Benchmark Entre Diferentes Linguagens *Server-side* Implementando o *TUS* – Protocolo Aberto Para *Upload* de Arquivos Recuperáveis

Luiz Felippe A. Gomes, Giuliano P. M. Giglio

Centro de Ensino Superior de Juiz de Fora (CES-JF)
Juiz de Fora – MG – Brasil

{lfaugusto.gomes,giucontato}@gmail.com

Abstract: *TUS is a recent protocol, open source, bringing the feature of resuming interrupted uploads, being those interruptions purposeful or not. There are implementations of TUS in server-side languages, and the purpose of this article is to provide a benchmark between those implementations wrote in the languages Node.js, Java and Go, showing the differences, strengths, weaknesses and usability.*

Resumo: *O TUS é um protocolo recente, open source, que oferece o recurso de retomar uploads interrompidos, sejam estas interrupções propositais ou não. Existem implementações do TUS em linguagens server-side, e o objetivo deste artigo é prover um benchmark (comparativo) entre estas implementações nas linguagens Node.js, Java e Go, apresentando com detalhes as diferenças, pontos fortes e fracos e usabilidade.*

1. Introdução

A gestão de documentos digitais é uma área que demanda bastante da Tecnologia da Informação, fundamental em diversos contextos, sejam coorporativos, redes sociais, ou para uso pessoal. Ferramentas como *Dropbox* e *Google Drive* são importantes neste contexto para compartilhar estes documentos, apesar de não trazerem o conceito de gestão embutidos ficando presos na velha estrutura hierárquica de diretórios e arquivos. Entre os diversos tipos de compartilhamento de arquivos, o foco deste artigo será o compartilhamento de arquivos através da *Internet*, princípio de todas estas ferramentas de compartilhamento de arquivos digitais.

A *Internet* é utilizada por diversos dispositivos como celulares, *tablets*, *desktops*, *notebooks*, entre outros, para diferentes fins. Com o advento das redes sociais, tais como *Facebook*, *Twitter*, *Instagram*, entre outras, o envio de mídias como fotos e vídeos é massivo, e grande parte deste envio é feita através dos dispositivos móveis.

De acordo com Lowe (2016), cerca de 85 milhões de fotos e vídeos são enviados para o *Instagram* diariamente, sendo que no *Facebook* este contingente atinge 350 milhões.

Considerando não apenas as redes sociais supracitadas, mas também outros *websites* cujo envio de arquivos é fundamental, este envio é comumente realizado através do protocolo *HTTP*, o qual é a base para transferência de dados através da *Internet*. Da forma mais básica de envio, até às mais avançadas, a preocupação com o

processo de envio dos arquivos visando poupar o uso da *Internet*, bem como reagir a problemas com este uso, é um assunto plausível e aberto no cenário atual. Este artigo apresentará dados e implementações de um novo protocolo: o *TUS – Open Protocol for Resumable File Uploads*.

O *TUS* [Tus 2016] é um protocolo recente, *open source*, que oferece o recurso de retomar *uploads* interrompidos, sejam estas interrupções propositais, ou seja, efetuadas pelo próprio usuário através do recurso de pausa, ou não propositais, no caso de possíveis problemas de performance na rede, equipamentos, entre outras falhas. Existem implementações do *TUS* em linguagens *server-side* [Tus Implementations 2016], e o objetivo deste artigo é prover um *benchmark* entre estas implementações nas linguagens *Node.js*, *Java* e *Go*, apresentando com detalhes as diferenças, pontos fortes e fracos e usabilidade.

As unidades seguintes deste artigo são referentes ao cenário atual de *uploads* na *WEB*, destacando protocolos utilizados, bem como componentes e técnicas diferentes de *upload*. Será apresentado com detalhes o protocolo *TUS*, quais são suas características, suas implementações disponíveis tanto no lado cliente quanto no lado servidor, fazendo, em seguida, um *benchmark* (comparativo) entre as linguagens *Node.js*, *Java* e *Go*. A conclusão e as considerações referentes ao protocolo e aos resultados obtidos no *benchmark* encerram o artigo, deixando sugestões para possíveis trabalhos futuros.

2. O Cenário Atual de *Uploads* na *WEB*

O compartilhamento de arquivos na *WEB* acontece graças ao protocolo *HTTP* (camada de aplicação), que por sua vez utiliza o protocolo *TCP/IP* (camada de transporte). Existem maneiras diferentes de se realizar *uploads* pela *WEB*, através do *HTML* simples, ou *Javascript/AJAX*, os quais serão detalhados a seguir.

Embora o foco deste artigo seja o protocolo *TUS*, que por sua vez baseia-se no protocolo *HTTP*, existem outros protocolos que focam no compatilhamento de arquivos através da *Internet*, como o *FTP* e o *Bittorrent*, e ambos serão comentados ao decorrer do tópico 2.

2.1. O Protocolo *FTP* (*File Transfer Protocol*)

O *FTP* é um protocolo padrão de compartilhamento de arquivos na *Internet* construído com base na arquitetura Cliente/Servidor [Wikipedia 2017]. A ideia é que se tenha um servidor *FTP* o qual armazena os arquivos do usuário, podendo este usuário fazer o *download* destes arquivos a qualquer momento. O *FTP* requer uma autenticação para que o acesso seja concebido, e normalmente trata-e de *login* e senha, mas também pode permitir acesso anônimo se o servidor estiver configurado para tal.

Vale ressaltar que existe também o protocolo *SFTP (SSH File Transfer Protocol* ou *Secure File Transfer Protocol*), o qual consiste em um pacote separado com a adição do *SSH* e permite o envio e *download* de arquivos através de conexões seguras [Ellingwood 2017].

Segundo Joan (2017), o protocolo *FTP* é menos popular em relação ao *HTTP* devido ao número de pessoas que o utilizam, e muitas pessoas que o utilizam não fazem ideia que estão o utilizando. Além disso, ele tende a ser subtituído ao longo do tempo devido à grande popularidade do protocolo *HTTP*.

2.2. O Protocolo *Bittorrent*

A principal ideia do protocolo *Bittorrent*, segundo Appel (2017), é que os clientes deste protocolo possam saber de onde buscar partes de determinado arquivo em diversos computadores, conhecidos como *peers*, ao invés de fazer o *download* do arquivo de uma única central, ou seja, um único servidor.

O protocolo *HTTP* [Appel 2017] representa um arquivo em um único *stream* através do servidor. A transferência é realizada de modo mais rápido possível, tanto da parte do servidor quanto do cliente e da rede em si. Logo após a requisição, o *HTTP* alcança a velocidade máxima muito rapidamente, sendo que a transferência é diretamente influenciada pela carga de requisições presente neste servidor.

Com o *Bittorrent*, o qual é baseado na arquitetura *P2P* (*peer to peer*), o cliente faz requisições aleatórias de partes do arquivo a ser transferido. Ele começa de forma lenta vai alcançando a velocidade máxima aos poucos. E, feito o *download*, o cliente pode começar a disponibilizar partes do arquivo para que outros usuários que buscam por este mesmo arquivo possam fazer o *download*.

2.3. *Uploads* com *HTML*

O *HTML* (*HypertText Markup Language*) contempla por si só o envio de arquivos através do protocolo *HTTP*. Basicamente, através da *tag input*, configurada com o tipo *file*, é possível fazer o envio de arquivos anexados na página *WEB* para que sejam recepcionados e processados pelas linguagens *server-side*. A mecânica é simples: o formulário deve estar configurado para enviar os dados pelo método *POST*, contendo o atributo *enctype* com o valor *multipart/form-data*. Os dados do arquivo são enviados de forma binária e tratados pela linguagem utilizada no servidor.

Existem diversos componentes, como o *Mini Ajax File Upload* e o *Dropzone Js* [Kingsley 2016], por exemplo, que customizam a *API* de envio de arquivos do *HTML* de forma a deixá-la mais intuitiva para o usuário final, e estes componentes vão fazer o uso de mecânicas diferentes para que o envio seja efetuado.

2.4. *Uploads* com *AJAX (Asynchronous Javascript and XML)*

É possível fazer o *upload* de arquivos utilizando a tecnologia *AJAX*. O envio também é feito via *POST*, e o arquivo também é enviado de forma binária. Porém, neste caso, o arquivo é tratado pela *API FormData* do *Javascript* [West 2017], além de ser possível realizar este envio de forma assíncrona, que é o cenário ideal para diversos casos de uso. Bibliotecas *Javascript* como o *jQuery* possuem uma *API* simples para utilização do *AJAX*, e de fato estará facilitando este tipo de implementação.

2.5. Exemplos de Componentes Disponíveis

Existe uma série de componentes criados para incrementar o recurso de *upload* com *HTML* e *Javascript*. A seguir serão apresentados dois destes componentes [Kingsley 2016], com o intuito de contextualizar os recursos que estão disponíveis atualmente na *WEB* para complementar o compartilhamento de arquivos.

- ***Mini Ajax File Upload:*** componente eficiente para *uploads* simples. Ele contempla envio múltiplo e simultâneo, com recurso de *Drag and Drop* (arrastar

e soltar) e barra de progresso simples e clara, podendo o usuário cancelar o *upload* a qualquer momento durante o progresso.

- ***Dropzone Js***: biblioteca *jQuery* muito bem documentada e leve para a performance dos *uploads*. Ela contempla *design* responsivo, trazendo as miniaturas dos arquivos anexados e suportando múltiplos envios com opção de remover anexos.

2.6. O Problema de *Uploads* de Grandes Arquivos

Um agravante que a *WEB* enfrenta atualmente [About Tus 2016] é o cenário da maioria das aplicações que contêm *uploads* de arquivos grandes que, em casos de problemas de conexão, perde-se tais operações forçando o usuário a reiniciá-los por completo, além de consumir banda considerável de *Internet*. Considerando o cenário *mobile*, onde as câmeras estão cada vez mais evoluídas em resolução e qualidade, o *upload* de imagens e principalmente vídeos tende a ser um desafio atentando ao fato de que se por qualquer motivo o *upload* for interrompido, o usuário precisa refazer o envio por completo.

Redes móveis, principalmente no Brasil, adicionam fragilidade e vulnerabilidade a este problema. Interrupções vão acontecer quando os usuários se encontram em lugares remotos que, por sua vez, dificultam a captação do sinal da *Internet*. Há também o fato de se desconectar de uma rede *Wi-fi*, onde o dispositivo automaticamente utilizará a conexão móvel caso ela esteja habilitada e com disponibilidade para o uso. Em ambos os casos é comum que haja a desconexão, mesmo que temporária, e esta perda de conexão comprometerá o *upload* em andamento.

Segundo Creative Js (2016), *uploads* utilizando *HTML* nativo sempre deixou a desejar para os desenvolvedores. Com isso, tem-se um conceito denominado *chunked upload*. Trata-se de uma técnica de "quebrar" o arquivo a ser enviado em pequenos *chunks*, ou seja, pequenos pedaços para que o envio de cada um deles possa ser realizado, mesmo sendo apenas um arquivo enviado. A grande vantagem de tal utilização é a possibilidade de dar continuidade ao *upload* caso algum evento negativo venha a ocorrer, sendo este evento a queda da conexão com a *Internet*, por exemplo. Como o envio é feito *chunk* por *chunk*, o usuário não necessita refazer o *upload* desde o princípio. Com isso, surge um novo cenário: *pause* e *resume*. Através deste recurso, no caso do envio de múltiplos arquivos, pode-se dar prioridade para determinados arquivos pausando os demais, e tudo isso pode ser efetuado manualmente pelo próprio usuário.

Muitas empresas como a *Google*, *Dropbox*, *Vimeo* e *Amazon*, por exemplo, possuem suas próprias soluções para o envio de partes de um arquivo. Porém, a maioria destas implementações está atrelada a uma linguagem de programação específica ou a algum caso de uso específico [Tus Blog 2016]. Algumas implementações são compatíveis entre si, porém isso não compõe o conceito de compatibilidade ao nível de um protocolo.

A seguir serão apresentados dois exemplos de *APIs* que contemplam o conceito de *upload* por partes.

- ***Amazon Web Services – multipart upload:*** a *Amazon* atualmente contempla o recurso de se fazer *uploads* de arquivos quebrando-os em partes [Amazon Web Services 2016]. Se o *upload* de uma ou mais partes falhar, estas partes podem ser enviadas novamente sem comprometer as demais partes cujo *upload* tenha

sido concluído. Existem algumas restrições da *AWS* para esta funcionalidade [Tus Blog 2016], como por exemplo, o tamanho mínimo de *5MB* para cada parte do objeto (com exceção do último *chunk,* o qual pode ser menor).

- **API *Filesystems* do *HTML5:*** com a *API FileSystems* é possível lidar com arquivos de forma completa: criar, modificar, excluir, lidar com diretórios, entre outras características [Bidelman 2016]. No caso do *upload*, quando um arquivo ou diretório é selecionado para o envio, ele copia este arquivo para uma área local reservada, quebra o arquivo em *chunks* e envia um *chunk* por vez. E, de fato, possibilita que o recurso de *pause* e *resume* seja implementado.

As *APIs* estão ficando desorganizadas neste cenário considerando que em diversos projetos tem-se implementado componentes próprios de *upload*, e em muitos casos há o problema de incompatibilidade. O protocolo *TUS* vem para atacar este problema de forma satisfatória, padronizando as implementações nas aplicações [About Tus 2016].

3. O Protocolo *TUS*

O *TUS* é um protocolo de código aberto criado e mantido por uma comunidade para atender à necessidade de poder-se retomar *uploads* de arquivos [Tus 2016]. Diversas interrupções podem ocorrer ao longo de um *upload*, podendo ser a queda da conexão com a *Internet*, ou até mesmo a intervenção do usuário cancelando este *upload*. E, ao tratar-se de *Internet*, vale ressaltar que redes móveis hoje em dia possuem certa limitação que pode (e em alguns casos irá) comprometer o compartilhamento de arquivos, como visto anteriormente. Esta limitação pode estar relacionada ao sinal da rede móvel que, dependendo do local, pode ser de difícil captação; pode estar relacionada à banda de *Internet* do usuário que, por sua vez, venha estar escassa e/ou próxima do limite de uso. Por outro lado, em uma visão mais abrangente, a falta de bateria do dispositivo móvel ou qualquer outro comprometimento físico pode interromper o *upload*. Até mesmo em redes de computadores pode haver interrupções, como a queda de energia, falha nos servidores, entre outras.

Sendo assim, um protocolo bem definido para centralizar a funcionalidade de se retomar um *upload* a partir do ponto em que este *upload* foi interrompido torna-se uma ferramenta essencial para o cenário contemporâneo de compartilhamento de arquivos. O *TUS* ataca diretamente este problema de interrupções, permitindo que um *upload* interrompido possa ser retomado, e já está sendo usado por grupos como *Transloadit*, *Vimeo*, entre outros [Tus 2016].

Nos tópicos seguintes serão detalhadas características da versão *1.0.0*, a qual é a versão atual, tanto do protocolo em si quanto das implementações, sejam estas clientes ou servidores.

3.1. Características

O protocolo foi construído fazendo o papel de uma camada do protocolo *HTTP*, e por isso pode ser facilmente integrado a bibliotecas, *firewalls* e *proxies*. Além disso, pode ser usado diretamente de qualquer *website* [Tus 2016]. O protocolo está pronto para ser usado em produção e, por ser de código aberto, é frequentemente melhorado e a

comunidade recebe *feedbacks* significativos de colaboradores de grandes entidades, como *Google* e *Vimeo*, por exemplo.

A especificação do protocolo requer um conjunto pequeno de características para que ele possa ser implementado por clientes e servidores, além de possuir uma série de extensões que estarão contemplando funcionalidades extras.

Por se tratar de um protocolo, as regras estabelecidas pelo *TUS* devem ser respeitadas pelos clientes e pelos servidores de forma a se ter uma solução genérica durante o compartilhamento dos arquivos. O *TUS* possui uma estrutura bem definida, com cabeçalhos e tipos diferentes de requisições, onde todo este cenário baseia-se no protocolo *HTTP*.

No entanto, o protocolo *HTTPS* ainda não é diretamente suportado pelo *TUS*. Isso foi definido assim para que se pudesse limitar a quantidade de recursos contida no repositório, as quais tem de ser desenvolvidas, testadas e mantidas [Tusd 2017]. Sendo assim, para que se possa enviar requisições seguras, deve-se usar um *proxy* reverso que aceite requisições *HTTPS* e redirecione estas requisições para o servidor *TUS* utilizando *HTTP* simples.

3.2. *Core Protocol*

O *Core Protocol* descreve como se deve proceder ao retormar-se um *upload* interrompido. Ele assume que a implementação já possui uma *URL* para o *upload*. Todos os clientes e servidores devem implementar o *Core Protocol*.

A Figura 1 e a Figura 2 fazem uma pequena demonstração da comunicação entre um cliente e um servidor, ambos implementando o *TUS*. No cenário deste exemplo, um arquivo de *100 bytes* foi enviado, e seu envio foi interrompido quando 70% do arquivo foram enviados com sucesso.

Request:

```
HEAD /files/24e533e02ec3bc40c387f1a0e460e216 HTTP/1.1
Host: tus.example.org
Tus-Resumable: 1.0.0
```

Response:

```
HTTP/1.1 200 OK
Upload-Offset: 70
Tus-Resumable: 1.0.0
```

Figura 1. Exemplo de um *upload* interrompido aos 70%

Na Figura 1, o arquivo de *100 bytes* teve seu envio iniciado pelo cliente e foi interrompido aos 70%. O servidor, por sua vez, retorna uma resposta indicando quantos *bytes* do arquivo foram enviados (*offset*) através do cabeçalho *Upload-Offset*.

Request:

```
PATCH /files/24e533e02ec3bc40c387f1a0e460e216 HTTP/1.1
Host: tus.example.org
Content-Type: application/offset+octet-stream
Content-Length: 30
Upload-Offset: 70
Tus-Resumable: 1.0.0

[remaining 30 bytes]
```

Response:

```
HTTP/1.1 204 No Content
Tus-Resumable: 1.0.0
Upload-Offset: 100
```

Figura 2. Retomando o *upload* que havia sido interrompido aos 70%

Na Figura 2, o cliente faz uma nova requisição ao servidor, desta vez utilizando o método *PATCH* do protocolo *HTTP*, para que se possa retomar o *upload* do arquivo enviando os 30% restantes. O servidor devolve uma nova resposta, indicando que o envio foi concluído.

3.3. Cabeçalhos

Existe uma série de cabeçalhos [Tus Protocol 2016] que estarão presentes nas requisições e respostas entre um cliente e um servidor que estejam implementando o *TUS*. Cada cabeçalho possui uma característica única, podendo eles ser obrigatórios dependendo do tipo de requisição. A Tabela 1 traz uma relação dos principais cabeçalhos do *TUS*.

Tabela 1. Relação de cabeçalhos do protocolo *TUS*

Cabeçalho	Tipo	Definição
Upload-offset	Requisição e Resposta	Indica um deslocamento (*offset*) de *bytes*, ou seja, partes que já foram enviadas.
Upload-Length	Requisição e Resposta	Indica o tamanho em *bytes* do arquivo que está sendo enviado
Tus-version	Resposta	Lista de versões do *TUS* suportadas pelo servidor
Tus-resumable	Requisição e Resposta	Contempla a versão do protocolo utilizada pelo cliente e pelo servidor. Em caso de discrepância, o servidor devolve o *status 412* (falha na pré-condição)
Tus-extension	Resposta	Lista de extensões suportadas pelo servidor
Tus-max-size	Resposta	Tamanho máximo em *bytes* permitido para cada arquivo enviado

3.4. Requisições

Existem diferentes requisições [Tus Protocol 2016] que podem ser feitas pelos clientes durante o *upload* utilizando o *TUS*. Cada uma tem um propósito, e todas elas são baseadas no protocolo *HTTP*. A Tabela 2 traz uma relação das principais requisições presentes no *TUS*.

Tabela 2. Relação de requisições presentes no protocolo *TUS*

Requisição	Características
HEAD	Servidores devem incluir o *upload-offset* nas respostas a esta requisição. Se o tamanho do *upload* for conhecido, o *upload-length* também deve estar incluído na resposta. O *status 404* (não encontrado), *410* (*gone*) ou *403* (proibido) devem ser retornados em casos de requisições *HEAD* a *uploads* não encontrados
PATCH	O servidor deve aceitar requisições *PATCH* diante de qualquer *URL* e aplicar os *bytes* contidos no *offset*. O servidor deve notificar o sucesso da requisição através do *status 204* (nenhum conteúdo). É através desta requisição que os clientes enviarão o restante do arquivo cujo *upload* foi interrompido
OPTIONS	É através desta requisição que o cliente obtém informações a respeito da configuração atual do servidor. A resposta deve conter o *status 204* (nenhum conteúdo) ou *200* (ok), contendo os cabeçalhos *tus-version*, *tus-extensions* e *tus-max-size*

3.5. Extensões

O *TUS* possui várias extensões que podem ser implementadas pelo cliente e pelo servidor [Tus Protocol 2016]. Estas extensões trazem funcionalidades extras ao protocolo. A Tabela 3 traz uma relação das principais extensões do *TUS*.

Tabela 3. Relação de extensões do protocolo *TUS*

Extensão	Cabeçalhos extras	Requisições extras	Características
Creation	*Upload-defer-length* *Upload-metadata*	*POST*	Responsável por criar um novo recurso de *upload*. Ambos cliente e servidor devem implementá-la
Expiration	*Upload-expires*	-	É através desta extensão que o servidor pode remover *uploads* expirados. Existe um valor no formato de data que indica quando determinado *upload* deve expirar

Checksum	*Tus-checksum-algorithm* *Upload-checksum*	-	Através desta extensão, clientes e servidores podem verificar a integridade das informações trafegadas utilizando os algoritmos especificados
Termination	-	*DELETE*	Permite o encerramento de *uploads* completados ou *uploads* não terminados visando liberar recursos no servidor
Concatenation	*Upload-concat*	-	Esta extensão permite a concatenação de múltiplos *uploads* tornando-os um *upload* único, fazendo com que clientes possam efetuar *uploads* em paralelo

4. *TUS* e as Linguagens *Server-side*

Estão disponíveis diversas implementações *server-side* do protocolo *TUS*, tanto para a versão atual *1.0.0* quanto para versões anteriores, sendo que todas as implementações estão sob a licença *MIT (Massachusetts Institute of Technology)*[1]. Mais informações em Tus Implementations (2016).

O foco será sobre as implentações nas linguagens *Node.js*[2], *Java*[3] e *Go*[4]. Todas estas implementações são multi-plataforma, ou seja, independem de *hardware* ou *software* para que sejam implementadas.

4.1. *Tus-node-server*

O *tus-node-server* é a versão *server-side* do *Node.js* para a implementação do *TUS, e* está disponível no *GitHub* [Tus-node-server 2017] contemplando a versão *1.0.0* do protocolo. Esta versão possui flexibilidade de armazenamento de arquivos, possibilitando o armazenamento local, armazenamento no *Google Cloud* e existe a intenção de se disponibilizar o armazenamento direto no *Storage* da *Amazon (S3)*. Esta é uma das implementações oficiais do protocolo, ou seja, desenvolvida pela própria comunidade, e pode ser instalada utilizando-se o gerenciador de pacotes *Node Package Manager - NPM* [Node Package Manager 2017].

4.2. *Tus-servlet*

O t*us-servlet* é a versão *server-side* do *TUS* em *Java,* e está disponível no *GitHub* [Tus-servlet 2017] contemplando versão *1.0.0* do protocolo. Trata-se de um *Servlet Java*, o

[1] Licença que permite a reutilização de *software* licenciado em programas livres ou proprietários
[2] Mais informações sobre a linguagem *Node.js* em https://nodejs.org/en/
[3] Mais informações sobre a linguagem *Java* em https://www.java.com/en/
[4] Mais informações sobre a linguagem *Go* em https://golang.org/

qual pode ser instalado utilizando o gerenciador de pacotes e dependências do *Java*: o *Maven* [Maven, 2017]. Vale ressaltar que *tus-servlet* não suporta requisições *GET*, e, portanto, não suporta *download* de arquivos de forma direta após a conclusão do *upload*. Por padrão, ela disponibiliza o serviço do *TUS* na porta *8080*.

4.3. *Tusd*

O *tusd* é a versão *server-side* do *TUS* em *Go*, sendo a versão oficial implementada pela própria comunidade. Esta versão foi construída para ser flexível e permitir o uso de diferentes mecanismos de armazenamento de arquivos. Por padrão, o *tusd* traz consigo o recurso de armazenamento em um diretório no próprio servidor, e está disponível no *GitHub* [Tusd 2017].

No caso de usuários possuírem diferentes necessidades, diversos tipos de armazenamento podem ser configurados para que estas necessidades sejam atendidas. Pode ser feito o armazenamento direto no servidor, bem como no *Storage* da *Amazon* (*S3*), ou em algum servidor *FTP* remoto, entre outros.

O *Tusd* também possibilita verificação e autenticação de dados enviados através de um sistema de eventos. Isso possibilita a execução de rotinas customizadas durante a criação de um *upload*, por exemplo. Os metadados do arquivo podem ser validados e rotinas específicas podem ser executadas de forma com que o *Tusd* possa rejeitar ou aceitar aquela requisição.

5. Clientes de Implementação do *TUS*

Existe uma série de clientes disponíveis sob a licença *MIT* (*Massachusetts Institute of Technology*) que implementam o *TUS*. Estes clientes são *open source*, ou seja, possuem código aberto, e estarão contemplando versões diferentes do *TUS*. Mais informações sobre a diversificação destes clientes em Tus Implementations (2016). A seguir tem-se dois exemplos destes clientes.

5.1. *Uppy*

O *Uppy* é um *uploader Javascript open source* criado pela *Transloadit* que está em forte desenvolvimento. A ideia é permitir o usuário anexar arquivos de diversas fontes, ou seja, não apenas do dispositivo local, mas de fontes externas como *Google Drive*, *Dropbox*, *Instagram*, e até mesmo *URLs* remotas [Uppy 2017]. Ele é construído à base de *plugins*, totalmente modular, fazendo com que seja leve e customizável. A Figura 3 traz um exemplo de código *Javascript* de configuração básica para a utilização do *Uppy*.

```
var Uppy = require('uppy')

var uppy = new Uppy.Core({wait: false})
var files = uppy
  .use(Uppy.DragDrop, {target: 'body'})
  .use(Uppy.Tus10, {endpoint: '//tusd.tus.io/files'})
  .run()
```

Figura 3. Trecho de código de configuração do *Uppy*

Devido à fácil instalação, flexibilidade e usabilidade, o *Uppy* foi escolhido para ser o cliente na execução do *benchmark* deste artigo.

5.2 *Tus-js-client*

O *tus-js-client* é um cliente desenvolvido em *Javascript* puro com suporte em diversos navegadores *WEB*. Seu uso é simples, e ele suporta a implementação de várias extensões do *TUS*. Ele conta com diversos recursos de configuração e eventos que estarão auxiliando os desenvolvedores, e se encontra disponível no *GitHub* (Tus-js-client 2017). A Figura 4 traz a interface gráfica exibida no navegador *WEB* do *tus-js-client*.

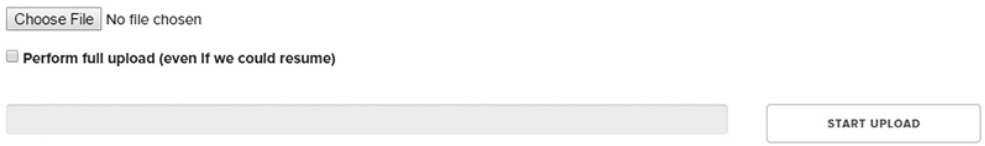


Figura 4. Interface gráfica do *Tus-js-client*

6. Estratégia de *Benchmark*

A seguir será apresentado o resultado de um *benchmark* entre as implementações do *TUS* nas linguagens *Node.js*, *Java* e *Go*, sendo todas elas referentes à versão *1.0.0* do protocolo, ou seja, a versão atual. A escolha das linguagens *Node.js* e *Go* se deu pelo fato de serem implementações da própria comunidade, onde a versão *Go* é a versão oficial. A escolha do *Java* se deu pela consolidação da linguagem no mercado de Tecnologia da Informação.

A elaboração da estratégia de *benchmark* consistiu de várias etapas, desde a escolha e preparação do servidor de hospedagem, até os monitoramentos finais através de ferramentas específicas, as quais são detalhadas no item 6.1. Este tópico trará todos os detalhes referentes a este cenário. A ideia consistiu em colocar as implementações do *TUS* em prática, processando dois cenários:

- O *upload* de um arquivo de vídeo de *22,5MB*, com duração de 4 minutos 48 segundos no formato *MP4*;
- O *upload* múltiplo simultâneo de seis arquivos de imagem no formato *JPG*, pesando respectivamente, *2,979MB*, *3,928MB*, *3,324MB*, *2,348MB*, *2,130MB* e *1,991MB*.

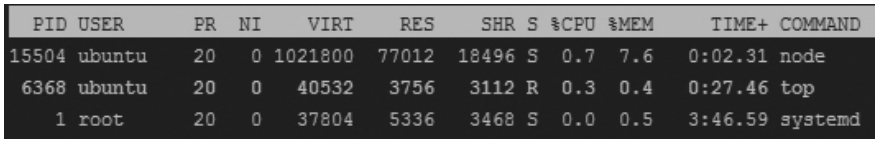
Durante a execução de cada um destes dois cenários, foram observados o uso de memória física e processamento do servidor de hospedagem, bem como o tempo que cada implementação precisou para processar as requisições feitas pelo cliente, ou seja, foram calculados os tempos entre a chegada de cada requisição e a saída da respectiva resposta do servidor.

A rede de *10mbps* utilizada para realizar os *uploads* foi totalmente desconsiderada para a realização do *benchmark*. Isso deve-se ao fato de que o objetivo consistiu em fazer uma comparação entre a performace de cada linguagem, ou seja, o tempo e recursos utilizados pelos servidores à medida que os *uploads* estavam sendo executados. O cliente utilizado para todos os casos foi o *Uppy* (item 5.1).

6.1. Ferramentas Utilizadas

Para fazer a medição do consumo de memória e processador, bem como o tempo em segundos que o servidor precisou para responder cada requisição, foram utilizadas duas ferramentas:

- O gerenciador de tarefas do *Ubuntu*, acionado através do comando *top*. Com este comando é possível analizar, através do *terminal* do *Ubuntu*, todas as tarefas que estão sendo executadas na memória do servidor de hospedagem, juntamente com o consumo de cada tarefa em relação à memória e ao processamento. A Figura 5 exibe tarefas em execução durante a análise de performance da implementação do *Node.js*.



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
15504	ubuntu	20	0	1021800	77012	18496	S	0.7	7.6	0:02.31	node
6368	ubuntu	20	0	40532	3756	3112	R	0.3	0.4	0:27.46	top
1	root	20	0	37804	5336	3468	S	0.0	0.5	3:46.59	systemd

Figura 5. Exemplo do uso do comando *top* do *Ubuntu*

  Como pode ser observado na Figura 5, o processo do *Node.js* (*node*) consumiu 0,7% do processador e 7,6% de memória *RAM*.

- O *software justniffer*, responsável por capturar pacotes trafegados na rede através de diversos protocolos, como *HTTP*, *SSL*, o próprio *TCP*, podendo monitorar portas específicas e trazer dados parametrizados, tais como cabeçalhos *HTTP* das requisições e respostas (Justniffer 2017). O uso do *justniffer* deu-se através da *IDE PHPStorm*, acessando o servidor de hospedagem via *SSH*.

6.2. Hospedagem

Para a hospedagem das implementações foram utilizados os serviços da *Amazon Web Services (AWS)*. Através de uma micro instância no *EC2* (*Elastic Compute Cloud*) foi criado um ambiente *Linux* com as seguintes configurações:

- Sistema Operacional *Ubuntu Xenial v. 16.04*;
- Processador *Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz*;
- Memória *RAM* de *1GB*;
- Disco Rígido de *8GB*.

Com o sistema operacional preparado, o servidor *Apache* foi configurado para atender às requisições dos navegadores *WEB*, onde foram utilizadas *URLs* e portas diferentes, cada uma redirecionando para sua respectiva implementação. Todas as implementações foram instaladas no servidor de hospedagem a partir do *GitHub*. O tópico seguinte traz detalhes de cada implementação.

6.3. Implementação *Node.js*

Para o cenário do *Node.js*, foram instalados os seguintes itens no servidor de hospedagem:

- A linguagem *Node.js*;
- O *Node Package Manager – NPM* [Node Package Manager 2017];
- O *tus-node-server* (item 4.1), configurado para atender à porta *8000*.

6.4. Implementação *Java*

Para o cenário do *Java*, foram instalados os seguintes itens no servidor de hospedagem:

- O *java default-jre*;
- O *java default-jdk*;
- O *java openjdk-7-jre*;
- O *maven* [Maven 2017];
- O *tus-servlet* (item 4.2), configurado para atender à porta *8080*.

6.5. Implementação *Go*

Para o cenário do *Go*, foram instalados os seguintes itens no servidor de hospedagem:

- O pacote *golang-go*;
- A linguagem *go* versão *1.6*;
- O *tusd* (item 4.3), configurado para atender à porta *1080*.

Para cada implementação, a Tabela 4 exibe a disponibilidade para navegadores *WEB* e onde o *upload* foi configurado para disparar os dados para os respectivos endereços.

Tabela 4. Endereços das implementações e *uploads*

Implementação	Endereço para Navegadores	Endereço do *Upload*
Node.js	*http://devlf.dynu.net/node*	*http://devlf.dynu.net:8000/files*
Java	*http://devlf.dynu.net/java*	*http://devlf.dynu.net:8080/files*
Go	*http://devlf.dynu.net/go*	*http://devlf.dynu.net:1080/files*

7. Análise Comparativa dos Resultados Obtidos

Os testes foram realizados primeiramente com o *Node.js*, seguido do *Java* e por último o *Go*. Para cada caso, o consumo de memória durante a inicialização do servidor foi catalogado, bem como o consumo de processador e o tempo de execução durante cada *upload*.

7.1. Resultados do *Node.js*

Durante a inicialização do servidor *Node.js*, realizada através do comando *npm run demo* no terminal do *Ubuntu*, o *Node* obteve um pico de aproximadamente *86,18MB* de memória (equivalente a 8,7% do total), consumindo cerca de 31,9% do processador. O tempo decorrido para inicializar o servidor foi 1,81 segundos. A Figura 6 mostra este pico obtido através do gerenciador de tarefas do *Ubuntu*.

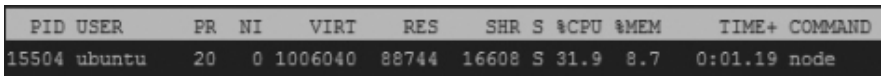


Figura 6. Consumo do *Node.js* durante a inicialização do servidor

Durante o *upload* do arquivo de vídeo de *22,5MB*, o *Node* obteve um pico de aproximadamente *75,28MB* de memória (7.6% do total), utilizando 0,7% do processador. O *Node* precisou, no total, de 0,013635 segundos para processar todas as requisições.

Durante o *upload* das seis imagens (total de *16,58MB*), o *Node* obteve um pico de aproximadamente *76,27MB* de memória (total de 7.7% do total), utilizando 2% do processador. O *Node* precisou, no total, de 0,068056 segundos para processar todas as requisições. A Figura 7 exibe o progresso do *upload* das seis imagens.

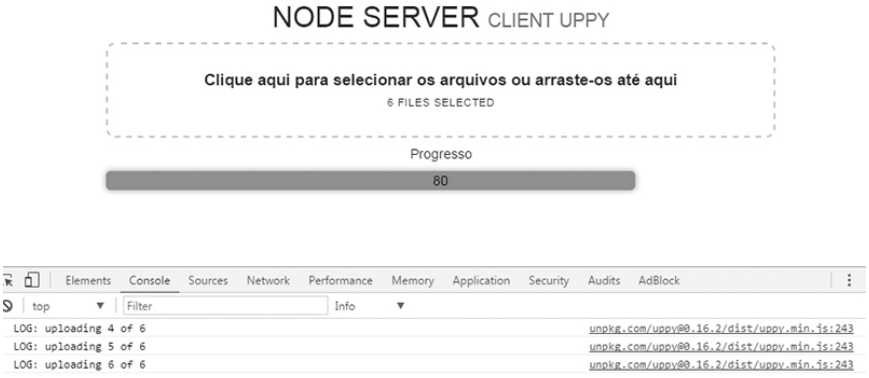


Figura 7. *Upload* das seis imagens para o servidor *Node*

A Tabela 5 traz os resultados obtidos pelo *tus-node-server*.

Tabela 5. Resultados *tus-node-server*

Tus-node-server	Uso da Memória (*MB*)	Uso do Processador (%)	Tempo de execução (segundos)
Inicialização do servidor	86,18 (8,7%)	31,9	1,81
Upload do vídeo de *22,5MB*	75,28 (7,6%)	0,7	0,013635
Upload das seis imagens (*16,58MB*)	76,27 (7,7%)	2	0,068056

7.2. Resultados do *Java*

Durante a inicialização do servidor *Java*, realizada com o comando *mvn jetty:run* no terminal do *Ubuntu*, o *Java* obteve um pico de aproximadamente *124,82MB* de memória (equivalente a 12,6% do total), consumindo cerca de 44,5% do processador. O tempo decorrido para inicializar o servidor foi 6,37 segundos. A Figura 8 mostra este pico obtido através do gerenciador de tarefas do *Ubuntu*.

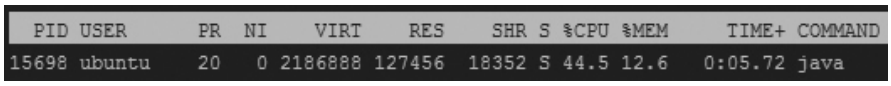


Figura 8. Consumo do *Java* durante a inicialização do servidor

Durante o *upload* do arquivo de vídeo de *22,5MB*, o *Java* obteve um pico de aproximadamente *131,75MB* de memória (13,3% do total), utilizando em média 2,6% do processador. O *Java* precisou, no total, de 0,555089 segundos para processar todas as requisições.

Durante o *upload* das seis imagens, o *Java* obteve um pico de aproximadamente *131,75MB* de memória (total de 13.3% do total), utilizando em média 1,5% do processador. O *Java* precisou, no total, de 0,127464 segundos para processar todas as requisições. A Tabela 6 traz os resultados obtidos pelo *tus-servlet*.

Tabela 6. Resultados *tus-servlet*

Tus-servlet	Uso da Memória (*MB*)	Uso do Processador (%)	Tempo de execução (segundos)
Inicialização do servidor	124,82 (12,6%)	44,5	6,37
Upload do vídeo de *22,5MB*	131,75 (13,3%)	2,6	0,555089
Upload das seis imagens (*16,58MB*)	131,75 (13,3%)	1,5	0,127464

7.3. Resultados do *Go*

Durante a inicialização do servidor *Go*, realizada com o comando *go run cmd/tusd/main.go* no terminal do *Ubuntu*, o *Go* obteve um pico de aproximadamente *13,86MB* de memória (equivalente a 1.4% do total), consumindo cerca de 3% do processador. O tempo decorrido para inicializar o servidor foi 2,52 segundos. A Figura 9 mostra este pico obtido através do gerenciador de tarefas do *Ubuntu*.

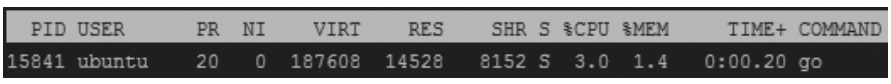


Figura 9. Consumo do *Go* durante a inicialização do servidor

Durante o *upload* do arquivo de vídeo de *22,5MB*, o *Go* obteve um pico de aproximadamente *9,9MB* de uso memória (1% do total), utilizando em média 0,5% do processador. O *Go* precisou, no total, de 0,001342 segundos para processar todas as requisições.

Durante o *upload* das seis imagens, o *Go* obteve um pico de aproximadamente *11,88MB* de memória (total de 1,2% do total), utilizando em média 1% do processador. O *Go* precisou, no total, de 0,003311 segundos para processar todas as requisições. A Tabela 7 traz os resultados obtidos pelo *tusd*.

Tabela 7. Resultados *tusd*

Tusd	Uso da Memória (*MB*)	Uso do Processador (%)	Tempo de execução (segundos)
Inicialização do servidor	13,86 (1,4%)	3	2,52
Upload do vídeo de *22,5MB*	9,9 (1%)	0,5	0,001342
Upload das seis imagens (*16,58MB*)	11,88 (1,2%)	1	0,003311

8. Conclusão e Considerações Finais

O *TUS* é um protocolo criado para que se possa atacar um problema no contexto atual de uso da Tecnologia da Informação, mais especificamente durante o compartilhamento de arquivos digitais. O protocolo permite que *uploads* possam ser retomados em caso de interrupções, e com isso pode-se poupar tempo e recursos de rede. Pode-se admitir que em diversos casos, principalmente nas redes móveis, as quedas e interrupções podem e vão ser um empecilho para os usuários. Sendo assim, possuir uma solução centralizada e flexível para ajudar na resolução deste cenário negativo é algo promissor. O *TUS* é uma tecnologia recente, se encontra pronto para ser utilizado em produção e tende a ser melhorado à medida que a comunidade, junto às grandes entidades como *Google*, continue contribuindo.

Na perspectiva técnica, embora *Java* e *Node.js* sejam duas linguagens *server-side* consolidadas, a implementação do *TUS* em *Go* obteve um maior destaque de uma forma geral. A utilização de memória *RAM* e de processamento apresentou-se mais balanceada, além de o aproveitamento de tempo ter sido muito superior. O uso de memória e processamento foi mais acentuado na versão *Java*, linguagem para qual normalmente os servidores de aplicação são mais exigentes quanto ao consumo de recursos. A versão do *Node.js* pode ser considerada um meio termo entre *Java* e *Go*, consumindo razoavelmente a memória e o processador, e apresentando um tempo de execução aceitável. A Tabela 8 traz um quadro comparando todas as três implementações, destacando a implementação de melhor desempenho em cada categoria testada.

Tabela 8. Quadro comparativo das linguagens testadas

	Uso da Memória (*MB*)	Uso do Processador (%)	Tempo de execução (segundos)

Inicialização do servidor	*Tus-node-server: 86,18 (8,7%)*	*Tus-node-server: 31,9*	***Tus-node-server: 1,81***
	Tus-servlet: 124,82 (12,6%)	*Tus-servlet: 44,5*	*Tus-servlet: 6,37*
	Tusd: 13,86 (1,4%)	***Tusd: 3***	*Tusd: 2,52*
Upload do vídeo de *22,5MB*	*Tus-node-server: 75,28 (7,6%)*	*Tus-node-server: 0,7*	*Tus-node-server: 0,013635*
	Tus-servlet: 131,75 (13,3%)	*Tus-servlet: 2,6*	*Tus-servlet: 0,555089*
	Tusd: 9,9 (1%)	***Tusd: 0,5***	***Tusd: 0,001342***
Upload das seis imagens (*16,58MB*)	*Tus-node-server: 76,27 (7,7%)*	*Tus-node-server: 2*	*Tus-node-server: 0,068056*
	Tus-servlet: 131,75 (13,3%)	*Tus-servlet: 1,5*	*Tus-servlet: 0,127464*
	Tusd: 11,88 (1,2%)	***Tusd: 1***	***Tusd: 0,003311***
Melhor desempenho: Tusd			

É importante ressaltar que, embora a implementação em *Go* tenha apresentado um desempenho melhor, os argumentos que pesam para a escolha de uma destas linguagens para contemplar o protocolo *TUS* podem variar de acordo com a necessidade e contexto dos desenvolvedores. Dependendo do perfil do desenvolvimento, adicionar uma linguagem *server-side* extra apenas para processar os *uploads* pode ser inviável considerando a organização do projeto como um todo.

Para trabalhos futuros, é válido implementar testes e *benchmarks* utilizando plataformas *mobile* e considerando, talvez, outras linguagens *server-side* com outros clientes do *TUS*.

Referências

About Tus (2016) “About tus”, http://tus.io/about.html

Amazon Web Services (2016) “Multipart Upload Overview”, http://docs.aws.amazon.com/AmazonS3/latest/dev/mpuoverview.html

Appel, A., Chapman, D. (2017) “Difference Between FTP and HTTP”, https://daniel.haxx.se/docs/bittorrent-vs-http.html

Bidelman, E. (2016) “Exploring the FileSystem APIs”, https://www.html5rocks.com/en/tutorials/file/filesystem

Creative Js (2016) “Advanced Uploading Techniques - Part 1”, http://creativejs.com/tutorials/advanced-uploading-techniques-part-1/index.html

Ellingwood, J. (2017) “How to Use SFTP to Securely Transfer Files with a Remote Server”, https://www.digitalocean.com/community/tutorials/how-to-use-sftp-to-securely-transfer-files-with-a-remote-server

Joan, B. (2017) “bittorrent vs HTTP”, http://www.differencebetween.net/technology/difference-between-ftp-and-http

Justniffer (2017) “Justniffer”, http://justniffer.sourceforge.net/#!/home

Kingsley, A. (2016) “14 Best HTML5 jQuery File Upload Scripts”, https://dcrazed.com/html5-jquery-file-upload-scripts

Lowe, L. (2016) “125 Amazing Social Media Statistic You Should Know in 2016”, https://socialpilot.co/blog/125-amazing-social-media-statistics-know-2016

Maven (2017) “Apache Maven Project”, http://maven.apache.org

Node Package Manager (2017) “Build Amazing Things”, https://www.npmjs.com

Tus (2016) “Open Protocol for Resumable File Uploads”, http://tus.io

Tus Blog (2016) “S3 as a Storage Back-End”, http://tus.io/blog.html

Tus Implementations (2016) “Implementations”, http://tus.io/implementations.html

Tus Protocol (2016) “Protocol”, http://tus.io/protocols/resumable-upload.html

Tus-js-client (2017) “A Pure Javascript Client for the Tus Resumable Upload Protocol”, https://github.com/tus/tus-js-client

Tus-node-server (2017) “Node.js Tus Server”, https://github.com/tus/tus-node-server

Tus-servlet (2017) “Java Servlet Implementing Server Side of Tus File Upload Protocol”, https://github.com/terrischwartz/tus_servlet

Tusd (2017) “The Official Server Implementation of the Tus Resumable Upload Protocol”, https://github.com/tus/tusd

Uppy (2017) “The Uppy Blog”, https://uppy.io/blog

West, M. (2017) “Uploading Files with Ajax”, http://blog.teamtreehouse.com/uploading-files-ajax

Wikipedia (2017) “File Transfer Protocol”, https://en.wikipedia.org/wiki/File_Transfer_Protocol