Instituto de Matemática e Estatística
Universidade de São Paulo

Trabalho de Formatura Supervisionado
Bacharelado em Ciência da Computação

Um Servidor HTTP/2 Reativo em Scala

Daniel Q. Miranda
Orientador: Prof. Dr. Daniel Macêdo Batista

São Paulo, Brasil, Janeiro 2016

Resumo

O protocolo de rede HTTP/2 foi padronizado em 2015, atualizando o atualmente mais utilizado método de comunicação entre aplicações na Internet. Esse trabalho descreve uma implementação de um servidor web HTTP/2 utilizando a linguagem de programação funcional Scala, aplicando as metodologia de programação reativa e orientada a fluxos, em conjunto com o modelo de concorrência de atores, visando produzir um sistema elegante, modular e eficiente.

Palavras-chave: HTTP/2, Scala, modelo de concorrência de atores, programação orientada a fluxos de dados, programação reativa, redes de computadores

Abstract

The HTTP/2 network protocol was made a standard in 2015, updating the currently most used communication method for Internet applications. This article describes an implementatin of an HTTP/2 web server in the Scala language, applying principles of reactive, data-flow oriented programming and the actor model of concurrency, with the goal of creating an elegant, modular and efficient system.

Keywords: HTTP/2, Scala, actor model, data-flow programming, reactive programming, network programming

Sumário

1 Introdução

O protocolo de comunicação de rede HTTP (HyperText Transfer Protocol) é o alicerce do funcionamento da Web - e por extensão, de grande parte da Internet - e define os mecanismos de comunicação utilizados por um enorme número de aplicações conectadas por redes de computadores. Foi criado como método de transmissão para documentos, mas hoje carrega todo tipo de mídia - como aplicações complexas, áudio e vídeo - devido à sua flexibilidade.

Em 2015 a Internet Engineering Task Force (IETF), grupo de maior influência na definição de padrões e práticas da Internet, propôs, após mais de 20 anos da original, uma nova versão modernizada do HTTP, denominada HTTP/2 ("Hypertext transfer protocol version 2 (HTTP/2)" 2015). Ela mantém a semântica original das mensagens, mas define um novo mecanismo de transmissão mais eficiente, com melhoras no aproveitamento de recursos, desempenho, extensibilidade e segurança.

Visando sanar essas deficiências o HTTP/2, dentre outras melhorias:

- Introduz novos mecanismos para explorar ao máximo recursos de rede disponíveis, em especial através do paralelismo de transmissões em uma única conexão TCP. (vide seção 2.4).
- Substitui uma representação textual, na qual delimitadores de conteúdo e definições são sequências especiais de caracteres, por uma binária, onde diversos tipos de mensagens são definidos com métodos de codificação individuais. Essa mudança permite a utilização de operações mais complexas, porém mais eficientes. (vide seção 2.2)
- Permite que servidores listem de antemão recursos relacionados a outros, para que clientes possam adquiri-los em simultâneo. Esse mecanismo visa acelerar em especial o carregamento de páginas e aplicações Web complexas, que incluem dezenas de recursos externos.

- Compressão de metadados (*headers*) eficiente e resistente a ataques contra criptografia.

O HTTP/2 vem sendo velozmente adotado por diversos projetos de software de grande utilização, em especial de código-aberto. Navegadores como Chrome e Firefox ("HTTP/2 Frequently asked questions" 2015) e servidores como Apache e ngnix (Nottingham 2015) o suportam em caráter final ou experimental. Existem, porém, oportunidades para novas implementações explorando diferentes metodologias e aspirações.

Este trabalho descreverá uma implementação do HTTP/2 na linguagem de programação orientada a objetos e funcional Scala ("The Scala programming language", [s.d.]), utilizando o modelo de atores (Hewitt 1977) implementado pelo conjunto de ferramentas ("Akka", [s.d.]). Esta combinação foi escolhida devido a características como:

- Expressividade da linguagem, permitindo elegância e coesão de implementação
- Eliminação de compartilhamento de dados entre entidades concorrentes, facilitando a escalabilidade para múltiplos processadores e máquinas
- Maturidade da plataforma Java e seus ecossistema de software
- Facilidade de uso e eficiência da Akka para criação de sistemas concorrentes eficientes

Denominar-se-a daqui para frente o protocolo original HTTP e suas revisões até 1.1 como "HTTP/1". A versão 2 definida em 2015 será mencionada como "HTTP/2", e aspectos comuns a ambas serão referidos genericamente como "HTTP".

2 O protocolo HTTP

Para compreender as diferenças entre o HTTP/1 e o HTTP/2, é interessante, se não necessário, o conhecimento da estrutura das informações que podem ser transmitidas por ambos. Esta estrutura foi intencionalmente mantida em todas as transições para novas versões do protocolo até o presente momento, permitindo que aplicações existentes recebam a menor quantidade possível de modificações para usufruir de possíveis melhorias na eficiência de transmissão na rede.

2.1 Descrição e posição no modelo da Internet

O HTTP é um protocolo de nível de aplicação para comunicação efêmera (sem estado) na Internet. Ele define a estrutura e representação de *mensagens* a serem trocadas por sistemas de informação, mas não é responsável pela entrega destas mensagens a seus destinatários. Esta responsabilidade é delegada ao *Transmission Control Protocol* (TCP), que provê a capacidade de criar uma *conexão* entre dois sistemas, através da qual podem ser enviados e recebidos *octetos* - unidades de informação compostas por 8 bits - de maneira ordenada.

Esta disposição segue o modelo de comunicação de rede da Internet, no qual protocolos de níveis de abstração mais baixos e escopos menos extensos servem de base para outros gradativamente mais próximos das aplicações. Embora a separação em camadas excessivamente definidas não seja um objetivo explícito de engenharia da Internet, e sua composição se mantenha em fluxo com a evolução de seus componentes ao longo do tempo, é interessante analisar o papel de todos os protocolos sobre os quais o HTTP se posiciona.

Kurose e Ross (2010) definem cinco camadas de protocolos que atuam em conjunto na Internet, que podem ser descritas brevemente, da mais abstrata à menos abstrata:

- **Camada de Aplicação** Abrange a comunicação de dados entre aplicações, em uma mesma ou diferentes máquinas, de maneira especializada em cada protocolo para um dado fim. Nela residem diversos protocolos, dentre os quais: HTTP/1, HTTP/2, SMTP (e-mail), FTP (transferência de arquivos).
- **Camada de Transporte** Responsável pela transmissão efetiva dos dados entre as aplicações, usualmente através dos protocolo TCP ou UDP. O TCP é orientado a conexão, exigindo o pré-estabelecimento de um canal de comunicação entre aplicações para garantir a entrega dos dados na ordem de envio. O UDP fornece apenas o envio de *datagramas* individuais de tamanho variável sem garantias de entrega ou ordem.
- **Camada de Rede** Responsável pela transmissão de dados entre máquinas quaisquer, incluindo seu trânsito entre redes distintas, se necessário, para alcançar o destino final. O IP (*Internet Protocol*) é o único mecanismo utilizado na Internet para esse fim.

- **Camada de Interface** Responsável pela transmissão direta de dados entre máquinas de uma mesma rede. Em geral um protocolo desta camada, como Ethernet ou Wi-Fi, movimenta pacotes IP dentro de cada segmento de rede que compõe o caminho completo entre o remente e o destinatário.
- **Camada Física** Representa uma ligação física entre máquinas que permite que bits que compõe os dados das camadas superiores transitem. Um mesmo protocolo na Camada de Interface pode ser utilizar meios físicos diversos. O protocolo Ethernet, por exemplo, suporta fibra-óptica e cabos de par-trançado de cobre.

2.2 Estrutura e semântica de mensagens

O HTTP ("Hypertext transfer protocol (HTTP/1.1): Semantics and content" 2014) define *mensagens* compostas de dados e metadados, a serem transmitidas entre aplicações, representando *recursos* identificados por caminhos denominados URLs (*Uniform Resource Locator*). Mensagens podem ser *pedidos*, indicando uma requisição de informação ou tomada de ação, ou *respostas*, representando total ou parcialmente o estado de recursos sujeitos de um pedido. Pedidos contém exatamente um *método* indicando seu objetivo e um caminho indicando o recurso a qual se aplicam, e respostas contém exatamente um *código* numérico que corresponde a ação efetiva tomada pelo destinatário do pedido após o processamento, ou uma condição de erro.

Metadados são representados por uma sequência de pares de *strings*, em disposição chave-valor. Algumas chaves tem comportamento pré-definido, mas outras não-padronizadas podem ser utilizadas a critério de cada interlocutor sob o prefixo "X-".

Nos pedidos, os metadados determinam a representação do recurso a ser obtida ou que está sendo enviada, selecionam parâmetros de comunicação ou comportamentos condicionais, fornecem credenciais, etc. Em respostas, fornecem informações adicionais sobre um recurso, como data de modificação, tipo de conteúdo, comprimento do corpo, diretivas de cache, URL atribuída, etc. Em ambos os casos, também podem ser enviados os chamados *cookies*, *strings* utilizados para armazenamento de estado, já que o HTTP em si não fornece essa funcionalidade.

O *corpo* de uma mensagem corresponde à representação do recurso em um formato negociado entre o *cliente* (que iniciou o pedido) e o *servidor* (que envia uma resposta). Os metadados devem indicar em qual formato o corpo se encontra, para que ele possa ser processado adequadamente após recebido.

A presença ou não de um corpo em mensagens é usualmente determinada pelo método utilizado no pedido, e no caso de respostas onde é opcional, possivelmente pelo código de resposta selecionado pelo servidor.

Adicionalmente, métodos são classificados como *seguros* e/ou *idempotentes*.

Métodos seguros não devem alterar informações armazenadas em recursos, ao menos de nenhuma maneira pela qual o cliente possa ser responsabilizado. Por exemplo, adquirir um recurso com GET não deve modificá-lo.

Pedidos de métodos idempotentes devem produzir um mesmo efeito sempre que repetidos. Por exemplo, repetir um GET deve sempre gerar uma resposta com a mesma representação do recurso selecionado caso ele não haja sido modificado entre repetições. Repetir um PUT deve sempre substutuir completamente um recursos pelas informações contidas no pedido.

Tabela 1: Características de métodos HTTP

Método	Propósito	Corpo no pedido	Corpo na resposta	Seguro	Idempotente
GET	Aquisição de recurso	Não	Sim, opcional para erros	Sim	Sim
HEAD	Aquisição de metadados	Não	Não	Sim	Sim
PUT	Criação ou substituição de recurso	Sim	Opcional	Não	Sim
POST	Criação de recurso ou requisição de processamento	Sim	Opcional	Não	Não

Método	Propósito	Corpo no pedido	Corpo na resposta	Seguro	Idempotente
DELETE	Deleção de recurso	Não	Opcional	Não	Sim
OPTIONS	Aquisição de opções de transmissão	Opcional	Opcional	Sim	Sim
TRACE	Verificação de transmissão	Opcional	Opcional	Sim	Sim
CONNECT	Criação de conexão intermediária (proxy)	Não	Opcional	Não	Não

2.3 Representação de rede do HTTP/1

O HTTP/1 define uma representação para a semântica do HTTP em formato textual. Uma mensagem sempre se inicia com uma linha contendo a versão do protocolo em uso e informações básicas de um pedido ou resposta. Num pedido, contém o método e URL desejados, e numa resposta, um código de resultado numérico e possivelmente uma mensagem correspondente.

Em seguida são listados os pares chave-valor que compõe os metadados, separados por quebras de linha. O fim da chave é indicado por um caractere ":" (dois pontos) seguido de espaço, e tudo que segue até o fim da linha é atribuído ao valor. Uma quebra de linha dupla sinaliza o fim dos metadados, e possível início do corpo. O fim do corpo pode ser sinalizado pela inclusão de seu comprimento dentre os metadados, com o fim da conexão caso uma única mensagem seja transmitida por sentido durante sua existência, ou com algum mecanismo especial definido por um tipo de conteúdo mais complexo.

Embora pareça simples superficialmente, o reconhecimento de documentos HTTP e todas as suas peculiaridades, incluindo suporte a enorme gama de *software* que participa da Internet, se mostra complexo e conducente a falhas quando implementado. Diversas vulnerabilidades de segurança ("CVE-2012-0053" 2012; "CVE-2015-6290" 2012; "CVE-2012-1180" 2012) em múltiplos

projetos de software são causadas por essas dificuldades.

```
GET /hello.txt HTTP/1.1
Host: www.example.com:80
```

Fragmento 1: Exemplo de pedido HTTP/1.1

```
HTTP/1.1 200 OK
Date: Fri, 28 Jan 2016 11:55:00 GMT
Content-Type: text/plain
Content-Length: 13
Hello, World!
```

Fragmento 2: Exemplo de resposta HTTP/1.1

3 O protocolo HTTP/2 - Motivação e principais características

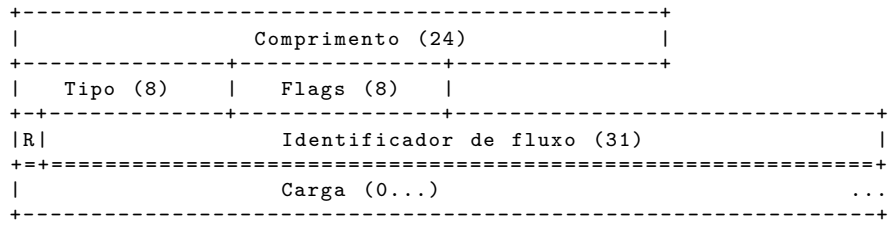
Como sucessor do HTTP/1, o HTTP/2 foi desenvolvido para atender a diversos casos de uso de comunicação na Internet de maneira mais eficiente que seu predecessor. Dentre os quais um dos mais utilizados é a visualização de documentos e aplicações na Web. Para alcançar tais objetivos, utiliza mecanismos mais complexos que o HTTP/1, que serão descritos a seguir.

3.1 Formato Binário

O HTTP/2 substitui a representação textual de mensagens do HTTP/1 por um conjunto de pacotes independentes denominados *frames*, com diversos propósitos diferentes, especificados em termos de sua representação binária. O significado semântico dos elementos de uma mensagem é compartilhado com versões anteriores do HTTP, mas sua transmissão pode ser mais eficiente devido a adição de funcionalidades baseadas no formato binário.

Um frame é composto por um *cabeçalho* de tamanho fixo, seguido de uma *carga* composta por um número variável de octetos. O cabeçalho indica o tipo do frame, seu comprimento total incluindo a carga, um inteiro representando *flags* de parâmetros de comportamento e/ou composição da carga, e um indentificador de fluxo (a ser explicado nas seções seguintes).

Os frames são as unidades mínimas de transmissão do HTTP/2, muito diferentemente do HTTP/1, que especifica somente a entrega de mensagens completas. Esta distinção é de vital importância para que uma única conexão TCP possa ser utilizada para transmissão de um número arbitrário de mensagens em um período longo de tempo, eliminando ineficiências trazidas por reconexões - em especial a espera introduzida por algoritmos de controle de congestão do TCP que iniciam transmissões com largura de banda reduzida e acrescida gradativamente.



Fragmento 3: Disposição de um frame HTTP/2 (traduzio do RFC7540). Números são comprimentos em bits.

3.2 Compressão de Headers (HPACK)

O HTTP/2 define um sub-formato próprio chamado HPACK para transmissão mais enxuta de metadados de mensagens. A criação de uma nova representação é motivada por deficiências de segurança observadas em mecanismos previamente utilizados com o HTTP/1. Tentativas anteriores de combinar compressão e privacidade mostraram-se vulneráveis a ataques, que alcançaram a extração de informações através da observação de padrões estatísticos nos dados comprimidos (CRIME (Kelsey 2002), BREACH (Prado, Harris, e Gluck 2013)).

O HPACK supera estas dificuldades utilizando técnicas e algoritmos mais simples, e talvez menos eficientes, mas que até o presente momento se mostram resistentes a ataques.

A necessidade de proteção de conteúdo de *headers* se deve principalmente ao uso dos *cookies* para o compartilhamento de estado entre interlocutores. Estado esse que em muitos casos inclui credênciais de identificação, o que permite que um atacante capaz de interceptá-lo falsifique a identidade do atacado.

Três mecanismos são utilizados pelo HPACK para diminuir a banda de transmissão de *headers* sem comprometer a privacidade:

1. **Tabela estática de conteúdos pré-definidos** Uma lista de conteúdos mais comuns é compartilhada entre todos os programas que implementam o HTTP/2. Cada chave, valor, ou conjunto chave-valor que seja definido na tabela pode ser transmitido somente como um índice inteiro, substituindo a cadeia de caracteres usual.

2. **Tabela dinâmica de conteúdos transmitidos na conexão** Cada transmissão de *header*, caso uma exceção não seja explicitamente requisitada, é armazenada em uma *tabela dinâmica* de entradas. Repetições futuras de conteúdos equivalentes podem ser substituídas por um índice, explorando semelhanças entre mensagens de uma mesma conexão.

3. **Compressão de literais por codificação de Huffman** *Headers* que não possam fazer uso das regras anteriores podem ser comprimidas com um *Código de Huffman* simples, com símbolos de substituição pré-definidos, escolhidos para máxima compressão com base em amostras de tráfego real. Esse esquema não possui nenhuma fraqueza estatística conhecida.

3.3 Pré-carregamento (compromissos de envio)

No HTTP/1 toda resposta é atrelada a exatamente um pedido oriundo da outra parte. Já no HTTP/2 é possível responder a um pedido com diversos recursos através de *compromissos de envio*.

Um compromisso de envio consiste em um pedido sintético gerado por uma parte que normalmente estaria em posição de produzir uma única resposta, visando adiantar uma requisição provável sem que seja necessário que um pedido real aconteça.

O principal objetivo dessa funcionalidade é acelerar páginas e aplicações Web complexas, que incluem dezenas de recursos adicionais como imagens e *scripts*. Cada um destes exige uma requisição adicional, incorrendo gasto maior de banda com *headers*, e pior aproveitamento da conexão devido ao tempo ocioso entre cada transmissão. Práticas como combinação de imagens

tornaram-se comuns para evitar estes problemas (Connors e Sullivan 2010), mas são obsoletas no HTTP/2.

Para compreender este processo denominaremos "cliente" o interlocutor que envia um pedido inicial, e recebe uma resposta principal e vários compromissos, e "servidor" o que recebe o pedido inicial e responde. Os passos que compõe uma interação de pré-carregamento são os seguintes:

1. Cliente faz um pedido inicial;
2. Servidor recebe pedido, prepara resposta, e decide incluir compromissos;
3. Servidor envia resposta principal sem fechar o fluxo;
4. Servidor envia compromissos contendo pedidos sintético, no mesmo fluxo da resposta principal, reservando um novo fluxo para cada;
5. Servidor finaliza o fluxo de resposta principal;
6. Cliente tem a chance de observar os pedidos sintéticos dos compromissos e rejeitá-los se desejado, através do fechamento dos fluxos recém-abertos;
7. Em algum momento conveniente, servidor inicia transmissão de respostas nos fluxos dos compromissos que não foram rejeitados;
8. Servidor fecha fluxos dos compromissos quando as transmissão correspondentes terminam.

3.4 Multiplexação

O HTTP/2 define trocas de mensagens em termos de múltiplos fluxos de dados (*streams*) em uma mesma conexão.

Eles representam canais de comunicação independentes pelos quais múltiplas mensagens podem trafegar entre os interlocutores, possivelmente de maneira simultânea, utilizando uma única conexão TCP. Mensagens de um fluxo são divididas em diversos *frames*, e frames de diversos fluxos são intercalados na transmissão de rede, criando um nível de concorrência não antes presente no HTTP/1.

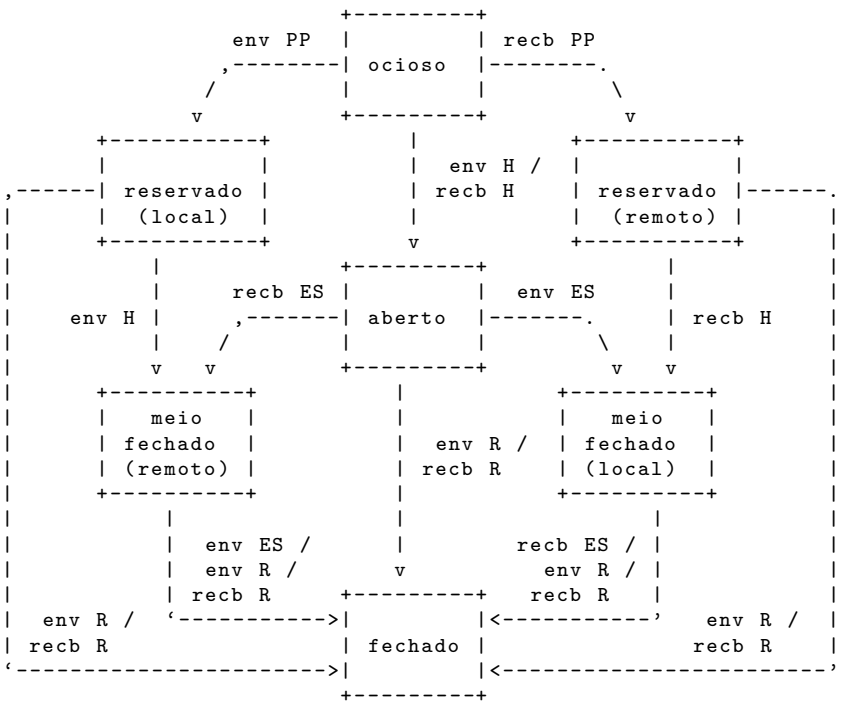
Esta capacidade de multiplexação pode ser considerada uma continuação ou extensão do *pipelining* definido no HTTP/1.1. Esta prática consiste em permitir que interlocutores enviem mensagens consecutivas sem aguardar respostas a cada uma delas, mas ainda exige que todas as mensagens sejam enviadas e processadas em ordem (*first-in-first-out*). Embora uma melhora significativa em comparação com o procedimento mais simplístico, o pipelining

sofria de deficiências, como a possibilidade de *head-of-line blocking*, situação onde um pedido excessivamente custoso impede que outros mais simples sejam processados concomitantemente.

A multiplexação do HTTP/2 remove limitações de ordem, permitindo que mensagens trafeguem de maneira realmente simultânea e independente, ao custo de complexidade de implementação de servidores e clientes, que precisam gerenciar estado muito mais complexo.

Fluxos são identificados por inteiros positivos de 31-bits selecionados de maneira crescente. Fluxos iniciados pelo cliente tem sempre valor par, e pelo servidor ímpar. O processo de envio de um pedido (por qualquer uma das partes) exige a abertura de um fluxo no qual ele transitará. A resposta e possíveis compromissos de envio (vide seção correspondente) utilizam o mesmo fluxo do pedido, que então é fechado.

Não existe contingência para o esgotamento do espaço de valores de índices de fluxo, o que implica num número máximo de 2147483647 trocas completas de mensagens por conexão. Implementações devem simplesmente reestabelecer a conexão ao encontrar tal caso.



```
env :   interlocutor envia este frame
recb:   interlocutor recebe este frame

H:  frame HEADERS (com respectivos frames CONTINUATION)
PP: frame PUSH_PROMISE (com respectivos frames CONTINUATION)
ES: frame com flag END_STREAM habilitada
R:  frame RST_STREAM
```

Fragmento 4: Diagrama de estados de um fluxo de transmissão no HTTP/2 (traduzido do RFC7540)

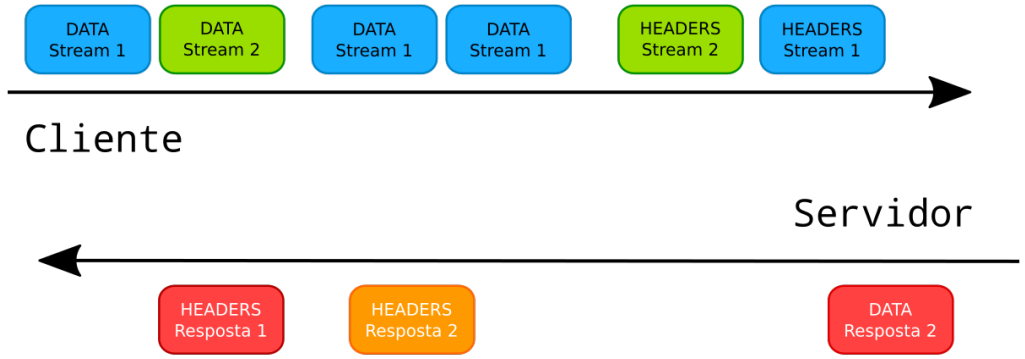


Figura 1: Representação de multiplexação numa conexão HTTP/2

3.5 Controle de fluxo

O HTTP/2 define *janelas de controle de fluxo* para a conexão como um todo e cada fluxo em separado. Elas definem uma capacidade máxima de recepção de dados de um interlocutor com recursos finitos, de modo que eles não sejam esgotados por uma contraparte de maior poder computacional.

A qualquer momento, um remetente envia um *frame* definindo uma quantidade de dados, em octetos, que deseja ou consegue receber até segunda ordem. Ao receber este *frame*, o destinatário deve contabilizar o que envia, e não deve exceder o volume definido até o recebimento de um novo valor. Em caso de esgotamento, somente uma *atualização de janela* permite que os envios sejam retomados.

A modalidade individual por fluxo da janela pode ser utilizada também como mecanismo de priorização, além dos outros já definidos pelo protocolo (vide

seção 2.8). Ao estabelecer um volume máximo atribuído a um fluxo em um período de tempo, outros podem receber chances de fazer uso da banda disponível.

Este mecanismo se assemelha conceitual e praticamente ao controle de fluxo existente na camada de transporte, como por exemplo no TCP. Sua posição na camada de aplicação do modelo de rede lhe provém usos e capacidades diferentes, porém. Ele permite que o controle seja aplicado pela aplicação, e não somente por pressão de recursos observada pelo sistema operacional, como é tradicional na camada de transporte. O HTTP/2 distingue *frames* de dados e controle na contagem da janela, o que evita situações de perda de responsividade devido a proximidade do esgotamento.

3.6 Priorização

É possível atribuir dependências e níveis de prioridades a fluxos distintos, indicando maior importância ou urgência a mensagens neles enviadas.

Fluxos subordinados só podem receber recursos caso seus superiores estejam ociosos ou em espera. Num mesmo nível da hierarquia, recebem recursos proporcionais a um valor inteiro atribuído a cada um, representando uma fração do total disponível. (por exemplo, 3 fluxos de prioridades 3, 5 e 10, respectivamente receberiam 3/18, 5/18 e 10/18 da banda de transmissão disponível)

4 Tecnologias e paradigmas utilizados

4.1 A Linguagem Scala

Scala é uma linguagem multi-paradigma, que tem como principal característica a combinação de orientação a objetos - compatível com a plataforma e linguagem Java - e programação funcional.

Possui um sistema de tipos poderoso, com funções de primeira-classe, tipos parametrizados e inferência local, e permite expressar sucintamente programas de diversos tipos.

Foi escolhida para elaboração desse trabalho devido a alta maturidade do ecossistema Java e da plataforma Akka, seu alto poder expressivo e foco em estruturas de dados imutáveis, concorrência e elegância de código.

4.2 A Plataforma Akka

Akka é uma plataforma que implementa o modelo de atores para criação de sistemas concorrentes, tolerantes a falhas e de alta escalabilidade em Scala ou Java. Permite programar fluxos de dados de maneira assíncrona e eficiente, incluindo servidores TCP como é necessário para utilização do HTTP.

Atores são entidades computacionais que se comunicam somente através de passagem de mensagens discretas, sem mecanismos de sincronização ou estado diretamente compartilhados entre eles. O modelo de implementação da Akka foi inspirado na linguagem Erlang, com adaptações necessárias para seu bom funcionamento sobre a *Java Virtual Machine*, mas que diferente significativamente do modelo tradicional de programação concorrente com *threads* de memória compartilhada. O mecanismo exato de execução dos atores não é especificado, apenas seus princípios de funcionamento, o que permite que múltiplas implementações de atores locais ou distribuídos sejam utilizadas conforme escolha do programador ou até do usuário de um programa baseado na platafora.

Um sub-projeto da Akka que é parte fundamental deste trabalho é a biblioteca *Akka Streams*, que implementa um modelo para programação reativa e orientada a fluxos de dados utilizando a concorrência de atores da *Akka*. Ela eleva o nível de abstração para permitir a programação sem o gerenciamento manual do tempo de vida de atores. Apenas é necessário especificar entidades que geram e transformam dados e como elas se conectam. Uma explicação detalhada de seu funcionamento pode ser encontrada nas seções seguintes.

4.3 O modelo de concorrência de atores

Para confecção de um software capaz de fazer uso de múltiplos processadores de maneira eficiente, correta e elegante, foi utilizado o modelo de atores como implementado pela plataforma Akka.

Um ator é uma entidade computacional que interage com outras - recebendo ou enviando informações - somente através de mensagens. Em resposta a uma mensagem, um ator pode enviar outras, criar novos atores, e/ou modificar seu comportamento para a próxima recepção. Diferentemente de outros modelos de concorrência, em especial os de nível mais baixo de abstração, o compartilhamento de estado entre os atores não é permitido. Eles devem se comunicar somente através de mensagens.

Com o Akka atores podem transitar livremente entre threads do sistema operacional em diferentes processamentos de mensagens, e operações asíncronas são modeladas através de *futuros* da linguagem Scala.

Um futuro encapsula um valor a ser produzido ou uma computaçãoa a ser efetuada em algum momento futuro indeterminado, e permite definir transformações sobre esse resultado que só serão executadas após sua produção. Esta combinação permite que o gerenciamento de threads seja completamente invisível ao programador se desejado.

4.4 Programação reativa e orientada a fluxos

Em adição ao modelo de atores, a metodologia de programação orientada a fluxos de dados foi empregada na construção do servidor. O comportamento do software é definido por um grafo de estágios de processamento independentes, através dos quais trafegam mensagens atômicas e imutáveis. As rotas e transformações aplicadas a estas mensagens representam as diferentes regras e funcionalidades definidas pelo protocolo HTTP/2, que combinadas correspondem ao fluxo de entrada e saída de bytes numa conexão TCP.

Em especial, foi utilizada a biblioteca *Akka Streams*, que aplica a chamada *filosofia reativa* (Jonas Bonér 2015) e fornece ferramentas para construção de sistemas de fluxos de dados complexos nas linguagens Java e Scala.

A filosofia reativa propõe que sistemas competentes devem observar certos princípios e possuir certas características, dentre as principais:

- *Responsividade*: Fornecer tempos de resposta rápidos, consistentes e confiáveis;
- *Resiliência*: Detectar e recuperar-se de falhas sempre que possível;
- *Elasticidade*: Absorver variações de demanda e ser escalável;

- *Orientação a mensagens*: Garantir baixo acoplamento e alto isolamento de componentes estabelecendo fronteiras claras entre eles;

Estas propriedades permitem que os sistemas lidem com grandes volumes de dados de maneira eficiente, façam uso de paralelismo computacional sem perda de correção e forneçam garantias de serviço a seus usuários.

Para alcançar essas metas, a *Akka Streams* define mecanismos para modelagem de *fontes*, *sumidouros* e *fluxos* de dados. Respectivamente, representam computações que *produzem*, *transformam* e *absorvem* dados. A composição destes elementos, fazendo com que dados fluam das entradas de uns para saídas de outros, permite modelar o trânsito de informações em alto nível de abstração, observando apenas exigências das transformações em si, e não de seus detalhes de implementação.

Conexões entre elementos de um grafo de fluxo tem demanda controlada pelo consumidor. Um nó do grafo só pode produzir um dado em uma de suas saídas se o nó que controla a entrada corresponde manifestou a capacidade ou necessidade de recebê-lo. Somente é definida a demanda binária: apenas a possibilidade de propagação de zero ou uma instância de dados é comunicada entre os nós em um dado momento. Aquisição de múltiplos dados é modelada por uma sequência de operações de recebimento em uma entrada intercaladas com os correspondentes envios nas saídas, garantindo o balanço do grafo como um todo.

Grafos de fluxos (e seus subgrafos) são construídos em duas etapas distintas. Define-se inicialmente um *molde* imutável, representando estágios de processamento e suas conexões, mas não os mecanismos que serão utilizados para executá-los. Nesse formato, múltiplos subgrafos podem ser combinados para formar grafos maiores, através do poliformismo de suas *formas* (*shapes*). Por exemplo, qualquer subgrafo que possui exatamente uma entrada e uma saída ainda não conectadas pode ser manipulado como se fosse um fluxo, facilitando a composição e reusabilidade, e a substituição de subgrafos simples por outros mais complexos caso se torne necessário.

A segunda etapa consiste na *materialização* de um grafo, que nada mais é do que a alocação e criação dos recursos necessários a execução, possivelmente customizada por parâmetros adicionais. Esse processo pode também produzir valores, em geral utilizados para fornecer vias de comunicação com os recursos recém-criados, ou com estágios internos do grafo com funções especializadas.

No exemplo do servidor do qual trata esse artigo, as produções da materialização representam o estado de uma conexão TCP iniciada com um cliente, e fornecem a capacidade de desconexão sob controle da aplicação.

5 Detalhes selecionados de implementação

5.1 Descrição de alto nível dos estágios de processamento

Dezenas de estágios simples foram compostos para construir essa implementação do HTTP/2. No presente momento somente o aspecto de servidor foi criado, mas o baixo acoplamento e natureza diminuta dos nós componentes permitiriam a elaboração de um cliente com adição de apenas uma fração do código total.

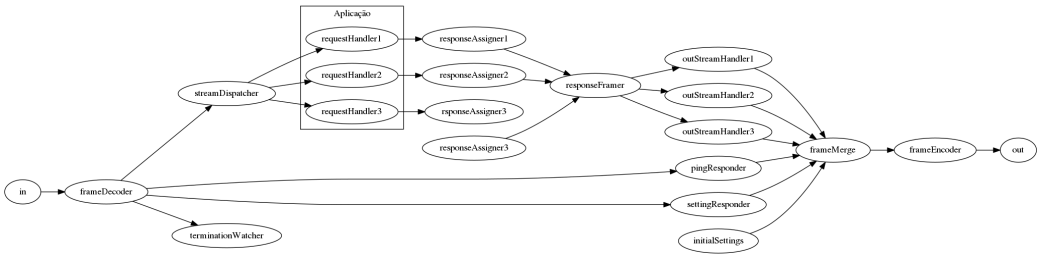


Figura 2: Representação do grafo de fluxo do servidor

in Entrada de bytes da conexão TCP (encapsulada por TLS)

frameDecoder Fluxo de interpretação de bytes para frames segundo a definição do HTTP/2

terminationWatcher Sumidouro que apenas observa frames recebidos procurando por indicação de fechamento remoto de conexão (GOAWAY), para sinalizar finalização para outros estágios

pingResponder Fluxo que responde a pedidos de teste de conectividade (PING)

settingsResponder Fluxo que responde confirmando aceitação de pedidos de mudança de configuração (SETTINGS)

streamDispatcher Subgrafo com múltiplas saídas responsáveis pelo processamento de pedidos recebidos. Ao receber um frame que indica o início de um pedido, uma saída é selecionada para receber todos os frames desse mesmo durante toda a vida da conexão. O número de saídas corresponde ao limite máximo de pedidos simultâneos configurado (e divulgado a contraparte nos parâmetros de conexão). Delega o gerenciamento de estado para um ator compartilhado com os fluxos subgrafos de saída da conexão, para manutenção da coerência entre as duas direções.

requestHandler(1..n) Subgrafos responsáveis pelo processamento de pedidos e geração de respostas. Um molde é provido pela aplicação, e então replicado o número de vezes necessário para preencher todos os fluxos de entrada.

responseAssigner(1..n) Subgrafo responsável por atribuir um fluxo de saída para cada resposta produzida. Delega o gerenciamento de estado para um ator (vide *streamDispatcher*) para manutenção de coerência.

responseFramer Fluxo responsável por converter respostas em fontes de *frames*, correspondendo a seus metadados, dados, e pedidos secundários (*server pushes*). Fontes, assim como quaisquer outros objetos, podem trafegar pelo grafo, e assim ocorre para desacoplar a ordem de produção das várias resposta de suas atribuições nos fluxos de saída.

outStreamHandler(1..m) Fluxo responsável por desagrupar *frames* de cada resposta e verificar sua aceitabilidade dado o estado atual do fluxo de saída correspondente. Delega o gerenciamento de estado para um ator (vide *streamDispatcher*) para manutenção de coerência.

initialSettings Fonte que produz um único *frame* contendo configurações iniciais que devem ser enviadas à contraparte imediatamente após o estabelecimento da conexão.

frameMerge Subgrafo responsável por linearizar *frames* recebidos por diversos caminhos, decidindo sua ordem de saída

frameEncoder Fluxo responsável por converter *frames* da saída em sequências de bytes

out Saída de bytes na conexão TCP (encapsulada por TLS)

5.2 Transport Layer Security - extensão ALPN

Para que uma parte seja capaz de identificar qual versão do HTTP/2 outra implementa (se alguma), são definidos diversos mecanismos de negociação para os diferentes tipos de transporte sob os quais a comunicação pode ocorrer.

O transporte em claro pode ser iniciado de duas maneiras:

1) *Com conhecimento prévio*: se uma das partes já possui informação externa que indica suporte da outra ao HTTP/2, apenas é necessário enviar um *preâmbulo* de conexão imediatamente após estabelecimento do canal de comunicação. Ele é uma sequência pré-determinada de bytes escolhida para evitar que software compatível apenas com HTTP/1 a reconheça.

2) *Atualização a partir do HTTP/1*: utilizando o mecanismo de *Upgrade* definida no HTTP/1.1, uma parte pode informar seu suporte ao HTTP/2 nos metadados de um pedido enviado, incluindo conjuntamente uma representação dos seus padrões de configuração, que em outros caso seriam transmitidos após o preâmbulo. Caso a contraparte aceite a atualização, também o comunica através dos metadados iniciais de suas resposta, e em seguida continua a comunicação já com o protocolo novo.

O transporte criptografado é feito através do padrão *Transport Layer Security*, assim como no HTTP/1.1, mas com a adição de uma extensão recentemente desenvolvida denominada *Application Layer Protocol Negotation* (ALPN). Ela codifica um mecanismo de negociação de protocolos de camada de aplicação centralizado, parte da negociação de parâmetros do TLS, liberando aplicações dessa tarefa.

Infelizmente a implementação de TLS da plataforma Java não suporta nativamente a extensão ALPN até sua versão mais corrente (JDK 8, novembro de 2015), e vários navegadores deliberadamente não suportam transporte em claro.

O projeto Jetty ("Jetty - servlet engine and http server - Eclipse", [s.d.]), porém, disponibiliza uma versão modificada das bibliotecas de classe do Java com suporte a ALPN. Estas foram utilizadas nna implementação do servidor, como único mecanismo de negociação disponível. A comunicação em claro não foi testada ou habilitada com nenhum dos modelos supracitados, já que sua

utilidade prática é limitada. A natureza da implementação dos fluxos torna, porém, sua implementação relativamente simples caso se mostre interessante no futuro.

5.3 Compressão de headers (HPACK)

5.3.1 Descrição detalhada

A transmissão de headers de mensagens no HTTP/2 é feita através de um protocolo próprio chamado HPACK, criado especificamente para este propósito. Como no HTTP/1, headers são uma série de chaves e valores definidas por cadeias de caracteres, representando metadados de uma mensagem, como o formato do conteúdo, localização, data de modificação, informações dos interlocutores, etc.

Como o HTTP não define um mecanismo para preservação de estado compartilhado entre múltiplas mensagens, headers são comummente utilizadas para esse fim. Um dos mecanismos mais populares são os *Cookies*, também mapeamentos entre chaves e valores, que são enviados por um remetente para que o destinatário possa o identificar, reiniciar uma sessão anterior, ou reaver qualquer outro tipo de informação. Sua funcionalidade os torna sensíveis a vazamento, já que protegem acesso a sistemas de todos os tipos na Web.

A natureza repetitiva destes metadados faz com que sua compressão seja muito vantajosa, especialmente para mensagens curtas, ou com metadados semelhantes entre si. No HTTP/1, durante anos utilizaram-se algoritmos de compressão aplicados sobre uma mensagem inteira, incluindo corpo e headers, protegida sob um protocolo de confidencialidade como o TLS. Descobriu-se, porém, que esse esquema é vulnerável a ataques que recuperam gradativamente informações através da observação da eficiência da codificação. (CRIME (Kelsey 2002), BREACH (Prado, Harris, e Gluck 2013)), e portanto essa prática é **banida** no HTTP/2.

Substitui-na o HPACK, que comprime headers individualmente, através da manutenção de uma tabela de cadeias comuns e previamente observadas em uma conexão. Cadeias longas ou que não foram observadas anteriormente são comprimidas através da Codificação de Huffman (Huffman 1952) com símbolos estáticos, que não é sujeita a nenhuma ataque de correlação conhecido.

Remetentes podem especificar que certas chaves são sensíveis e não devem ser comprimidas de nenhuma maneira (como por exemplo os já mencionados cookies).

5.3.2 Aplicação da Codificação de Huffman sem árvores

Para comprimir *headers* que não existem na tabela pré-definida de valores comuns ou que ainda não foram transmitidas na conexão atual o HPACK utiliza uma versão simplificada da Codificação de Huffman, na qual símbolos de tamanho fixo (um octeto) corresponde de maneira estática a códigos de tamanho variável (até 32-bits).

Estas particularidades permitem substituir a implementação tradicional, utilizando arvores de prefixos bit-a-bit, por um método que busca octetos completos em vetores gerados à partir da estrutura da codificação. O número de buscas efetuadas é proporcional ao número de octetos, e cada busca toma tempo de fator constante. O pré-processamento é linear, e evita a criação de centenas de nós de uma árvore.

O algoritmo de pré-processamento consiste em gerar um número pequeno de tabelas indexadas por valores de octetos, que imediatamente determinam se um octeto produz um símbolo, ou se é apenas prefixo de um código, e portanto deve determinar uma nova tabela a ser utilizada. O principal mecanismo dessa geração é a *extensão* dos códigos de menos de 8-bits para octetos completos, produzindo o conjunto de todos os octetos possíveis que tem o código como seu prefixo. Por exemplo, um código de 6-bits seria representado por 4 valores de octetos, correspondentes a concatenação de si mesmo e todos os sufixos de 2-bits possíveis (totalizando 8).

A decodificação consiste em observar octetos de entrada, e acessar as diferentes tabelas baseado no valor obtido. A tabela de símbolos atual é acessada com o octeto como índice. Caso não presente, o octeto é considerado um prefixo, e uma busca na tabela de prefixos é feita também, selecionando uma nova tabela de códigos. O processo é repetido até a produção de um símbolo ou a exaustão das tabelas (indicando falhada), e então reiniciado com os valores originais até o fim da entrada.

5.4 Testes

5.4.1 Testes Sintéticos

Para verificar a correção da implementação das etapas de codificação e decodificação das diferentes estruturas de dados e formatos definidos pelo protocolo HTTP/2, foi utilizado o conjunto de amostras fornecido pela Comunidade de HTTP/2 do Japão ("http2-frame-test-case - GitHub", [s.d.]). Dezenas de pares de sequências de bytes e suas correspondentes interpretações como delineadas na definição do protocolo são representados em formato JSON. Tais documentos são convertidos para as representações internas do software, e comparados com resultados de processos de leitura e escrita para verificar sua correção.

Adicionalmente, os casos de teste fornecidos na própria especificação (*Request for Comments*) do protocolo foram utilizados para verificar a correção do algoritmo de compressão.

5.4.2 Testes práticos

Testes de compatibilidade foram efetuados com *softwares* pré-existentes, para comprovar a capacidade de funcionamento no mundo real. O servidor provou-se capaz de responder a pedidos dos seguintes programas:

- Navegador Mozilla Firefox 41
- Navegador Google Chrome 46
- Cliente de linha de comando nghttp2 1.2.1

Nos três casos foram feitos quatro testes com diferentes padrões:

1) Requisição única de um arquivo de texto pequeno
2) Requisição única de um arquivo de texto pequeno, respondida com pré-carregamento (server push)
3) Requisição única de um arquivo binário grande (1GB)
4) Requisição de dois arquivos em seguida em uma mesma conexão

6 Interface de programação

O servidor foi construído visando integrar-se com aplicações como uma biblioteca, que pode por elas ser incluída. O controle dos parâmetros de rede, criptografia, gerenciamento de requisições, etc é de responsabilidade da aplicação, que inicia manualmente um atrelamento a uma porta de rede, e decide como tratar uma conexão no momento do seu estabelecimento.

Foi objetivo explícito manter a compatibilidade de modelos de dados com o projeto *Akka HTTP*, implementação do HTTP/1.1 desenvolvida em conjunto com o *Akka Streams*.

Respostas HTTP/2 são modeladas através de sequências de mensagens do modelo de dados, das quais uma é atribuída como resposta imediata para um pedido. Outras são consideradas sugestões de pré-carregamento e enviadas como tal.

Segue um exemplo de aplicação simples, que serve arquivos encontrados na pasta atual. A criação completa do contexto de criptografia (`createSSLContext`) foi omitida visando brevidade. É necessário notar, porém, que para utilizar a extensão ALPN como necessário (vide seção 4.4), a escolha do contexto exige configurações específicas. As ferramentas necessárias para isso estão incluídas nesse trabalho, mas ainda precisam ser utilizadas semi-manualmente.

Note-se que o exemplo difere em apenas poucos detalhes de um equivalente para HTTP/1.1 utilizando as mesmas bibliotecas, mas automaticamente aplicando paralelismo de conexões e outros benefícios do HTTP/2.

```
1  package net.danielkza.http2.examples
2
3  import java.io.{FileInputStream, File}
4  import java.nio.file.Files.probeContentType
5  import scala.concurrent.ExecutionContext
6  import com.typesafe.config.ConfigFactory
7  import akka.actor.ActorSystem
8  import akka.stream._
9  import akka.stream.io._
10 import akka.stream.scaladsl._
11 import akka.http.scaladsl.model._
12 import akka.http.scaladsl.server._
13 import akka.http.scaladsl.server.Directives._
14 import net.danielkza.http2.Http2
```

```
15 import net.danielkza.http2.Http2.Implicits._
16 import net.danielkza.http2.model.Http2Response
17
18 object ServerExample extends App {
19   implicit val actorSystem: ActorSystem = ActorSystem("
       ↪ ServerExample", config)
20   val matSettings = ActorMaterializerSettings(actorSystem)
21   implicit val materializer: ActorMaterializer =
       ↪ ActorMaterializer(matSettings, "http2")
22   implicit val ec: ExecutionContext = actorSystem.dispatcher
23
24   val sslContext: SSLContext = createSSLContext
25
26   /* Exatamente as mesma rotas poderiam ser usadas para HTTP
       ↪ /1.1 */
27   val routes: Route =
28     path(RestPath) { path =>
29       get {
30         val file = new File("./" + path)
31         if(file.isFile && file.canRead) {
32           complete {
33             val contentType = ContentType.parse(
                 ↪ probeContentType(file.toPath)).right.get
34             val source = HttpEntity(contentType,
                 ↪ SynchronousFileSource(file))
35             HttpResponse(StatusCodes.OK, entity = source)
36           }
37         } else {
38           complete {
39             (StatusCodes.NotFound, s"File `$path` not found")
40           }
41         }
42       }
43     }
44
45   val binding = Http2().bind("0.0.0.0", port = 8080).map(
46     _.handleWith(routes)
47   ).runServerIndefinitely(actorSystem)
48 }
```

Fragmento 5: Exemplo de implementação de servidor

7 Conclusões

A combinação de programação funciona, reativa e orientada a fluxos de dados é altamente benéfica para a construção de aplicativos para comunicação em redes, facilitando o desenvolvimento e integração de componentes modulares de melhor manutenibilidade, compreensão e correção. Seu uso em ambientes de programação concorrentes permite criar sistemas que implementam protocolos complexos como o HTTP/2 de maneira elegante e eficiente.

Desenvolvimento do trabalho terá continuidade no repositório GitHub:

http://github.com/danielkza/h2scala

8 Apreciação Subjetiva

8.1 Desafios

A construção desse trabalho me exigiu alto nível de esforço sob todas as óticas de seu mérito acadêmico, em especial dedicação e tempo para absorver e aplicar conhecimento de tecnologias emergentes, complexas e pouco exploradas. Minha escolha de tema partiu da apreciação da matéria de redes em 2014 com o professor Daniel, não obstante a dificuldade dos trabalhos. Poder criar algo que à época acreditava inédito (o que durante o processo se provou não ser o caso) atraiu minha curiosidade, interesse, e de certa maneira, vaidade.

Muitos dos meses iniciais do trabalho consistiram principalmente em muita pesquisa, um pouco de hesitação, e projeções não muito otimistas. A padronização do HTTP/2 era recente, mas suas implementações poucas e por projetos já estabelecidos na área, me levando a duvidar da possibilidade de repetir o feito - mesmo que em escala menor - sozinho. Esse fato, embora não completamente, parcialmente explicou uma procrastinação que se mostrou problemática (mas felizmente não fatal) que durou alguns meses.

A grande maioria do código foi escrito nos últimos dois ou três meses, entre setembro e a entrega em novembro, o que foi acidentalmente feliz, pois permitiu utilizar ferramentas experimentais, mas muitíssimo interessantes. Ao contrário dos períodos anteriores, nesse tempo despendi até excessivas

horas, em horários não-recomendáveis, na programação (mas não igualmente na escrita). Me felicita, porém, o sentimento do término: já repeti o processo de iniciar um projeto, me entusiasmar e logo abandoná-lo por diversas vezes, e vejo o resultado desse trabalho como prova pessoal de que o término é alcançável.

Agradeço ao professor Daniel pelas contribuições, em especial dado minhas falhas com prazos e informações ao longo do ano, e espero que leitores possam ver nesse trabalho alguma utilidade ou inspiração.

8.2 Aplicabilidade de disciplinas estudadas

1) **Programação para Redes de Computadores**: Candidata mais óbvia dessa seção, me deu toda a base necessária para compreender a gama de informações e detalhes necessária para implementar um protocolo de rede, desde o funcionamento de todas as camadas envolvidas, até a motivação para todas as decisões tomadas no HTTP/2.

O trabalho no qual fui obrigado a ler a descrição do protocolo FTP me deu conhecimento básico do processo que involve a engenharia da Internet, que é fundamental para produzir software capaz de interagir com todos os outros espalhados pelo mundo.

2) **Programação concorrente/Programação paralela e distribuída**: Ambas as disciplinas foram ensinadas em níveis de abstração que envolvem mas não cobrem exatamente o utilizado no trabalho. Mas o entendimento dos fundamentos do funcionamento de programas concorrentes, sua interação com o sistema operacional, etc são indispensáveis para evitar tropeços e erros durante o desenvolvimento de sistemas complexos.

Acredito, porém, que poderia ter me beneficiado da familiarização com modelos mais abstratos de concorrência e seu uso em ambientes modernos de programação.

3) **Sistemas operacionais**: Embora parcialmente isolado dos detalhes do sistema operacional pela plataforma Java, tive que aplicar vários ensinamentos da matéria, em especial a interação com sockets e seu funcionamento.

8.3 Passos futuros

Planejo refinar a implementação do servidor para corrigir eventuais problemas e incompatibilidades, e adicionar alguns detalhes faltantes como uma implementação competente de controle de fluxo e priorização. Quando observar um grau mínimo de estabilidade, então publicar o projeto como biblioteca para uso geral, observar se existe interesse, e então programar o cliente, e posteriormente trabalhar em melhorias de desempenho.

Inicialmente tinha planos de desenvolver o trabalho junto ao projeto Akka, mas observei que os interesses e disponibilidade de seus desenvolvedores não se mostravam totalmente compatíveis com os meus. Porém, discutindo sobre dificuldades e pedindo sugestões nos canais de comunicação do projeto recebi manifestações de interesse.

Referências

"Akka". [s.d.]. http://akka.io.

Connors, Adam, e Bryan Sullivan. 2010. "Mobile web application best practices". W3C.

"CVE-2012-0053". 2012. Available from MITRE, CVE-ID CVE-2012-0053. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0053.

"CVE-2012-1180". 2012. Available from MITRE, CVE-ID CVE-2012-1180. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1180.

"CVE-2015-6290". 2012. Available from MITRE, CVE-ID CVE-2015-6290. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6290.

Hewitt, Carl. 1977. "Viewing control structures as patterns of passing messages". *Artificial Intelligence* 8 (3): 323–64. doi:10.1016/0004-3702(77)90033-9.

"HTTP/2 Frequently asked questions". 2015. IETF HTTP Working Group. https://http2.github.io/faq/.

"http2-frame-test-case - GitHub". [s.d.]. https://github.com/http2jp/

http2-frame-test-case.

Huffman, D.A. 1952. "A method for the construction of minimum-redundancy codes". *Proceedings of the IRE* 40 (9): 1098–1101. doi:10.1109/JRPROC.1952.273898.

"Hypertext transfer protocol (HTTP/1.1): Semantics and content". 2014. RFC 7231. Internet Engineering Task Force (IETF); Internet Requests for Comments; RFC Editor. https://tools.ietf.org/html/rfc7231.

"Hypertext transfer protocol version 2 (HTTP/2)". 2015. RFC 7540. RFC Editor; Internet Requests for Comments; RFC Editor. http://www.rfc-editor.org/rfc/rfc7540.txt.

"Jetty - servlet engine and http server - Eclipse". [s.d.]. http://www.eclipse.org/jetty/.

Jonas Bonér, Roland Kuhn et al., Dave Farley. 2015. "The reactive manifesto". http://www.reactivemanifesto.org/.

Kelsey, John. 2002. "Compression and information leakage of plaintext". In *Fast software encryption*, organizado por Joan Daemen e Vincent Rijmen, 2365:263–76. Lecture notes in computer science. Springer Berlin Heidelberg. doi:10.1007/3-540-45661-9_21.

Kurose, J.F., e K.W. Ross. 2010. *Computer networking: A top-down approach*. Pearson Education, Limited. https://books.google.com.br/books?id=2hv3PgAACAAJ.

Nottingham, Mark. 2015. "Implementations - http2/http2-spec wiki". IETF HTTPbis Working Group. https://github.com/http2/http2-spec/wiki/Implementations.

Prado, Angelo, Neal Harris, e Yoel Gluck. 2013. "SSL, gone in 30 Seconds". In *Black hat USA*. http://breachattack.com/resources/BREACH%20-%20BH%202013%20-%20PRESENTATION.pdf.

"The Scala programming language". [s.d.]. http://www.scala-lang.org/.