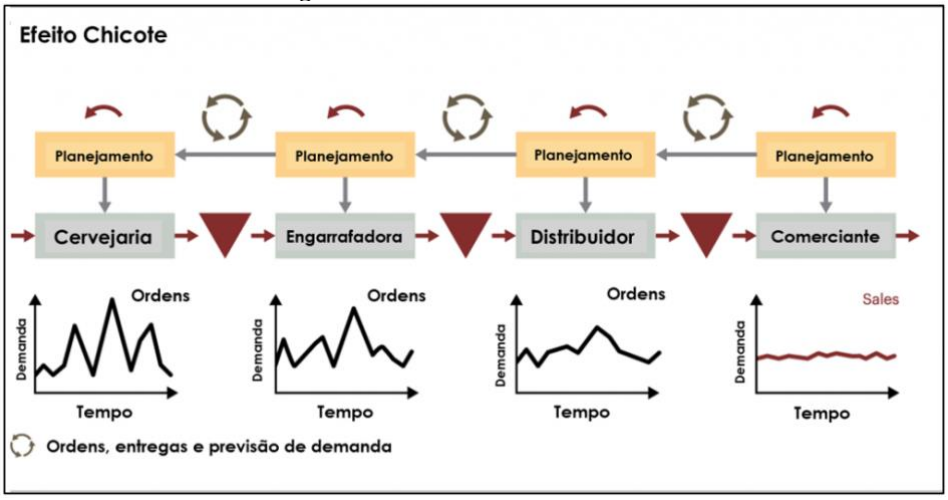
O desempenho logístico tem forte influência sobre o desempenho financeiro de indústrias e comércios. Com o seu desenvolvimento, em conjunto com o advento da Tecnologia da Informação – TI, surge a possibilidade de uma cadeia de suprimentos gerenciada, ou como é chamada, Supply Chain Management – SCM. (Coelho, Follmann e Rodriguez, 2009)

Coelho, Follman e Rodrigues afirmam que um dos problemas estudados e enfrentados por uma SCM é o efeito chicote. Esse efeito é a variação ou a impossibilidade de alinhamento da demanda à oferta. O grande desafio da logística é diminuir o hiato existente entre a produção e a demanda, de modo que os consumidores tenham bens e serviços quando e onde eles quiserem, e na condição

física que desejarem. Os autores definem o efeito chicote, ou "*Bullwhip Effect*", como o "o resultado de uma expectativa de demanda ou oferta que não se realiza, por diversos motivos, entre elas a incapacidade de prever a demanda dos clientes, e que se propaga por todas as empresas da cadeia, influenciando os níveis de estoques, os tamanhos dos pedidos e a produtividade".

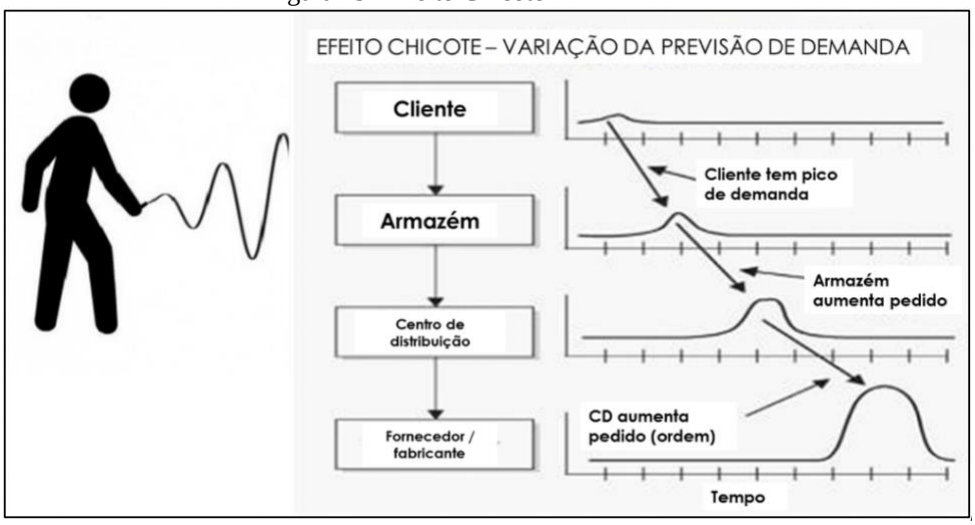
Moori, Perera e Mangini (2009), ilustram o efeito afirmando que em uma cadeia com vários membros, na qual cada um age independentemente do outro e compartilha apenas o mínimo de informações, alterações na demanda do consumidor final, mesmo que pequenas, resultam em uma demanda ampliada e crescente para empresas situadas no início da cadeia de suprimentos. Na Figura 18 e Figura 19 são apresentados exemplos do efeito chicote.

Figura 18 - Efeito Chicote



Fonte: (Nortegubisian, 2016)

Figura 19 - Efeito Chicote Ex2



Fonte: (Nortegubisian, 2016)

4 DESENVOLVIMENTO

Nesta seção será apresentado o funcionamento do jogo Beer Game, utilizado como referência de um jogo online para este trabalho. Serão apresentados desafios encontrados nas diversas abordagens contemporâneas de comunicação web, apresentando benefícios e dificuldades trazidas por cada uma delas.

Serão realizadas simulações práticas para validar os benefícios e dificuldades apresentados de forma teórica por este trabalho. Com as simulações é possível validar e propor, dentre as tecnologias no escopo proposto, a abordagem mais eficiente para jogos como o Beer Game.

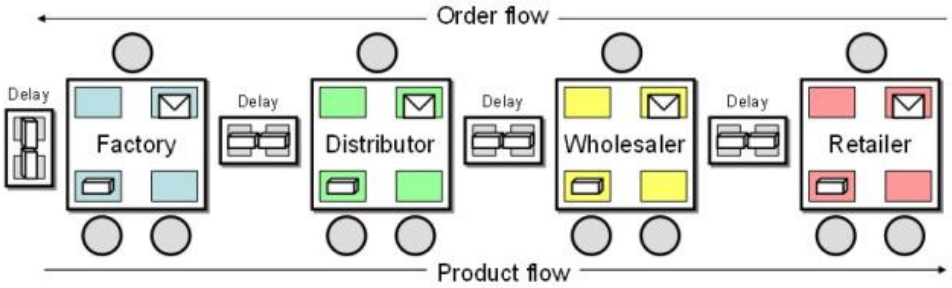
4.1 FUNCIONAMENTO DO BEER GAME

O *Beer Game* simula uma cadeia de suprimentos em quatro estágios: varejista, atacadista, distribuidor e fábrica.

Cada uma das etapas da cadeia tem o objetivo de produzir/entregar unidades de cerveja. A fábrica irá produzir, e as próximas etapas entregam até que o produto chegue ao consumidor final.

O objetivo do jogo é conseguir atender a demanda de cervejas que chegam a cada rodada para cada uma das entidades da cadeia. Na Figura 20 pode-se observar o fluxo das jogadas.

Figura 20 - Beer Game flow



Fonte: BeerGame.org (2021)

Um pedido enviado para a próxima etapa irá levar 2 rodadas até que chegue ao destino, o atraso está representado na Figura 20 pelo "*delay*" entre as etapas.

Cada rodada é representada por uma sequência de passos:

1. Receber pedido
2. Receber entregas entrada
3. Atualizar status do estoque
4. Enviar entregas
5. Realizar pedido

Todos os pedidos recebidos devem ser atendidos, caso não tenha estoque, o pedido fica em espera até que possa ser enviado para o cliente. Itens em estoque tem um custo de R$0,50 por semana, enquanto itens em espera custam R$1,00 por semana.

Seguindo estes passos o principal objetivo do jogo é ter custo reduzido, tendo mínimo em estoque sem deixar de atender os pedidos dos clientes.

Outras regras relevantes são: A fábrica não possui limite de fabricação e os pedidos de clientes são pré-definidos, a fim de estimular o efeito chicote com o tempo.

4.2 DESAFIOS TECNOLÓGICOS DO BEER GAME

Em uma aplicação do jogo beer game, existem diversos desafios tecnológicos que precisam ser analisados e tratados. Conforme o objetivo deste trabalho, serão identificados alguns dos desafios, e proposta uma solução utilizando tecnologias existentes. Nas subseções abaixo serão detalhados estes desafios.

4.2.1 Escalabilidade

Uma aplicação web (*server*) acessível por diversos jogadores (clients), precisa ter um desempenho satisfatório para que a experiência não seja afetada. Um dos desafios da aplicação analisada é que o retorno para os jogadores não se torne lento, ou pior, não seja feito.

É importante que essa aplicação tenha capacidade de processar os diversos pedidos simultâneos feitos pelos jogadores, e sem que seja necessário um recurso físico com custo alto, o que poderia inviabilizar a operação.

4.2.2 Comunicação em tempo real

Para o cumprimento dos objetivos do jogo é importante que os jogadores recebam as informações em tempo real, permitindo que todos tenham as mesmas informações e não sejam prejudicados no andamento da simulação.

Além de garantir que as informações sejam recebidas pelos jogadores de forma sincronizada, é importante ressaltar que apenas as informações relevantes para aquele jogador sejam recebidas por ele, garantindo a segurança dos dados, e eficiência do jogo.

4.2.3 Conectividade e Latência

Por se tratar de uma aplicação web, que se propõe a atender jogadores em locais distintos, e exigir uma sensação de tempo real, uma das principais preocupações se dá em relação à qualidade da conexão e latência que cada jogador terá. O ponto de atenção se dá pela diversidade de situações que podem existir num ambiente de ensino heterogêneo. Em uma análise desses ambientes é possível identificar alguns cenários:

- Instabilidade na conexão: Jogadores em redes distintas podem experimentar quedas e intermitência na conexão, o que é um problema para um jogo em tempo real. Um exemplo seria alguém jogando de uma rede móvel (3G), ou em uma área rural, onde normalmente a tecnologia disponível é internet a rádio, muito suscetível ao tempo;

- Latência: Atrasos na transmissão de dados podem ter efeito negativo em um jogo como o Beer Game, uma latência alta pode fazer com que a transmissão ou recepção de dados seja efetuada com atraso relevante, prejudicando a experiência dos usuários;

Os impactos causados pelos cenários listados são diversos, porém, é possível destacar:

- Experiência do jogo: Atrasos no jogo, ou falta de sincronia nas jogadas executadas podem criar uma experiência desfavorável para o jogador, frustrando e dificultando o objetivo principal que neste caso é o aprendizado;
- Inconsistência na simulação: Como citado na seção do jogo Beer Game, o jogo depende de decisões de jogadores em tempos específicos, caso os jogadores sejam afetados por algum dos problemas acima e não consigam efetuar sua jogada, a simulação será afetada;

4.3 APLICAÇÃO DOS DESAFIOS NAS TECNOLOGIAS ATUAIS

Nesta seção serão avaliados os principais desafios tecnológicos identificados no item 4.24.2 do ponto de vista de algumas tecnologias/protocolos existentes para comunicação web.

4.3.1 Polling

Escalabilidade – O polling tem por essência realizar diversos pedidos para o servidor buscando uma possível atualização, ou aguardando o fim de um processamento. Um número alto de requisições poderá exigir mais recursos do servidor utilizado na aplicação, sendo estes recursos: Processamento, memória alocada, possível acesso ao disco e uso de rede para cada uma das requisições realizadas. O elevado uso de recursos afeta negativamente a escalabilidade da aplicação, principalmente quando considerado que a aplicação pode ter um número elevado de jogadores.

Comunicação em tempo real – Por ser um formato de atualização que depende de cada *client* realizar uma chamada ao servidor, a comunicação em tempo real não é fornecida utilizando o *polling*, cada um dos jogadores tem um tempo distinto para receber a atualização.

Conectividade e Latência – A conectividade pode ser afetada pois uma falha na rede do jogador pode interromper o *request* http realizado, e será necessário esperar o próximo tempo de polling para uma nova chamada.

4.3.2 Long Polling

Escalabilidade – A escalabilidade da aplicação é menos afetada no Long Polling em comparação ao Polling, pois conforme descrito em 2.4, são realizados menos requisições ao servidor buscando atualizações. No entanto, o Long Polling ainda realiza pedidos ao servidor de forma recorrente, o que é um ponto de atenção ao avaliar as métricas de desempenho.

Comunicação em tempo real – Semelhante a escalabilidade, quando se trata de comunicação em tempo real, há um ganho ao utilizar o Long Polling, pois, considerando que todos os clientes possuem uma requisição aberta ao servidor, quando tivermos uma atualização, todos eles irão receber o retorno. Porém, deve-se considerar que algum dos clientes pode não estar com a conexão aberta no momento, o que pode trazer uma latência na sua atualização.

Conectividade e Latência – Com relação aos desafios de conectividade definidos o Long Polling ainda possui dificuldades, visto que ele ainda depende da uma conexão aberta para poder receber uma atualização.

4.3.3 WebSockets

Escalabilidade – Por permitir uma comunicação bidirecional em tempo real entre cliente e servidor, os WebSockets podem lidar com múltiplas conexões de forma simultânea com mais eficiência do que as outras tecnologias apresentadas previamente. A conexão WebSocket é persistente, evitando reconexões, o que reduz consideravelmente a carga de chamadas ao servidor.

Comunicação em tempo real – Por manter uma conexão sempre ativa, a comunicação bidirecional proporcionada pelos WebSockets permite que atualizações sejam enviadas de forma instantânea e sincronizada para todos os clientes. É importante ressaltar também que é possível segmentar as informações de forma que apenas informações relevantes sejam enviadas ao cliente, reforçando a segurança e evitando consumo de recursos desnecessários.

Conectividade e Latência – Com os WebSockets é possível realizar reconexões automáticas quando identificada uma queda, essa função é disponibilizada de forma nativa. Essa função ajuda a mitigar cenários de redes instáveis, como por exemplo 3g ou internet wireless. No aspecto da latência, a conexão sempre aberta trabalha para minimizar efeitos negativos, pois evita a sobrecarga realizada em reconexões HTTP padrão, focando apenas na transmissão dos dados relevantes para a operação.

4.4 EXPERIMENTOS

Para exemplificar e avaliar cada uma das tecnologias e abordagens apresentadas, esta seção detalhará simulações focadas na performance de diferentes cenários. As simulações serão execuções de clientes e servidores de forma a testar

os diversos cenários possíveis em uma aplicação real com múltiplos clientes, como é o Beer Game.

A análise considerará variáveis como a frequência de interações e o número de clientes conectados simultaneamente ao jogo. Será criado um ambiente de testes que emule cenários realísticos, espelhando as demandas de um jogo online ou qualquer aplicação que exija comunicação em tempo quase real entre diversos clientes e o servidor.

4.4.1 Ambiente

O ambiente no qual as simulações foram executadas possui as seguintes características:

- Sistema operacional: MacOS 13.3.1;
- Processador: 2,6 GHz 6-Core Intel Core i7;
- Memória RAM: 16 GB 2400 MHz DDR4;
- Linguagem de programação: Javascript;
- Interpretador: Node.js v16.13.1;

4.4.2 Procedimento

O processo de execução das simulações, detalhadas neste segmento, é orientado por dois parâmetros principais, listados abaixo:

- Tempo: Refere-se aos intervalos de tempo preestabelecidos em que as métricas pertinentes serão coletadas. Esta abordagem visa identificar como as métricas se comportam ao longo da duração da simulação, proporcionando uma visão mais ampla da performance de cada tecnologia em diferentes momentos. Os intervalos de tempo selecionados para a coleta de dados foram:

    - 5 segundos;
    - 30 segundos;
    - 1 minuto;

- 2 minutos;

- Número de Clientes: Este parâmetro é fundamental para a análise proposta neste trabalho, visto que sua variação permitirá compreender melhor as capacidades e limitações de cada tecnologia ao lidar com diferentes volumes de conexões simultâneas. Os valores escolhidos para o número de clientes, em cada simulação, são:

  - 1 cliente;
  - 5 clientes;
  - 10 clientes;
  - 30 clientes;
  - 100 clientes;

Polling

Para realizar a simulação de um ambiente *polling* foi criado um *server* utilizando a biblioteca *express.js* que trata cada requisição feita por cada cliente. Cada simulação possui um cenário de servidor específico que será tratado no próximo tópico deste trabalho.

Para simular o *client* foi utilizado a biblioteca *axios* que, para cada cliente, realiza um pedido a cada 50ms buscando uma atualização do servidor.

Long Polling

As bibliotecas utilizadas na simulação via *long polling* foram similares às utilizadas na simulação do *polling*.

WebSocket

A simulação via WebSocket foi efetuada por meio do módulo "ws" do Node.js que foi empregado tanto no lado do cliente (client) quanto no lado do servidor (server).

O módulo utilizado não fornece nativamente uma opção para reconexão automática após uma possível queda, para corrigir esta limitação, foi implantado o módulo “reconnecting-websocket” no lado do cliente.

Socket.IO

A simulação via Socket.IO utilizou os módulos “socket-io” para o lado do servidor (server) e “socket-io-client” para o lado do cliente (client).

4.4.3 Resultados

Neste tópico serão apresentados os resultados encontrados nas simulações executadas para cada um dos parâmetros e abordagens de comunicação apontados previamente.

Vazão

A vazão de dados (throughput) é a quantidade de dados sendo transferidos de um ponto a outro em um determinado período. No cenário apresentado por este trabalho, pode ser entendido como as jogadas ou ações realizadas por cada um dos jogadores e processadas pelo servidor. Uma taxa alta de vazão de dados indica que os dados e atualizações sejam recebidas e executadas pelos jogadores, o que é essencial para o bom funcionamento do jogo.

Nesta simulação serão comparadas as abordagens para identificar qual oferece uma melhor taxa de vazão para cada cenário proposto.

Funcionamento teste vazão

A simulação de vazão de dados consiste nos seguintes passos:

- Cliente faz uma requisição;
- Servidor recebe a requisição;
- Servidor responde com um corpo de tamanho padrão;
- Cliente recebe a resposta e contabiliza uma requisição com sucesso;

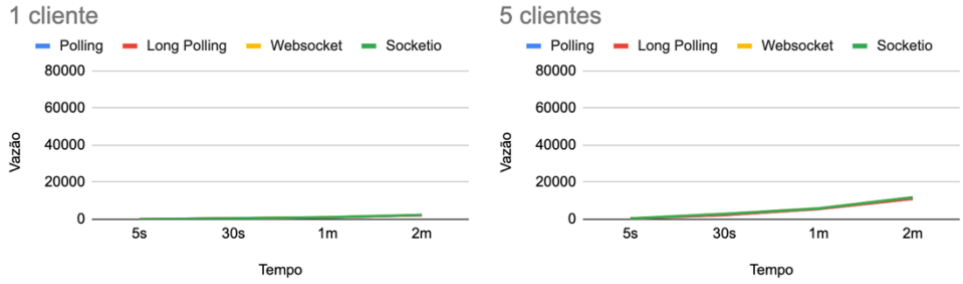
Para cada cenário de número de clientes são iniciadas instâncias que seguem estes passos de forma paralela, com o objetivo de testar um cenário real onde clientes estão buscando e enviando atualizações constantemente.

É importante ressaltar que cada abordagem de comunicação possui uma forma diferente de enviar a requisição.

Resultado teste vazão

Abaixo estão os resultados de vazão separados por número de clientes. Neste teste foram utilizados apenas os cenários de 1, 5, 10 e 30 clientes.

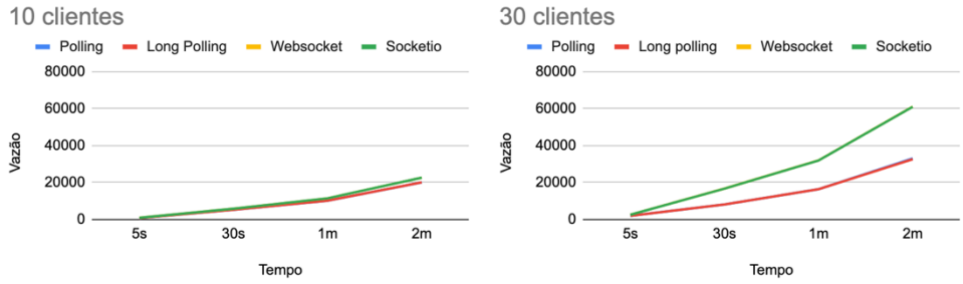
Figura 21 – Vazão 1 e 5 clientes



Fonte: elaborado pelo autor

Avaliando os gráficos apresentados acima, que simulam 1 e 5 clientes é possível identificar que para situações em que há uma quantidade pequena de jogadores todas as abordagens/tecnologias apresentam um resultado semelhante, não sendo possível perceber uma diferença relevante entre elas.

Figura 22 – Vazão 10 e 30 clientes



Fonte: elaborado pelo autor

No entanto, com o aumento da quantidade de jogadores, conforme apresentado nos gráficos de 10 e 30 jogadores, é possível identificar uma diferença considerável entre as abordagens, sendo possível diferenciar que Polling e Long polling possuem uma vazão inferior à apresentada por WebSockets e Socket.IO.

No gráfico a diferença entre Polling e Long polling não é perceptível, assim como WebSocket e Socket.IO, pois os resultados são muito próximos, a imagem

abaixo representa o resultado em forma de tabela para o cenário de 30 cliente, para diferenciar também estas abordagens.

Figura 23 - Vazão tabela de resultados

	5s	30s	1m	2m
Polling	2053	8206	16407	33018
Long polling	2051	8207	16417	32622
Websocket	2655	16752	31991	60971
Socketio	2602	16702	31942	60922

Fonte: elaborado pelo autor

Ao avaliar os resultados apresentados acima, verifica-se que para uma aplicação do jogo Beer Game em ambientes controlados e de poucos jogadores, a abordagem de comunicação é indiferente quando avaliada a vazão de dados, porém, em ambientes mais robustos, por exemplo de uma sala de aula (30 pessoas), é possível identificar que a tecnologia utilizada começa a ter um impacto na performance do jogo.

Latência

A latência (comumente medida em milissegundos) representa o tempo levado para uma informação ir de uma origem até seu destino, podendo ser também identificada como o atraso que a informação teve até que chegasse ao objetivo.

Dentro de um jogo online, por exemplo, uma latência reduzida significa que há um mínimo atraso na transmissão de informações entre os jogadores. Esta característica é importante para buscar uma performance otimizada do jogo, e alcançar os objetivos de aprendizado, especialmente em escopos que buscam atualizações de dados em tempo real.

Funcionamento teste latência

A simulação de latência utilizada neste trabalho consiste nos seguintes passos:

- Cada cliente se conecta, ou busca informações de forma ativa no servidor;
- O servidor gera atualizações com intervalos entre 1 e 2 segundos aleatoriamente;

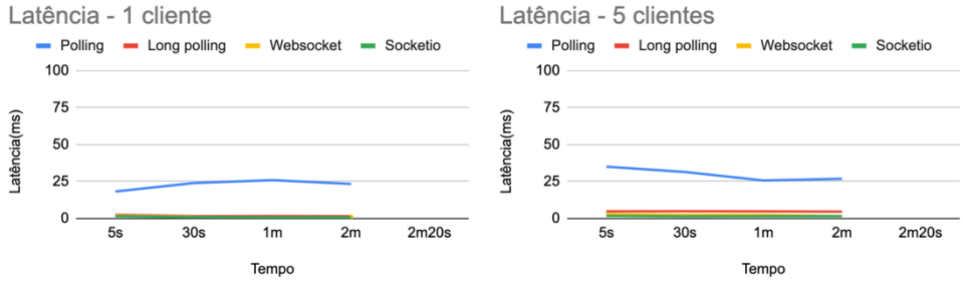
- Cada atualização possui no seu corpo a data de criação da mesma;
- Cada cliente, ao receber a atualização, identifica o tempo entre o recebimento e a criação, contabilizando o atraso;

Com a contabilização dos atrasos feita por cada um dos clientes, foi coletada a informação de latência média para cada abordagem de comunicação nos tempos definidos. Para um melhor entendimento a diferenciação das abordagens, esta simulação teve também um cenário considerando 100 clientes conectados, além dos já previstos.

Resultado teste latência

Abaixo estão os resultados de vazão separados por número de clientes. Neste teste foram utilizados os cenários de 1, 5, 10 e 30 clientes e agregado também um cenário com 100 clientes.

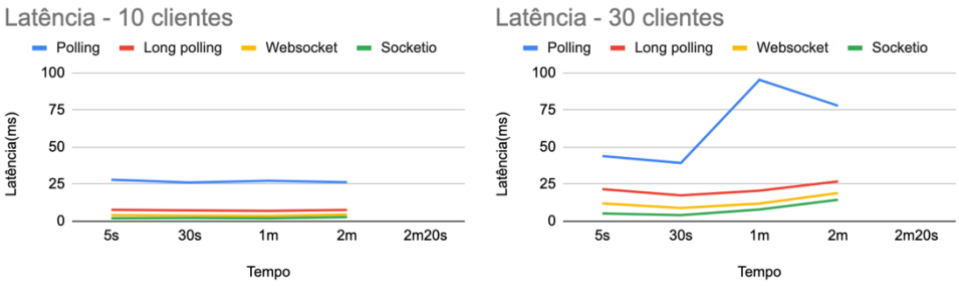
Figura 24 - Latência 1 e 5 clientes



Fonte: elaborado pelo autor

Assim como apresentado na simulação de Vazão de dados, a análise de latência para pequenos números de clientes indica que quase não há diferença entre as abordagens de comunicação utilizadas. Apesar de ser possível identificar uma latência maior na abordagem *polling*, todas se mantêm com uma variação muito baixa e em um valor também reduzido, menos de 50ms, o que não indica impacto no contexto de um jogo educativo como o Beer Game.

Figura 25 - Latência 10 e 30 clientes

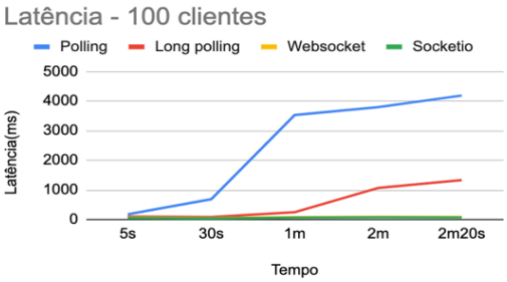


Fonte: elaborado pelo autor

Ao analisar os gráficos dos cenários com 10 e 30 clientes, é possível identificar que, diferente da vazão de dados, o ganho que as abordagens WebSocket e Socket.IO têm sobre o polling é possível de ser visualizado, porém, no contexto estudado, não é relevante.

Identifica-se que já existe um padrão de piora no desempenho do *polling* conforme o número de clientes aumenta, como representado mais claramente no gráfico com o cenário de 30 clientes.

Figura 26 - Latência 100 clientes



Fonte: elaborado pelo autor

Ao analisar o gráfico no cenário de 100 clientes já é possível identificar que a latência média no polling é crescente, indicando que o servidor não consegue responder os clientes na velocidade em que as atualizações e pedidos são realizadas. O mesmo ocorre para o Long polling, com um impacto menor, mas ainda relevante, já ultrapassando 1000ms de latência com 2 minutos de simulação.

WebSocket e Socket.IO não são impactadas de forma relevante, mantendo uma latência de menos de 100ms em toda a simulação.

Recursos (Memória RAM e CPU)

Nesta etapa da simulação, realizou-se um teste para identificar o uso de recursos computacionais, como CPU e Memória RAM, pelo servidor da aplicação ao executar em cada um dos cenários propostos.

Diferentemente dos testes de Vazão e Latência, que são medidas focadas no todo da aplicação, o presente teste tem como objetivo monitorar o lado do servidor, identificando o uso de recursos de cada abordagem.

O consumo de recursos em uma simulação como a proposta pode ser uma variável crucial, pois, é possível que se utilize uma abordagem não otimizada para a aplicação do jogo, e que seja compensado com o aumento de recursos e complexidade da aplicação. Um exemplo possível para tratar uma abordagem não otimizada seria utilizar uma aplicação com diversos processos em paralelo que atendem os diversos pedidos, porém, essa alteração teria impactos não só no tempo de desenvolvimento, mas também no custo de execução.

Desenvolver uma aplicação que ofereça boa latência, capacidade de processar várias mensagens e que não demande muitos recursos computacionais é fundamental para que um jogo, como o proposto, seja viável em termos de custo e qualidade.

Para a coleta de recursos computacionais foi utilizado na simulação o módulo “pidusage”.

Funcionamento teste recursos

A simulação de recursos computacionais utilizada neste trabalho consiste nos seguintes passos:

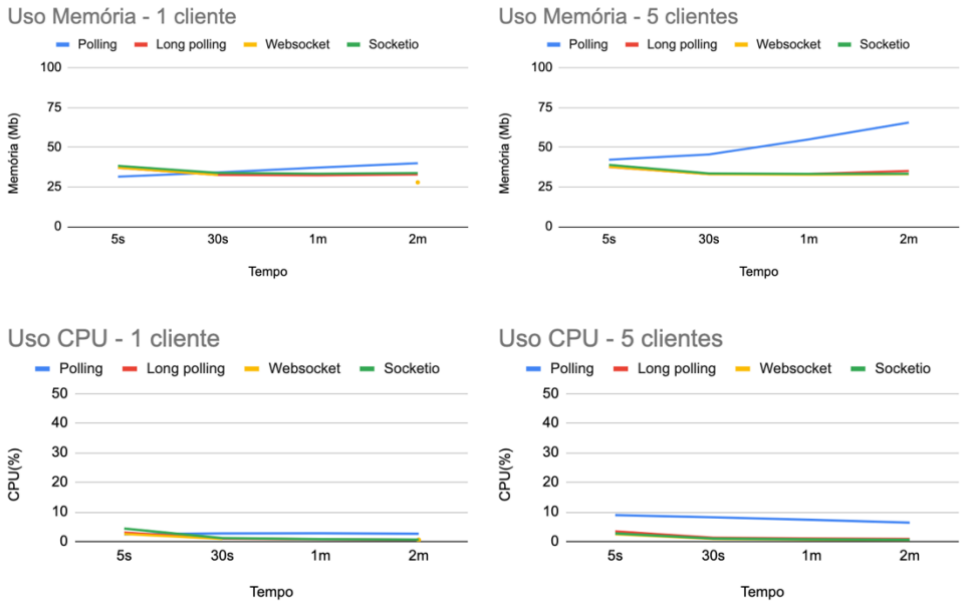
- É executada uma rotina de requisições similar à utilizada no teste de Vazão para exigir do servidor;
- A cada 1 segundo um serviço paralelo solicita ao servidor uma coleta de recursos sendo utilizados pela aplicação, e armazena esta informação;

Em cada período pré-definido neste trabalho é feito um cálculo de média de uso de recursos desde o início da aplicação e armazenado.

Resultado teste recursos

Abaixo estão os resultados de vazão segmentados por número de clientes. No teste foram utilizados os cenários de 1, 5, 10 e 30 clientes e agregado também um cenário com 100 clientes.

Figura 27 - Recursos 1 e 5 clientes



Fonte: elaborado pelo autor

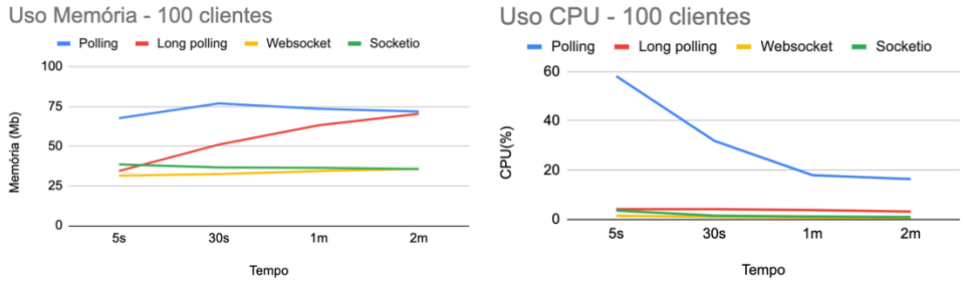
Analisando os gráficos para 1 e 5 clientes é possível inferir que o uso de recursos não é alto, e que qualquer abordagem consegue suprir facilmente a demanda de atualizações com estes parâmetros. Contudo, destaca-se o aumento contínuo de memória na abordagem polling após os 30 segundos.

Figura 28 - Recursos 10 e 30 clientes

Fonte: elaborado pelo autor

No cenário de 10 e 30 clientes é possível identificar desafios para a abordagem de polling. Ao avaliar a memória utilizada, nota-se que não existe um crescimento brusco, o que seria esperado. Porém, ao analisar o uso de CPU é possível perceber um declínio no uso, o que possivelmente ocorre pois o servidor não foi capaz de responder requisições, fazendo com que elas fossem ignoradas.

Figura 29 - Recursos 100 clientes



Fonte: elaborado pelo autor

Como é possível analisar no gráfico de 100 clientes, o uso de memória pouco é afetado quando comparado ao gráfico de 30 clientes, contudo, o gráfico de CPU tem

uma queda muito brusca logo nos primeiros 30 segundos de simulação, indicando que o início da rejeição de requisições iniciou muito antes neste cenário.

4.5 ANÁLISE QUALITATIVA DO SOCKET.IO

Nas análises e simulações apresentadas na seção 4.4, é possível identificar as principais características de cada abordagem, especialmente em ambientes com muitas requisições e necessidade constantes de atualização. O Socket.IO foi, junto com o WebSocket o destaque em quase todas as avaliações, com uma perda mínima de performance nos testes realizados.

No entanto, ao comparar o Socket.IO com a abordagem pura do WebSocket, é possível identificar diversos benefícios, tais como:

- **Compatibilidade**: O Socket.IO oferece adaptabilidade a navegadores que não suportam diretamente a tecnologia WebSocket. Ele realiza um "fallback" para uma comunicação via Long Polling. Esta funcionalidade não apenas inclui suporte para ambientes mais antigos, mas também facilita o desenvolvimento de software;

- **Eventos personalizados**: Com o Socket.IO, os desenvolvedores têm a capacidade de criar eventos personalizados. Permitindo que cada cliente da aplicação escute apenas os eventos que são pertinentes para ele, eliminando a necessidade de um gerenciamento manual. Esta abordagem além de simplificar o desenvolvimento, pode otimizar o uso de recursos tanto do cliente quanto do servidor;

- **Segmentação de clientes**: Utilizando “namespaces” e salas, o Socket.IO promove uma melhoria na segurança, permitindo enviar mensagens apenas para os clientes que deveriam receber cada uma das informações. Essa separação auxilia também na gestão de uso de recursos computacionais;

- **Tratamento de desconexões**
    - **Reconexão automática**: Se um cliente se desconectar, o Socket.IO tentará, automaticamente, restabelecer essa conexão, eliminando a necessidade de intervenções adicionais por parte dos desenvolvedores. Para reduzir a carga no servidor, caso muitas conexões sejam perdidas ao mesmo tempo, a biblioteca possui um mecanismo de “back-off delay”, reduzindo a frequência de tentativas de reconexão de forma incremental**;**

    - **Buffering de mensagens**: A ferramenta fornece de forma nativa uma solução para que, caso um cliente tenha sua conexão interrompida, as mensagens (atualizações) sejam armazenadas e enviadas ao cliente assim que a conexão for reestabelecida;

5 CONCLUSÃO

Este trabalho teve como objetivo analisar um cenário de jogo educativo do ponto de vista de comunicação em tempo real, definindo as principais abordagens utilizadas na atualidade e identificando as dificuldades de cada uma delas. Para atingir este objetivo foram levantadas as tecnologias e realizada uma comparação objetiva de parâmetros considerados relevantes em um ambiente educacional.

Através de uma análise detalhada das principais abordagens de comunicação foi possível identificar que, enquanto os métodos tradicionais, de fácil implementação e manutenção, ainda têm espaço em aplicações com menor escala ou estáticas, a demanda tratada neste trabalho por um ambiente dinâmico e de fácil escalabilidade exige soluções mais avançadas.

Os resultados obtidos neste trabalho sugerem que a escolha da tecnologia de comunicação cliente-servidor é pouco significativa quando se trata de ambientes com até 10 clientes conectados simultaneamente. No entanto, ao ultrapassar essa quantidade de conexões simultâneas, observa-se que as técnicas mais tradicionais, como o polling e o long-polling, começam a apresentar limitações, com uma taxa de vazão de mensagens mais baixa e um aumento no tempo de resposta e no uso de recursos, com destaque para o polling. Por outro lado, as metodologias mais recentes, utilizando WebSocket, mantiveram sua eficiência, mesmo com o aumento no número de clientes. Portanto, para situações com alta concorrência de usuários, a adoção de tecnologias mais modernas pode ser crucial para manter o desempenho e a fluidez da comunicação.

Durante a análise deste trabalho realizou-se uma comparação mais aprofundada das diferenças entre a utilização da comunicação cliente-servidor utilizando WebSocket puro e sua alternativa, o Socket.IO. Observou-se nas simulações que a diferença de performance quantitativa entre as duas abordagens é irrelevante para o cenário de um jogo como o Beer Game. Apesar de o Socket.IO ser altamente baseado na tecnologia WebSocket, foram apresentados os principais benefícios que a utilização da biblioteca trás, focando nas ferramentas disponíveis de forma nativa para otimização de uso de recursos, segurança e facilidades para o desenvolvedor. Foi possível também identificar que, o Socket.IO pode proporcionar

uma solução de melhor usabilidade para os jogadores, com facilidades como a recuperação automática de quedas de conexão e buffering de mensagens.

Este trabalho pôde proporcionar uma visão mais aprofundada da utilização e comparação entre as tecnologias em pontos essenciais para o bom funcionamento de um jogo online como o Beer Game. Contudo, uma análise mais completa do cenário de quedas de conexões se faria necessária para ter uma visão clara de mais benefícios da biblioteca Socket.IO. Uma análise desse parâmetro poderia nos indicar de forma ainda mais clara o quanto um cliente (aluno neste cenário) seria beneficiado por uma escolha mais eficiente da tecnologia utilizada.

5.1 TRABALHOS FUTUROS

Para trabalhos futuros, considerando o cenário do presente trabalho são relevantes e possíveis avanços:

- Análise quantitativa de ganhos para o ambiente do jogo na utilização das ferramentas disponibilizadas pela biblioteca Socket.IO, se atentando principalmente à recuperação de falhas utilizando o back-off delay e o buffering de mensagens.
- Comparação das abordagens propostas em um ambiente utilizando HTTP/2, onde já é disponível o push de servidor e análise do HTTP/3.
- Implementação do jogo utilizando as abordagens apresentadas neste trabalho utilizando load balancer como HAPROXY ou NGINX e múltiplos servidores atendendo os pedidos analisando o impacto dessa tecnologia em ambientes de maior escala.

REFERÊNCIAS

Riverbed – Web Transaction Analysis. Disponível em: https://www.riverbed.com/riverbed-wp-content/uploads/2023/03/SteelCentral-AppResponse11_WTA_Solution_Brief.pdf. Acesso em: 13 set. 2023

OLUWATOSIN, Haroon - Client-Server Model. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1083.8741&rep=rep1&type=pdf. Acesso em: 13 set. 2023

FIELDING, Roy - RFC 2068. Disponível em: http://wwwhomes.doc.ic.ac.uk/~pjm/nac/inet_manual/HTTP.pdf. Acesso em: 13 set. 2023.

BAKER, Mary; **LAI**, Kevin; **CHENG**, Stephen - Analysis of HTTP/1.1 Performance on a Wireless Network. Disponível em: http://shiftleft.com/mirrors/www.hpl.hp.com/personal/Mary_Baker/publications/CSL-TR-99-778.pdf. Acesso em: 13 set. 2023.

GAMAGE, Thilina - Evolution of HTTP — HTTP/0.9, HTTP/1.0, HTTP/1.1, Keep-Alive, Upgrade, and HTTPS. Disponível em: https://medium.com/platform-engineer/evolution-of-http-69cfe6531ba0. Acesso em: 13 set. 2023.

MOZILLA - Protocol upgrade mechanism. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Protocol_upgrade_mechanism. Acesso em: 13 set. 2023.

INFORMIT - Understanding Application Layer Protocols. Disponível em: https://www.informit.com/articles/article.aspx?p=169578. Acesso em: 13 set. 2023.

PAESSLER – IT Explained: REST. Disponível em: https://www.paessler.com/it-explained/rest. Acesso em: 13 set. 2023.

TILKOV, Stephan – A Brief Introduction to REST. Disponível em: https://www.espinosa-oviedo.com/web-programming/files/readings/A-Brief-Introduction-to-REST.pdf. Acesso em: 13 set. 2023.

FIELDING, Roy - Representational State Transfer (REST). Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Acesso em: 13 set. 2023.

PLANSKI, Ricardo – Definição, restrições e benefícios do modelo de arquitetura REST. Disponível em: https://imasters.com.br/desenvolvimento/definicao-restricoes-e-beneficios-modelo-de-arquitetura-rest. Acesso em: 13 set. 2023.

WOMAKERSCODE - Estratégia de Cache / Update / Refresh para GET requests de REST API. Disponível em: https://womakerscode.gitbook.io/pwa-workshop/4.-estrategia-de-cache-para-rest-api. Acesso em: 13 set. 2023.

SNYDER, Larry – A Brief History of the Beer Game. Disponível em: https://medium.com/opex-analytics/a-brief-history-of-the-beer-game-7dd3c325766e. Acesso em: 13 set. 2023.

CADENHEAD, Tyson– Socket.IO Cookbook. Disponível em: https://books.google.com.br/books?hl=pt-BR&lr=&id=BgOACwAAQBAJ&oi=fnd&pg=PP1&dq=socket.io+&ots=b2N-P-Ck-a&sig=aUkZYWpMupAlKZ3BtBgEYz5s-hA&redir_esc=y#v=onepage&q=socket.io&f=false. Acesso em: 02 nov. 2023.

VARVELLO, Matteo– To HTTP/2, or Not To HTTP/2, That Is The Question. Disponível em: https://arxiv.org/pdf/1507.06562.pdf. Acesso em: 20 set. 2023.

APPELQVIST, Rasmus; **ORNMYR,** Oliver– Performance comparison of XHR polling, Long polling, Server sent events and Websockets. Disponível em: https://www.diva-portal.org/smash/get/diva2:1133465/FULLTEXT01.pdf. Acesso em: 20 set. 2023.

Socket.IO – Introduction to Socket.IO. Disponível em: https://socket.io/docs/v4/. Acesso em: 30 set. 2023.

GONZALES, Carlos; **LUCAS,** Simon– A web tool for monitoring HTTP asynchronous method invocations. Disponível em: https://www.researchgate.net/publication/261051143_A_web_tool_for_monitoring_HTTP_asynchronous_method_invocations. Acesso em: 30 set. 2023.

Wikipedia – Polling (computer science). Disponível em: https://en.wikipedia.org/wiki/Polling_(computer_science). Acesso em: 30 set. 2023.

GULARTE, Gabriel – Entendendo Aplicações Web Realtime. Disponível em: https://medium.com/@gabrielfgularte/understanding-realtime-web-apps-5aef47d51a45. Acesso em: 30 set. 2023.

KILBRIDE-SINGH, Kieran – WebSockets vs Server-Sent Events. Disponível em: https://medium.com/ably-realtime/websockets-vs-server-sent-events-9344f164ed79

. Acesso em: 30 set. 2023.

LIU, Qigang; **SUN,** Xiangyang– Research of Web Real-Time Communication Based on Web Socket. Disponível em: https://www.scirp.org/journal/paperinformation.aspx?paperid=25428. Acesso em: 30 set. 2023.

COELHO, Leandro; **FOLLMANN,** Neimar; **RODRIGUEZ,** Carlos– O impacto do compartilhamento de informações na redução do efeito chicote na cadeia de abastecimento. Disponível em: https://www.scielo.br/j/gp/a/cxGzDtxr54FjZk5TygfNLSx/?format=pdf&lang=pt. Acesso em: 30 set. 2023.

MOORI, Roberto; **PERERA,** Luiz; **MANGINI,** Eduardo– UMA ANÁLISE INVESTIGATIVA DO EFEITO CHICOTE NA CADEIA DE

SUPRIMENTOS DA INDÚSTRIA ALIMENTÍCIA. Disponível em: https://www.revistas.usp.br/rege/article/download/36748/39469/43285. Acesso em: 30 set. 2023.

NORTEGUBISIAN – Efeito Chicote e a Gestão de Estoques. Disponível em: https://www.nortegubisian.com.br/blog/efeito-chicote-e-a-gestao-de-estoques/. Acesso em: 30 set. 2023.

Beergame.org – Structure and Rules. Disponível em: https://beergame.org/the-game/structure-rules/. Acesso em: 30 set. 2023.

APENDICE A -- Artigo monografia

COMUNICAÇÃO EM TEMPO REAL PARA JOGO EDUCATIVO BEER GAME

Mateus C. de Souza

1Curso de Sistema de Informação - Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis - SC - Brasil

mateus.carlos@grad.ufsc.br

Abstract. *This research evaluates real-time communication technologies for educational online games, using the Beer Game as a reference. The study compares XHR Polling, Long Polling, WebSockets, and Socket.IO, focusing on data throughput, latency, and resource usage with different user loads. Findings suggest that while technology choice is less critical for up to 10 users, WebSockets perform better for larger groups. Socket.IO stands out for its resource optimization and stability in unstable connections. The study recommends Socket.IO for educational games, proposing future research in connection instability scenarios.*

Resumo. *Esta pesquisa avalia tecnologias de comunicação em tempo real para jogos online educativos, usando o Beer Game como referência. O estudo compara XHR Polling, Long Polling, WebSockets e Socket.IO, focando na vazão de dados, latência e uso de recursos com diferentes cargas de usuários. Os resultados sugerem que, enquanto a escolha da tecnologia é menos crítica para até 10 usuários, WebSockets se sobressaem em grupos maiores. O Socket.IO destaca-se pela otimização de recursos e estabilidade em conexões instáveis. O estudo recomenda o Socket.IO para jogos educativos, propondo pesquisas futuras em cenários de instabilidade de conexão.*

1. INTRODUÇÃO

Neste trabalho, aborda-se o uso de dinâmicas de ensino através de jogos, com foco no Beer Game, um simulador de cadeia de abastecimento criado pelo MIT na década de 1960, que exemplifica o efeito chicote na cadeia de distribuição. O objetivo principal é analisar e aprimorar a comunicação em tempo real entre cliente e servidor no Beer Game, utilizando a

tecnologia WebSocket, para representar de forma eficaz a dinâmica do mercado e garantir a execução fluida e tolerante a falhas do jogo. Os objetivos incluem o mapeamento de tecnologias de comunicação web, análise detalhada dessas tecnologias, e avaliação prática do jogo com WebSockets, visando aprimorar a experiência dos jogadores e comparar com outras abordagens de comunicação contemporâneas. Este estudo propõe uma solução prática para melhorar a comunicação dinâmica em ambientes educacionais através de simulações de jogos.

2. CONCEITOS

Nesta seção serão apresentados os conceitos e fundamentos necessários para a análise deste trabalho. Serão identificadas as principais características das tecnologias estudadas.

2.1 CLIENTE-SERVIDOR

O princípio de cliente-servidor é definido pela separação por completo entre o cliente, interface de uma aplicação web por exemplo, e o servidor, onde ocorrem interações com a base de dados e processamentos da lógica de negócio.

Fielding (2000) aponta que realizando essa separação, há uma portabilidade maior nas plataformas utilizadas na interface de usuário, assim como permite uma escalabilidade maior na parte do servidor, por simplificar os componentes utilizados. Essa separação, permite que os dois lados evoluam sem dependências, se adequando as necessidades de escala do mercado.

2.2 POLLING

Tradicionalmente, em aplicações web estáticas, o servidor atua passivamente, atualizando informações apenas quando solicitado pelo cliente. Com o aumento das páginas web dinâmicas, surge a necessidade de atualizações quase imediatas, sem intervenção do usuário. O polling é uma abordagem de comunicação que visa atender a essa necessidade: o cliente faz solicitações periódicas ao servidor para verificar o status de um pedido anterior, sem necessidade de ação humana. Esta técnica envolve ciclos regulares de solicitações (Polling Cycle) do cliente ao servidor. Apesar de eficaz na entrega rápida de informações, o polling consome significativos recursos computacionais tanto do cliente quanto do servidor, e requer um balanceamento entre a frequência dos ciclos e a qualidade da experiência do usuário, variando conforme a aplicação.

2.3 LONG POLLING

O Long Polling é uma variação do Polling tradicional, com o objetivo de otimizar a comunicação entre cliente e servidor para atualizações de dados. Gularte (2014) descreve o processo da seguinte maneira: o cliente inicia uma conexão com o servidor, que mantém essa conexão aberta por um tempo pré-definido. Se não ocorrerem atualizações nesse intervalo, o servidor retorna ao cliente sem informações. Porém, se houver alguma atualização durante esse período, o servidor responde imediatamente. Appelqvist e Örnmyr (2017) destacam que, ao contrário do Polling convencional, que envolve múltiplas solicitações sem dados significativos, o Long Polling retorna ao cliente apenas se houver um timeout ou atualizações a serem enviadas. Se a frequência de atualizações for similar ao ciclo de Polling, ambas as técnicas apresentam performance comparável.

2.4 WEBSOCKET

O WebSocket é uma tecnologia de comunicação bidirecional entre cliente e servidor, integrada ao HTML5. Ela estabelece um único socket TCP/IP, permitindo interações full-duplex, isto é, comunicação simultânea em ambas as direções. Comparado a métodos anteriores como Polling e Long Polling, o WebSocket se destaca pela menor utilização de banda e redução na latência.

O protocolo é definido por Carter (2008) com características essenciais como a capacidade de atravessar firewalls e roteadores, permitir comunicações entre domínios autorizados, suportar autenticação baseada em cookies, integrar-se a balanceadores de carga HTTP e ser compatível com dados binários. A conexão via WebSocket inicia com um handshake: o cliente solicita a abertura da conexão e o servidor aceita. Uma vez estabelecida a conexão, os eventos podem ser subscritos, como "Conexão iniciada", "Conexão encerrada" e "Mensagens". A API do WebSocket utiliza uma URI com prefixo “ws://” para iniciar a conexão.

Apesar de suas vantagens, Kilbride-Singh (2020) identifica limitações no protocolo WebSocket padrão, como a incapacidade de se recuperar automaticamente de falhas de comunicação e desafios em trabalhar com navegadores antigos. No entanto, existem implementações e ferramentas que complementam o WebSocket padrão, oferecendo funcionalidades adicionais e facilitando o trabalho dos desenvolvedores.

2.5 SOCKET.IO

Socket.IO é uma biblioteca de código aberto desenvolvida por Guillermo Rauch, construída sobre Engine.IO, uma abstração do WebSocket. Ela é utilizada para comunicação bidirecional entre cliente e servidor, com uma abordagem baseada em eventos. Um dos atrativos da Socket.IO, lançada em 2014, é o recurso de "HTTP long-polling fallback", que permite a comunicação em tempo real mesmo em navegadores não compatíveis com WebSocket, usando a abordagem long-polling.

3. DESENVOLVIMENTO

3.1 DESAFIOS TECNOLOGICOS

Escalabilidade

Uma aplicação web (server) acessível por diversos jogadores (clients), precisa ter um desempenho satisfatório para que a experiência não seja afetada. Um dos desafios da aplicação analisada é que o retorno para os jogadores não se torne lento, ou pior, não seja feito. É importante que essa aplicação tenha capacidade de processar os diversos pedidos simultâneos feitos pelos jogadores, e sem que seja necessário um recurso físico com custo alto, o que poderia inviabilizar a operação.

Comunicação em tempo real

Para o cumprimento dos objetivos do jogo é importante que os jogadores recebam as informações em tempo real, permitindo que todos tenham as mesmas informações e não sejam prejudicados no andamento da simulação. Além de garantir que as informações sejam recebidas pelos jogadores de forma sincronizada, é importante ressaltar que apenas as informações relevantes para aquele jogador sejam recebidas por ele, garantindo a segurança dos dados, e eficiência do jogo.

Conectividade e Latência

Por se tratar de uma aplicação web, que se propõe a atender jogadores em locais distintos, e exigir uma sensação de tempo real, uma das principais preocupações se dá em relação à qualidade da conexão e latência que cada jogador terá. O ponto de atenção se dá pela diversidade de situações que podem existir num ambiente de ensino heterogêneo.

4. EXPERIMENTO E RESULTADOS

Para exemplificar e avaliar cada uma das tecnologias e abordagens apresentadas, esta seção detalhará os experimentos realizados. O experimento consiste em execuções de clientes e servidores de forma a validar os diversos cenários possíveis em uma aplicação real com múltiplos clientes, como é o Beer Game.A análise considera variáveis como a frequência de interações e o número de clientes conectados simultaneamente ao jogo.

AMBIENTE DO EXPERIMENTO

- SISTEMA OPERACIONAL: MACOS 13.3.1;
- PROCESSADOR: 2,6 GHZ 6-CORE INTEL CORE I7;
- MEMÓRIA RAM: 16 GB 2400 MHZ DDR4;
- LINGUAGEM DE PROGRAMAÇÃO: JAVASCRIPT;
- INTERPRETADOR: NODE.JS V16.13.1;

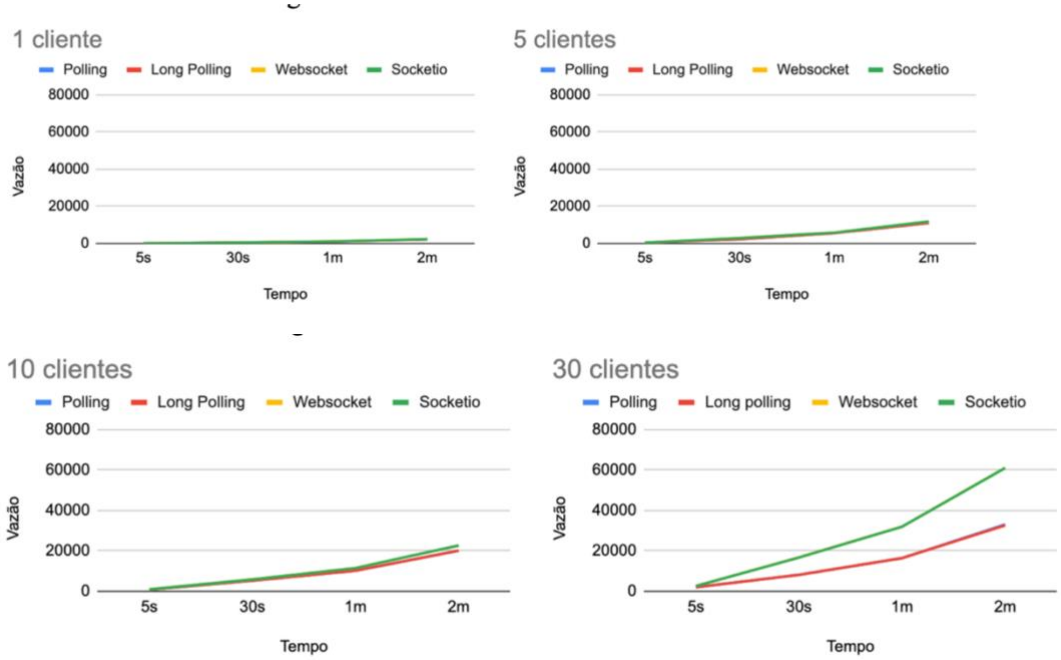
4.1. RESULTADOS

Os experimentos realizados têm como objetivo identificar métricas que identificam a eficiência de cada abordagem de comunicação de acordo com os desafios tecnológicos apresentados. As métricas selecionadas foram Vazão de dados (throughput), Latência e uso de recursos computacionais.

Os testes tem como base um ambiente de cliente servidor onde o servidor simula um ambiente de jogo online, e vários clientes buscam atualizações constantemente.

Vazão de dados

A vazão de dados é a quantidade de dados sendo transferidos de um ponto ao outro em um período. O experimento teve por objetivo identificar o número de atualizações tratadas por cada uma das abordagens de acordo com o número de clientes ativos simultaneamente. Abaixo são apresentados os resultados.



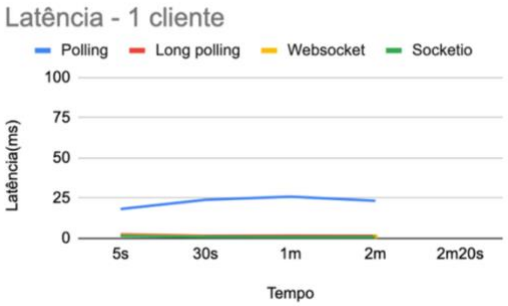
Avaliando os gráficos apresentados acima, que simulam 1 e 5 clientes é possível identificar que para situações em que há uma quantidade pequena de jogadores todas as abordagens/tecnologias apresentam um resultado semelhante, não sendo possível perceber uma diferença relevante entre elas.

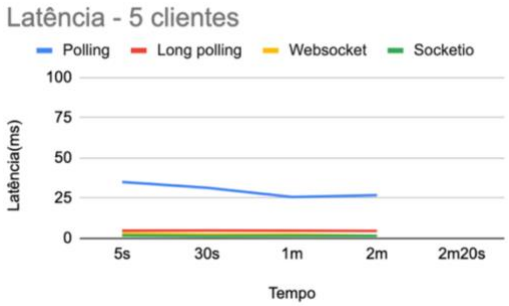
No entanto, com o aumento da quantidade de jogadores, conforme apresentado nos gráficos de 10 e 30 jogadores, é possível identificar uma diferença considerável entre as abordagens, sendo possível diferenciar que Polling e Long polling possuem uma vazão inferior à apresentada por WebSockets e Socket.IO.

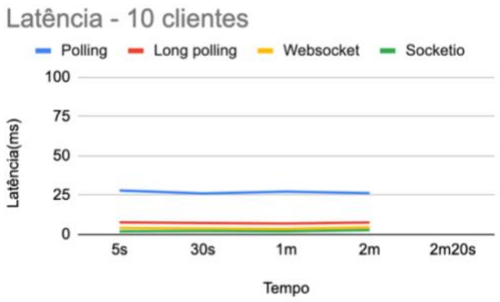
Latência

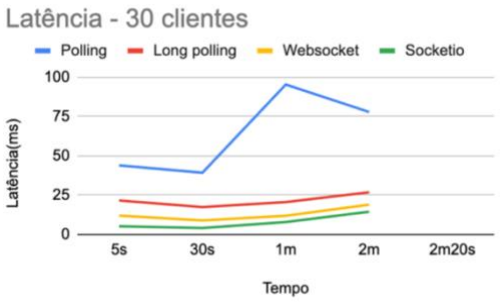
A latência (comumente medida em milissegundos) representa o tempo levado para uma informação ir de uma origem até seu destino, podendo ser também identificada como o atraso que a informação teve até que chegasse ao objetivo.

A simulação de latência simula clientes conectando-se ativamente a um servidor que gera atualizações aleatórias a cada 1-2 segundos. Cada atualização inclui a data de criação, e os clientes medem o atraso entre recebimento e criação desta atualização para calcular a latência média. Abaixo são apresentados os resultados.

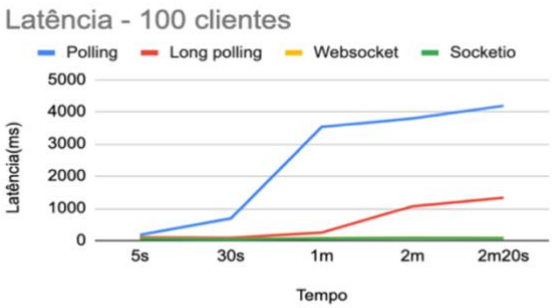








A análise de latência para pequenos números de clientes indica que quase não há diferença entre as abordagens de comunicação utilizadas. Cenários com 10 e 30 clientes indicam que, diferente da vazão de dados, existe um ganho nas abordagens WebSocket e Socket.IO quando comparadas ao polling.

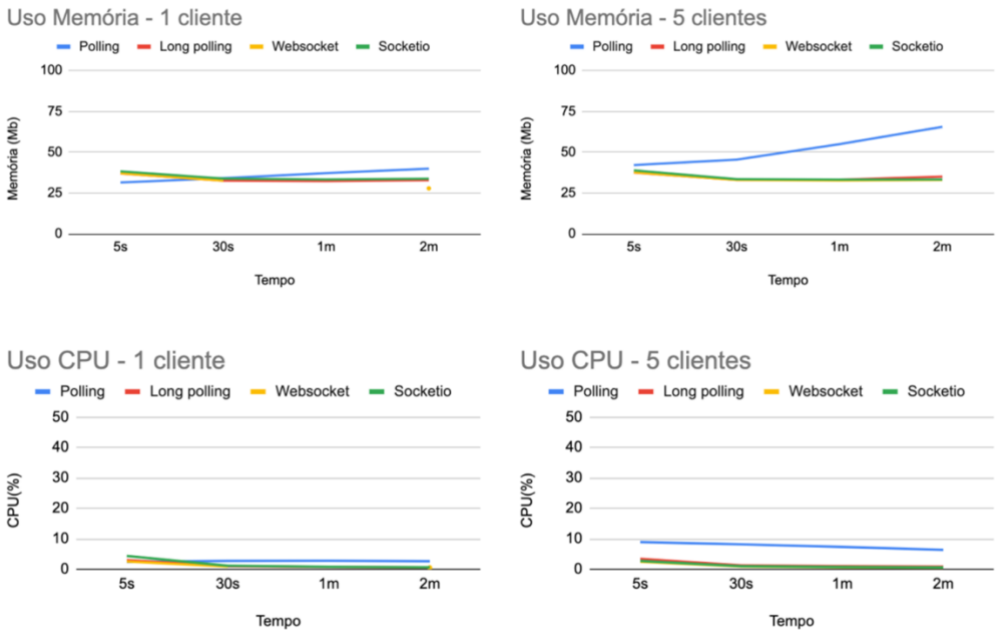


No cenário de 100 clientes já é possível identificar que a latência média no polling é crescente, indicando que o servidor não consegue responder os clientes na velocidade em que as atualizações e pedidos são realizadas. O mesmo ocorre para o Long polling, com um impacto menor, mas ainda relevante, já ultrapassando 1000ms de latência com 2 minutos de simulação.

Recursos (Memória e CPU)

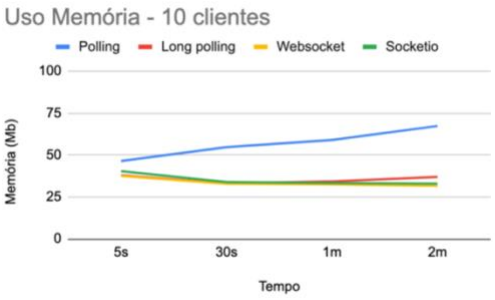
O consumo de recursos em uma simulação como a proposta pode ser uma variável crucial, pois, uma abordagem de comunicação não otimizada para a aplicação do jogo, pode ser compensada com o aumento de recursos e complexidade da aplicação, gerando custos desnecessários e um tempo de desenvolvimento maior do que o esperado.

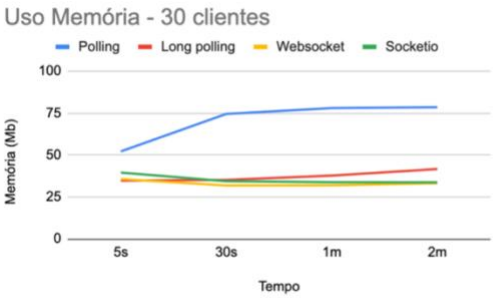
A simulação de recursos computacionais envolve executar rotinas de requisições ao servidor, semelhantes aos testes de vazão. Um serviço paralelo coleta e armazena dados sobre o uso de recursos a cada segundo. Periodicamente, calcula-se e armazena-se a média de uso de recursos desde o início da aplicação.

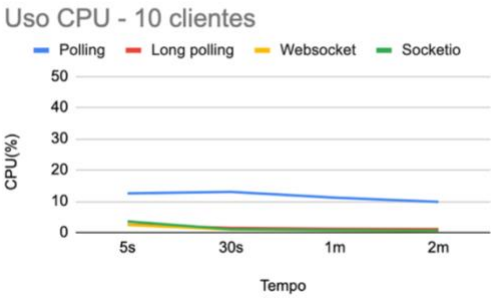


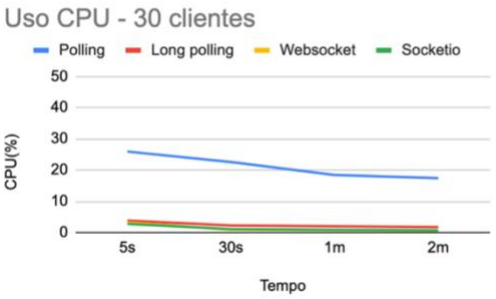
a

Para simulações com 1 e 5 clientes verifica-se que o uso de recursos não é alto, e que qualquer abordagem consegue suprir facilmente a demanda de atualizações.

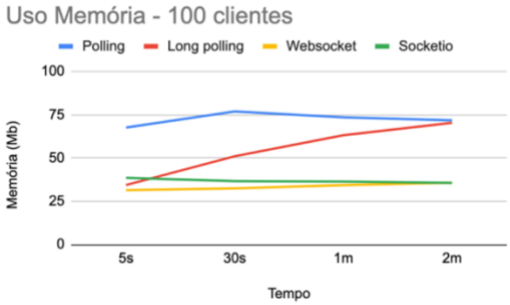


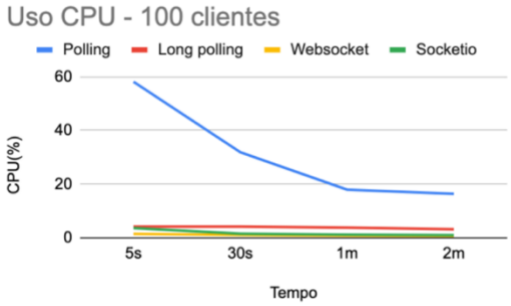






A simulação de 10 e 30 clientes apresenta uma dificuldade na abordagem de polling para processar todos os pedidos. Pode ser observado que o uso de memória fica constante enquanto o uso de CPU apresenta um declínio, que pode ser explicado pelo servidor ignorando novas requisições.





No gráfico de 100 clientes, o uso de memória pouco é afetado quando comparado ao gráfico de 30 clientes, contudo, o gráfico de CPU tem uma queda muito brusca logo nos

primeiros 30 segundos de simulação, indicando que o início da rejeição de requisições iniciou muito antes neste cenário.

Socket.IO

Nas análises e simulações apresentadas, o Socket.IO, em conjunto com o WebSocket, destaca-se por uma performance consistente com mínima perda nos testes realizados. Comparando-o com o WebSocket em sua forma pura, o Socket.IO revela uma série de vantagens significativas. O principal diferencial sendo a adaptabilidade a navegadores que não suportam nativamente o WebSocket, optando por Long Polling como alternativa. Esta flexibilidade não apenas amplia a compatibilidade com ambientes mais antigos, mas também simplifica o processo de desenvolvimento.

O Socket.IO permite a criação de eventos personalizados, permitindo que os clientes da aplicação se concentrem apenas nos eventos que são relevantes para eles, otimizando o uso de recursos e simplificando a gestão. A ferramenta também promove uma melhoria na segurança e na eficiência da comunicação, através da segmentação de clientes usando "namespaces" e salas. Outras funcionalidades, como a reconexão automática e o buffering de mensagens, garantem a continuidade da comunicação mesmo em casos de desconexão, demonstrando a superioridade do Socket.IO em termos de flexibilidade, segurança e eficiência na comunicação para desenvolvimento de aplicações web.

5. CONCLUSÃO

Este trabalho focou em analisar o cenário de um jogo educativo sob a ótica da comunicação em tempo real, destacando as principais abordagens atuais e suas respectivas dificuldades. Para isso, foram avaliadas diferentes tecnologias e feita uma comparação baseada em parâmetros relevantes para ambientes educacionais. A análise detalhada revelou que métodos tradicionais de comunicação, como polling e long-polling, adequados para aplicações menores ou estáticas, se mostram limitados em ambientes dinâmicos e escaláveis, exigindo soluções mais avançadas para atender às necessidades de um jogo educativo interativo.

Os resultados do estudo indicaram que a escolha da tecnologia de comunicação cliente-servidor tem impacto limitado em cenários com até 10 clientes conectados simultaneamente. No entanto, ao ultrapassar esse número, as limitações das técnicas tradicionais, como o polling, tornam-se evidentes, com redução na taxa de vazão de mensagens e aumento no tempo de resposta e no uso de recursos. Em contraste, tecnologias mais modernas, como o WebSocket, mantiveram sua eficiência independentemente do aumento no número de usuários, sugerindo sua importância em situações de alta concorrência.

Uma comparação mais aprofundada entre o WebSocket puro e o Socket.IO revelou que, embora a diferença de desempenho seja mínima para jogos como o Beer Game, o Socket.IO oferece vantagens adicionais, como otimização no uso de recursos, segurança e facilidades para desenvolvedores. Destacam-se também as funcionalidades de recuperação automática de conexões perdidas e buffering de mensagens, melhorando a usabilidade para os jogadores. Contudo, este estudo ressalta a necessidade de análises mais completas sobre quedas de conexão para entender melhor os benefícios do Socket.IO, especialmente em relação à experiência do usuário em jogos online educativos.

REFERÊNCIAS

KILBRIDE-SINGH, Kieran (2020) – WebSockets vs Server-Sent Events

APPELQVIST, Rasmus; ORNMYR, Oliver (2017) – Performance comparison of XHR polling, Long polling, Server sent events and Websockets.

FIELDING, Roy (2000) - Representational State Transfer (REST)

GULARTE, Gabriel – Entendendo Aplicações Web Realtime.

COELHO, Leandro; FOLLMANN, Neimar; RODRIGUEZ, Carlos (2009)– O impacto do compartilhamento de informações na redução do efeito chicote na cadeia de abastecimento.

GONZALES, Carlos; LUCAS, Simon(2011)– A web tool for monitoring HTTP asynchronous method invocations.