

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

MURILLO RODRIGUES DE PAULA

**Implementação de uma biblioteca
cliente HTTP/2 multiplataforma em
Lua**

Goiânia
2018

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

**AUTORIZAÇÃO PARA PUBLICAÇÃO DE TRABALHO DE
CONCLUSÃO DE CURSO EM FORMATO ELETRÔNICO**

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos “reproduzir” e “publicar” conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Implementação de uma biblioteca cliente HTTP/2 multiplataforma em Lua

Autor(a): Murillo Rodrigues de Paula

Goiânia, 07 de Dezembro de 2018.

Murillo Rodrigues de Paula

Murillo Rodrigues de Paula – Autor

Bruno Oliveira Silvestre

Bruno Oliveira Silvestre – Orientador

MURILLO RODRIGUES DE PAULA

Implementação de uma biblioteca cliente HTTP/2 multiplataforma em Lua

Trabalho de Conclusão apresentado à Coordenação do Curso de Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Bruno Oliveira Silvestre

Goiânia
2018



MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

MURILLO RODRIGUES DE PAULA


Implementação de uma biblioteca cliente HTTP/2 multiplataforma em Lua


Trabalho de conclusão de curso
apresentado à Universidade Federal de
Goiás como parte dos requisitos para a
obtenção do título de Bacharel em Ciências
da Computação.

Orientador: Prof. Dr. Bruno Oliveira
Silvestre

Aprovado em 07 de Dezembro de 2018.

BANCA EXAMINADORA


Prof. Dr. Bruno Oliveira Silvestre
Universidade Federal de Goiás
Instituto de Informática


Prof. Me. Marcelo Akira Inuzuka
Universidade Federal de Goiás
Instituto de Informática

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador(a).

Murillo Rodrigues de Paula

Graduando em Ciência da Computação na UFG - Universidade Federal de Goiás. Durante sua graduação, foi monitor da disciplina Linguagens de Programação no Instituto de Informática da UFG por um ano, realizou atividades de estágio pela Rede Nacional de Ensino e Pesquisa (RNP) por um ano e participou da Maratona de Programação ICPC por dois anos consecutivos.

Dedico este trabalho à minha mãe e ao meu pai.

Agradecimentos

Eu serei eternamente grato por todas as pessoas que de uma forma ou de outra me ajudaram na produção deste projeto. Primeiro, eu agradeço a Deus pela proteção e pela habilidade de poder estudar e trabalhar.

Ao Professor Bruno Silvestre, meu orientador. Enquanto meu orientador, professor e mentor, ele já me ensinou mais do que eu poderia escrever aqui. Ele vem me mostrando, por seu exemplo, como um bom cientista deveria ser e agir. Sem sua ajuda e apoio do começo ao fim, este projeto não seria possível.

Ao Professor Marcelo Akira, que conheci como orientador inicial do presente trabalho e hoje integra a Banca Examinadora. Obrigado pelas contribuições iniciais deste trabalho e por me ensinar a ter proatividade em projetos de software livre.

Ao apoio dos meus amigos, principalmente do Eduardo. Obrigado pela inspiração, pelas críticas, pelo aprendizado e pela agradável companhia durante a graduação.

À minha companheira Larissa, pelo amor e carinho.

Por fim, ninguém tem sido mais importante para mim nessa jornada do que os membros da minha família. Eu gostaria de agradecer especialmente aos meus pais, cujo o amor e o apoio incondicional durante toda a minha vida me possibilitaram lutar pelos meus sonhos.

Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains.

Steve Jobs,
Entrevista com Business Week, 1998.

:

Resumo

Paula, Murillo Rodrigues de. **Implementação de uma biblioteca cliente HTTP/2 multiplataforma em Lua**. Goiânia, 2018. 54p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

Bibliotecas clientes HTTP/2 propiciam a escrita de códigos de clientes HTTP/2 altamente genéricos, principalmente nas linguagens de programação de *scripting*. No entanto, apenas uma biblioteca cliente HTTP/2 está disponível para a linguagem Lua e ela não é multiplataforma, pois funciona apenas em sistemas operacionais derivados do UNIX. Isso limita a portabilidade de clientes HTTP/2 que utilizam essa biblioteca uma vez que Lua é uma linguagem muito portátil. Neste trabalho, implementamos uma biblioteca cliente HTTP/2 multiplataforma em Lua. Ela é capaz de oferecer suporte às principais funcionalidades do protocolo HTTP/2, incluindo multiplexação de fluxos em uma única conexão TCP e compressão de cabeçalhos com o protocolo HPACK. Também realizamos testes que comprovam a eficiência de um cliente HTTP/2 escrito em cima da biblioteca quando comparado com outros três clientes HTTP/2: um em C (nghttp), um em JavaScript (Node.js) e outro na biblioteca HTTP/2 atualmente disponível para Lua (lua-http).

Palavras-chave

Cliente HTTP/2, Linguagem Lua, Programação Orientada a Eventos.

Abstract

Paula, Murillo Rodrigues de. **Implementation of a multi-platform HTTP/2 client library in Lua.** Goiânia, 2018. 54p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

HTTP/2 client libraries allow the writing of highly generic HTTP/2 client code, especially in *scripting* programming languages. However, only one HTTP/2 client library is available for the Lua language and it is not multiplatform, since it works only on UNIX-like operating systems. This limits the portability of HTTP/2 clients that use this library because Lua is a very portable language. In this work, a multi-platform HTTP/2 client library in Lua was developed. It is capable of supporting the main features of the HTTP/2 protocol, including stream multiplexing in a single TCP connection and header compression with the HPACK protocol. Tests were performed that prove the efficiency of an HTTP/2 client written on top of the library when compared with three other HTTP/2 clients: one in C (nghttp), one in JavaScript (Node.js) and another in the currently available HTTP/2 library for Lua (lua-http).

Keywords

HTTP/2 Client, Lua Language, Event-driven Programming

Sumário

Lista de Figuras	11
Lista de Códigos de Programas	12
1 Introdução	13
2 Fundamentos Teóricos	15
2.1 HTTP/2	15
2.1.1 Breve história do HTTP/2	17
2.1.2 Enquadramento de Mensagens	18
2.1.3 Fluxos de Mensagens	19
Ciclo de vida de um fluxo	20
Controle de fluxo	21
2.1.4 HPACK: Compressão de Cabeçalhos	23
2.1.5 Priorização de Requisições	24
2.1.6 <i>Push</i> de Requisição/Resposta do Servidor	25
2.2 Lua	26
2.2.1 Tabelas e Funções	27
2.2.2 Co-rotinas	30
2.2.3 Copas	31
2.3 Trabalhos Relacionados	33
3 Implementação	35
3.1 Modelo de Eventos	35
3.2 Modelo de Concorrência	37
3.3 Modelagem do Código-fonte	38
3.3.1 Módulo <i>connection</i>	39
3.3.2 Módulo <i>stream</i>	41
3.3.3 Módulo <i>framer</i>	41
3.3.4 Módulo <i>hpack</i>	41
3.3.5 Módulo <i>http2</i>	42
<i>Dispatcher</i>	43
3.4 API	44
3.5 Considerações Finais	45
4 Resultados Experimentais	47
5 Considerações Finais	50
5.1 Trabalhos Futuros	51

Lista de Figuras

2.1	Uma mensagem de requisição POST HTTP/1.1 e uma mensagem de requisição POST HTTP/2 equivalente.	18
2.2	Formato de um quadro HTTP/2.	19
2.3	O ciclo de vida de um fluxo HTTP/2 ilustrado através de um diagrama com transição de estados.	21
2.4	Dependências e pesos de fluxos HTTP/2 como uma árvore de dependências.	25
3.1	Diagrama de sequência UML que modela a programação orientada a eventos utilizando a biblioteca Copas.	37
3.2	Dependências da biblioteca http2 por outras bibliotecas de Lua.	39
3.3	Componentes da biblioteca http2.	40
4.1	Tempo médio de execução de quatro clientes HTTP/2, onde cada um fez 10 requisições de um arquivo de 1,5 megabytes.	49
4.2	Tempo médio de execução de quatro clientes HTTP/2, onde cada um fez 10 requisições de um arquivo de 10 megabytes.	49
4.3	Tempo médio de execução de quatro clientes HTTP/2, onde cada um fez 10 requisições de um arquivo de 100 megabytes.	49

Lista de Códigos de Programas

2.1	Biblioteca "ponto"	28
2.2	Utilizando a biblioteca "ponto"	28
2.3	Classe	29
2.4	Objeto	29
2.5	Co-rotina	31
2.6	Repetidor Copas	33
3.1	Cliente HTTP/2	42
3.2	Dispatcher	45

Introdução

Redes de computadores revolucionaram o mundo da ciência da computação. A Internet pública, uma rede de computadores específica, já é uma parte indivisível da estrutura das sociedades modernas e a força motriz por trás do seu sucesso se deve aos avanços da Web e do *Hypertext Transfer Protocol* (HTTP). A Web é uma aplicação cliente-servidor que permite usuários obterem páginas Web de servidores sob demanda e seu principal protocolo é o HTTP, um protocolo da camada de aplicação sem estado de requisição/resposta.

Apesar do sucesso da Web e do HTTP, requisitar páginas Web é uma tarefa cada vez mais intensiva em recursos de rede. Com o advento das aplicações Web, do número de usuários, da velocidade dos enlaces e da capacidade da rede como um todo, a adição de mais recursos como JavaScript, CSS, imagens e fontes às páginas Web aumentaram substancialmente e atualmente essa prática é considerada aceitável.

Nos últimos sete anos e seis meses, o tamanho (em kilobytes) de todos os recursos de uma página Web requisitado por um computador pessoal aumentou em aproximadamente três vezes e aqueles requisitados por um dispositivo móvel aumentou em aproximadamente nove vezes [11]. Transportar eficientemente todos esses recursos na rede é uma tarefa cada vez mais difícil.

Uma das causas dessa dificuldade está relacionada ao modo como o HTTP utiliza o protocolo da camada de transporte TCP: cada requisição por uma página Web usa uma conexão TCP separada. Com muitas requisições sendo enviadas, o aumento do número de conexões TCP gera uma monopolização dos recursos de rede. A outra causa está presente nas próprias mensagens HTTP: cada mensagem é enviada em texto simples na conexão TCP, contendo muitos cabeçalhos repetitivos e extensos.

Para resolver ambos esses problemas, uma nova versão do protocolo HTTP foi padronizada, denominada *Hypertext Transfer Protocol version 2* (HTTP/2) [3]. Hoje, 46.5% dos 1 milhão de sites mais acessados do mundo recebem requisições HTTP/2 [11]. Nos navegadores, o HTTP/2 é suportado pelas últimas versões do Google Chrome, Mozilla Firefox, Microsoft Edge, Opera e Safari.

Além dos navegadores, requisições também são feitas por clientes HTTP/2

simples, implementados em uma linguagem de programação específica. Entre essas linguagens, destacam-se as linguagens dinâmicas. Geralmente, essas linguagens oferecem bibliotecas HTTP/2 reusáveis para a implementação de clientes que fazem as requisições para os servidores.

Usar bibliotecas promove mais modularidade aos códigos das aplicações e mais reusabilidade de código aos programadores, além de permitirem a escrita de códigos altamente genéricos. Com isso, clientes HTTP/2 podem ser implementados de modo independente desde que uma API (*Application Programming Interface*) bem definida seja fornecida pela biblioteca.

Entre as linguagens dinâmicas, a linguagem Lua [13] tem se destacado não só pela flexibilidade em criar bibliotecas, mas também pela sua alta eficiência. Essa combinação a torna uma linguagem interessante para implementar o lado cliente do HTTP/2, visto que esse protocolo demanda eficiência por parte das implementações para que seus benefícios de desempenho às aplicações sejam mais bem aproveitados.

Lua também foi criada com portabilidade em mente porque foi implementada em ANSI (ISO) C e é pequena em tamanho. A biblioteca HTTP/2 atual [17] disponível para Lua não segue esse objetivo porque ela só pode ser usada em sistemas operacionais derivados do UNIX e portanto não é uma biblioteca portátil multiplataforma. Ela é uma biblioteca externa e não faz parte do conjunto de bibliotecas padrões de Lua.

O propósito deste trabalho é apresentar uma implementação de uma biblioteca cliente HTTP/2 multiplataforma escrita em Lua. O intuito é fornecer suporte HTTP/2 para que aplicações Web escritas em Lua aproveitem os benefícios de desempenho trazidos por esse protocolo. Para isso, foi implementado uma biblioteca cliente capaz de prover uma API simples de um cliente HTTP/2 que pode ser utilizada em qualquer aplicação Web escrita em Lua 5.3.

Testes de carga (*benchmarks*) foram feitos com o objetivo de comparar o desempenho da implementação. Comparamos um cliente escrito em cima da biblioteca criada com outros três clientes HTTP/2: um em C (nghttp), um em JavaScript (Node.js) e outro na biblioteca HTTP/2 atualmente disponível para Lua (lua-http). Pelos resultados obtidos, o cliente escrito na biblioteca HTTP/2 deste trabalho obteve desempenho igual ou superior em relação ao desempenho dos demais clientes, demonstrando assim sua eficiência.

A estrutura e a organização deste trabalho foram delineadas da seguinte forma: o Capítulo 2 descreve as ferramentas conceituais do protocolo HTTP/2 e da linguagem Lua que viabilizaram a escrita da implementação; o Capítulo 3 apresenta a implementação, incluindo as decisões de projeto tomadas para orientar seu desenvolvimento e os problemas resolvidos; o Capítulo 4 apresenta os testes e comparações realizadas para comprovar a eficiência da implementação da biblioteca e o Capítulo 5 apresenta as considerações finais sobre este trabalho.

Fundamentos Teóricos

Este capítulo apresentará uma visão geral dos fundamentos teóricos que permitiram a implementação da biblioteca cliente HTTP/2 em Lua. Conceituar o HTTP/2 e a linguagem Lua é de suma importância para sustentar as ideias da implementação. Para isso, o capítulo foi organizado em duas seções: uma seção dedicada à descrição do protocolo HTTP/2 e outra à linguagem Lua. Tanto o HTTP/2 quanto Lua são assuntos naturalmente extensos e portanto cada seção consiste de um breve resumo de conceitos que foram pertinentes à escrita da implementação.

A seção 2.1 apresenta uma breve história do HTTP/2 e a natureza dos problemas de desempenho presentes no HTTP, destacando o escopo desses problemas, como eles se manifestam e como estudos passados tentaram resolvê-los. Depois, um breve resumo do protocolo HTTP/2 é apresentado. A completa definição do HTTP/2 é um produto dos RFCs 7540 [3] e 7541 [23] e podem ser consultados para informações mais detalhadas do protocolo.

A seção 2.2 apresenta as principais características da linguagem de programação Lua, juntamente com as bibliotecas escritas em Lua que permitiram uma implementação compatível com a natureza de um cliente HTTP/2. Como a implementação da biblioteca foi feita em Lua, essa seção destaca os principais paradigmas e construções da linguagem que viabilizaram a escrita da biblioteca. Todos os aspectos da linguagem Lua são cobertos pelo livro *Programming in Lua* [13].

2.1 HTTP/2

O *Hypertext Transfer Protocol version 2* (HTTP/2) é um protocolo da camada de aplicação sem estado de requisição/resposta para sistemas de informação hipertextos, distribuídos e colaborativos definido pelos RFCs 7540 [3] e 7541 [23]. O RFC 7540 especifica uma nova sintaxe para o HTTP/2, mas não altera a semântica já existente do HTTP/1.1 e o RFC 7541 define um formato de compressão para campos de cabeçalhos HTTP/2.

Um dos propósitos pelo qual a especificação do HTTP/2 foi criada é o de proporcionar um melhor desempenho às aplicações Web ao permitir que clientes e servidores HTTP/2 façam um uso mais eficiente da conexão TCP [4], onde toda conexão é persistente e as requisições podem ser feitas de forma concorrente. No HTTP/1.1, clientes e servidores têm a opção de usar conexões persistentes ou não persistentes. Porém, quando conexões persistentes são usadas no HTTP/1.1, cada requisição é feita de forma sequencial, ou seja, requisições devem esperar por cada resposta do servidor.

Para que as requisições não precisem esperar por cada resposta, o HTTP/1.1 permitiu paralelismo (*pipelining*) de requisições, mas uma requisição que necessita de um processamento significativo no lado do servidor pode resultar em uma resposta grande e lenta, podendo bloquear outras requisições. Esse problema é conhecido como bloqueio de cabeça de fila (HOL – *head-of-the-line blocking*) [16]. Além disso, alguns servidores e intermediários não implementam paralelismo corretamente, ao ponto dos principais navegadores em uso atualmente terem desabilitado *pipelining* por padrão.

Embora na prática um número significativo de conexões TCP com o servidor podem existir [18], realizar um número reduzido de conexões TCP é uma prática importante para diminuir o impacto negativo que o HTTP e o HTTPS têm sobre a rede. Por exemplo: o tempo de viagem de ida e volta (*round-trip time* – RTT) de pacotes de rede é reduzido porque a apresentação de três vias do TCP não precisa ser renegociada depois que cada requisição termina; menos pacotes são enviados na rede; existem menos chances do pacote SYN do TCP ser perdido; o reuso de uma sessão TLS é melhorado e menos *handshakes* TLS são feitos. Além disso, usar muitas conexões injustamente monopoliza recursos de rede por parte dos navegadores, por exemplo.

Um segundo propósito pelo qual o HTTP/2 foi criado está relacionado a composição das mensagens HTTP: campos de cabeçalhos redundantes e repetitivos (alguns com tamanho consideravelmente grande) são enviados pela conexão TCP em texto simples. Essa prática causa um tráfego de rede desnecessário e uma rápida negação do controle de congestionamento do TCP, levando a eventos de congestionamento e de retransmissão que prejudicam não só a aplicação, mas a rede como um todo.

O terceiro propósito da criação especificação do HTTP/2 refere-se a preservação da compatibilidade com os atuais usos do HTTP/1.1. Com o intuito de interoperar com a Web existente, o HTTP/2 destina-se a ser tão compatível quanto possível com o HTTP/1.1. Isso significa que, pela perspectiva da aplicação, as características do HTTP/1.1 foram em grande parte inalteradas. Os desenvolvedores não precisam modificar conteúdos de seus sites e as mesmas APIs do HTTP/1.1 ainda podem ser utilizadas.

Para alcançar isso, todas as semânticas de requisição e resposta do HTTP/1.1 [9] foram preservadas, mas uma nova sintaxe foi proposta para o HTTP/2. Apesar da semântica não ter sido alterada, um mapeamento otimizado das semânticas do HTTP/1.1

foi definido de forma a fazer um melhor uso da conexão TCP. Quanto à sintaxe de mensagens do HTTP/1.1 [8], ela não foi substituída, mas foi alterada para refletir o novo enquadramento de mensagens do HTTP/2. Isso significa que clientes e servidores devem implementar a nova sintaxe para que a comunicação HTTP/2 seja bem sucedida.

Além dos três propósitos descritos, novos mecanismos como *push* do servidor (ver Seção 2.1.6) e priorização de requisições (ver Seção 2.1.5) foram adicionados ao HTTP/2, todos visando o desempenho das aplicações. Em relação ao HTTP/1.1, o HTTP/2 acrescentou as seguintes funcionalidades: intercalação de mensagens de requisição e resposta na mesma conexão; codificação eficiente para os cabeçalhos de mensagens; priorização de requisições, fazendo com que requisições mais importantes terminem mais rápido e um processamento de mensagens mais eficiente através de um enquadramento binário de mensagens (ver Seção 2.1.2).

2.1.1 Breve história do HTTP/2

O precursor do HTTP/2 foi um protocolo experimental da camada de aplicação chamado SPDY¹ [2]. Desenvolvido principalmente pela Google, o SPDY tinha como objetivo central reduzir o tempo para carregar páginas Web em 50% enquanto mantinha compatibilidade com a semântica do HTTP/1.1. Em 2009, quando o SPDY foi anunciado, engenheiros da Google publicaram o código fonte de um cliente SPDY no navegador Google Chrome e o protótipo de um servidor SPDY, bem como a documentação do protocolo, compartilhando um resultado preliminar bastante promissor: em relação ao HTTP/1.x, as páginas carregaram com velocidade de até 60% mais rápidas sobre uma conexão TCP simples e até 55% mais rápidas sobre uma conexão SSL [1].

Esses resultados chamaram bastante atenção por parte da indústria. Além do Google Chrome, os principais navegadores também implementaram o lado cliente do SPDY: em 2009, o Mozilla Firefox; em 2012, o Opera e em 2013, o Internet Explorer. No lado servidor, grandes sites como o Google, Facebook, Twitter e Yahoo adotaram o SPDY em suas infraestruturas. Em 2012, os servidores HTTP Apache e Nginx também implementaram o lado servidor do SPDY, permitindo que sites mais pequenos suportassem o SPDY. Apesar de trabalhos como [26] apontarem que os benefícios do SPDY podem, paradoxalmente, prejudicar o tempo de carregamento de páginas Web, a utilização comercial desse protocolo não parou.

Em 2012, diante da grande tendência de adoção do SPDY por parte da indústria e com base nas experiências do desenvolvimento do SPDY, a IETF anunciou que daria início a um novo padrão, denominado HTTP/2, usando a documentação do SPDY como

¹O nome “SPDY” é uma marca comercial da Google e não é um acrônimo, visto que é pronunciado como “SPeeDY”, em inglês.

base para a escrita da especificação. Como resultado, os desenvolvedores ganharam experiências realistas com o HTTP/2 antes mesmo do protocolo ter sido aprovado porque dezenas de implementações de clientes e servidores SPDY já estavam extensivamente testados e em produção.

As decisões de projeto que foram tomadas para o SPDY foram [2]: permitir o envio de múltiplas requisições concorrentes dentro de uma única conexão TCP; realizar compressão de cabeçalhos redundantes e repetitivos; definir um protocolo que seja compatível com versões anteriores; permitir que servidores enviem múltiplas respostas contendo recursos associados a uma única requisição, sem que clientes tenham que requisitar cada recurso explicitamente e permitir que clientes atribuam prioridades para cada requisição. De fato, essas decisões foram precisas porque prevaleceram no HTTP/2.

2.1.2 Enquadramento de Mensagens

As mensagens (de requisição e resposta) HTTP eram constituídas por texto simples que usavam caracteres delimitadores de fim de linha. Esse tipo de sintaxe textual não é eficiente para processar e é difícil de implementar corretamente porque ela permite espaços em branco opcionais, sequências variadas de terminação e outras peculiaridades que levam a uma dificuldade em distinguir o cabeçalho do corpo da mensagem. Além disso, essa sintaxe possibilita a criação de vulnerabilidades de segurança devido a várias maneiras de processar sequências de caracteres inválidas.

Diferentemente do mecanismo textual de codificação de mensagens HTTP/1.x, o protocolo HTTP/2 codifica mensagens em um formato binário para que sejam enviadas pela conexão TCP de modo mais eficiente. Conforme ilustrado na Figura 2.1, o enquadramento de mensagens consiste em mapear as mensagens textuais HTTP/1.x em mensagens HTTP/2 equivalentes através de unidades denominadas quadros.

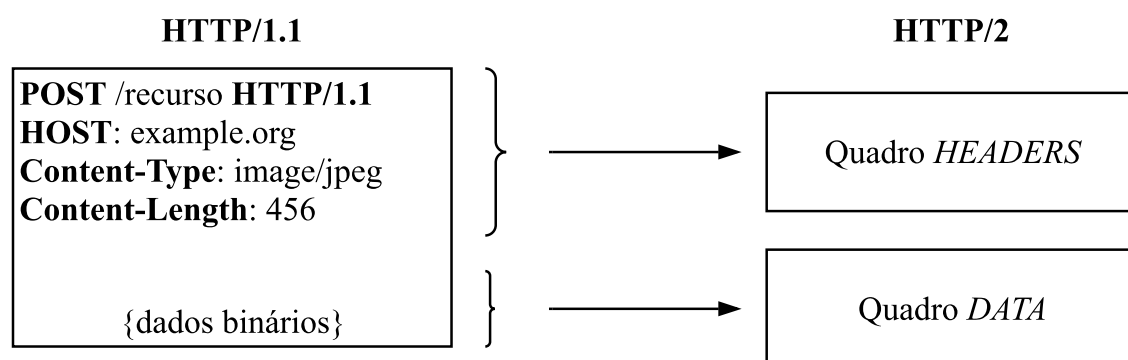


Figura 2.1: Uma mensagem de requisição POST HTTP/1.1 e uma mensagem de requisição POST HTTP/2 equivalente.

A especificação do HTTP/2 define um quadro como sendo a menor unidade de comunicação entre clientes e servidores. Um ou mais quadros podem ser enviados

e recebidos tanto por clientes quanto servidores assim que o cliente inicia uma conexão TCP. Cada quadro é composto por um cabeçalho de tamanho fixo de 9 bytes seguido por um corpo de tamanho variado que depende do tipo do quadro.

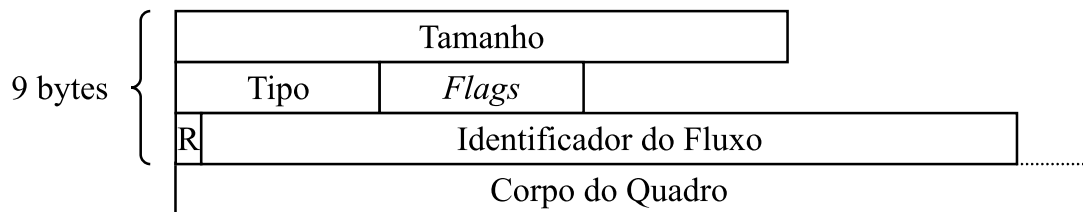


Figura 2.2: *Formato de um quadro HTTP/2.*

O formato de um quadro é mostrado na Figura 2.2. O campo referente ao corpo do quadro varia em estrutura e conteúdo conforme o tipo do quadro. Os campos do cabeçalho do quadro são definidos como:

- **Tamanho** (24 bits): carrega o tamanho do corpo do quadro (até $2^{24} - 1$ bytes).
- **Tipo** (8 bits): determina a estrutura e o conteúdo do corpo do quadro.
- **Flags** (8 bits): valor de 8 bits com significado específico para o tipo do quadro.
- **R** (1 bit): campo reservado que sempre possui valor 0.
- **Identificador do Fluxo** (31 bits): identifica unicamente um fluxo (ver Seção 2.1.3).

Os seguintes tipos de quadros são definidos, cada um possuindo um papel distinto na comunicação HTTP/2:

- **DATA**: transporta dados do corpo da mensagem.
- **HEADERS**: transporta campos de cabeçalhos da mensagem.
- **PRIORITY**: especifica a prioridade de um fluxo.
- **RST_STREAM**: permite o fechamento de um fluxo.
- **SETTINGS**: transporta características de clientes e servidores para a conexão.
- **PUSH_PROMISE**: sinaliza uma promessa para servir o recurso referenciado.
- **PING**: mede o RTT e verifica se uma conexão ociosa pode ser usada.
- **GOAWAY**: sinaliza o fechamento da conexão ou erros críticos.
- **WINDOW_UPDATE**: permite implementação de controle de fluxo.
- **CONTINUATION**: continua uma sequência de campos de cabeçalhos.

2.1.3 Fluxos de Mensagens

Um fluxo é definido como uma sequência independente e bidirecional de quadros dentro de uma conexão TCP e é identificado por um valor inteiro de 31 bits sem sinal. Tanto o cliente quanto o servidor podem criar e fechar um fluxo unilateralmente ou

compartilhadamente entre si. Porém, clientes devem sempre criar um novo fluxo com valor *ímpar* maior que o valor do fluxo mais recentemente criado e servidores devem sempre criar um novo fluxo com valor *par* maior que o valor do fluxo mais recentemente criado. O fluxo de identificador 0 é reservado para controlar a conexão e não pode ser usado.

Clientes e servidores processam quadros de um fluxo na ordem em que são recebidos dentro da conexão TCP, podendo intercalar quadros de múltiplos fluxos concorrentemente abertos. Dessa forma, clientes e servidores são permitidos realizarem multiplexação de requisições e respostas associando cada troca de mensagens com um único fluxo.

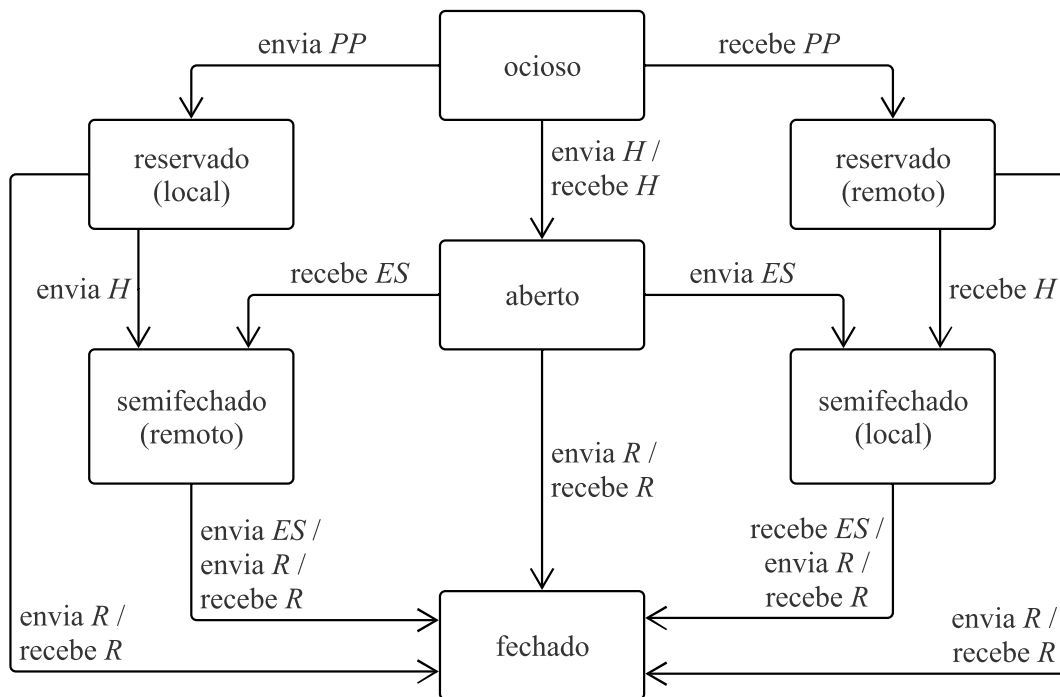
Partindo do conhecimento da estrutura de um quadro, um fluxo pode ser examinado para que diferentes tipos de quadros sejam identificados, cada um com diferentes tamanhos. Como o tamanho é conhecido, o próximo quadro do fluxo pode ser processado mais eficientemente. Ao processar o cabeçalho do quadro, o seu corpo pode ser facilmente interpretado com base no seu tipo. Com isso, mensagens HTTP/2 são mais eficientes de serem processadas e mais compactas em transporte porque são binárias; mais robustas porque são menos propensas a erros; e mais simples porque existe apenas uma forma de processar uma mensagem.

Ciclo de vida de um fluxo

O ciclo de vida de um fluxo é exibido no diagrama da Figura 2.3. As transições causam a mudança de um estado para outro de acordo com os quadros e *flags* enviados ou recebidos por clientes e servidores. Pressupõe-se que essas transições sejam geradas apenas pelo envio ou recebimento de quadros e *flags* ilustrados no diagrama. Por exemplo, não ocorre uma transição de estados quando quadros do tipo *CONTINUATION* seguem os quadros *HEADERS* ou *PUSH_PROMISE* ou quando quadros do tipo *DATA* são enviados no estado aberto. Os estados são definidos como se segue:

- ocioso: estado inicial de um fluxo.
- reservado (local): um cliente ou servidor *local* reservou um fluxo ocioso enviando um quadro do tipo *PUSH_PROMISE*.
- reservado (remoto): um cliente ou servidor *remoto* reservou um fluxo ocioso enviando um quadro do tipo *PUSH_PROMISE*.
- aberto: estado onde clientes e servidores podem enviar quadros de qualquer tipo.
- semifechado (local): estado onde apenas quadros *WINDOW_UPDATE*, *RST_STREAM* e *PRIORITY* podem ser *enviados*, porém qualquer quadro pode ser *recebido*.

- semifechado (remoto): estado onde apenas quadros *WINDOW_UPDATE*, *RST_STREAM* e *PRIORITY* podem ser *recebidos*, porém qualquer quadro pode ser *enviado*.
- fechado: indica o estado terminal de um fluxo, permitindo apenas o envio de quadros *PRIORITY*.



envia: cliente ou servidor envia este quadro
 recebe: cliente ou servidor recebe este quadro

H: quadro *HEADERS*
PP: quadro *PUSH_PROMISE*
ES: *flag END_STREAM*
R: quadro *RST_STREAM*

Figura 2.3: O ciclo de vida de um fluxo HTTP/2 ilustrado através de um diagrama com transição de estados.

Controle de fluxo

Se vários fluxos multiplexados (que carregam mensagens de requisição/resposta) estão sendo enviados pela conexão TCP, mas apenas um deles está entregando quadros, ele bloqueia outros fluxos. Visando resolver esse problema, um controle de fluxo é necessário para impedir que um fluxo interfira com outros fluxos, permitindo usá-los de modo mais eficiente. Para isso, a especificação do HTTP/2 define sete princípios de controle de fluxo, tais quais foram extraídos e traduzidos diretamente do documento RFC 7540 [3]:

1. O controle de fluxo é específico para uma conexão. Os dois tipos de controle de fluxo acontecem entre clientes e servidores em um único salto e não em todo o caminho de fim a fim.
2. O controle de fluxo é implementado com o uso de quadros *WINDOW_UPDATE* e é baseado em um esquema de créditos, já que cada destinatário (cliente ou servidor que está recebendo quadros) anuncia sua janela inicial (em bytes) que ele está preparado para receber em um fluxo e para toda a conexão. Essa janela é reduzida sempre que um remetente envia um quadro do tipo *DATA* e incrementado via quadros *WINDOW_UPDATE* enviados pelo destinatário.
3. O controle de fluxo é direcional com o controle geral fornecido pelo destinatário. Um destinatário pode optar por definir qualquer tamanho de janela que desejar para cada fluxo e para toda a conexão. Um remetente deve respeitar os limites de controle de fluxo impostos por um destinatário. Clientes, servidores e intermediários anunciam, independentemente, sua janela de controle de fluxo como um destinatário e cumprem os limites de controle de fluxo definidos por destinatários quando estiverem atuando como remetentes.
4. O valor inicial da janela de controle de fluxo é de 65.535 bytes tanto para novos fluxos quanto para a conexão em geral.
5. O tipo de quadro determina se o controle de fluxo se aplica a um quadro. Somente os quadros do tipo *DATA* estão sujeitos ao controle de fluxo; todos os outros tipos de quadro não consomem espaço na janela de controle de fluxo anunciada. Isso garante que estruturas de controle importantes não sejam bloqueadas pelo controle de fluxo.
6. O controle de fluxo não pode ser desativado.
7. O HTTP/2 define apenas o formato e a semântica do quadro *WINDOW_UPDATE*. A especificação não estipula como um destinatário decide quando enviar esse quadro ou o valor que ele envia, nem como um remetente escolhe enviar pacotes. As implementações são capazes de selecionar qualquer algoritmo que atenda às suas necessidades.

Ainda de acordo com a especificação, as implementações também são responsáveis por gerenciar como as requisições e as respostas são enviadas com base na prioridade, escolhendo como evitar o bloqueio de requisições e gerenciar a criação de novos fluxos. As escolhas de algoritmo para estas poderiam interagir com qualquer algoritmo de controle de fluxo. Os remetentes sempre estão sujeitos a respeitar a janela de controle de fluxo anunciada pelo destinatário, mas destinatários que não desejarem controle de fluxo podem anunciar uma janela de tamanho máximo ($2^{31} - 1$) e podem manter essa janela enviando um quadro *WINDOW_UPDATE* quando qualquer quadro *DATA* é recebido.

2.1.4 HPACK: Compressão de Cabeçalhos

Para toda requisição/resposta do HTTP/1.1, os mesmos cabeçalhos extensos e redundantes são enviados e recebidos durante uma conexão TCP. Isso resulta em problemas de congestionamento de rede. Para resolver isso, um padrão de compressão de cabeçalhos HTTP/2 para um formato binário foi aprovado pela IETF, denominado HPACK [23].

A especificação do HTTP/2 define listas de cabeçalhos como zero ou mais campos de cabeçalhos. Campos de cabeçalhos são os tradicionais nomes com um ou mais valores associados. Essas listas são serializadas em um bloco de cabeçalhos utilizando os algoritmos definidos em HPACK. Esse bloco de cabeçalhos é então dividido em uma ou mais cadeias de bytes e carregadas no corpo de quadros *HEADERS*, *PUSH_PROMISE* ou *CONTINUATION*. Um cliente ou um servidor que está recebendo essas cadeias de bytes os concatenam e depois descomprimem o bloco para reconstruir a lista decodificada. Os principais elementos do HPACK são os seguintes [27]:

- Campo de cabeçalhos.
- Bloco de cabeçalhos.
- Tabela estática.
- Tabela dinâmica.
- Codificador.
- Decodificador.

Um campo de cabeçalho é um par (nome, valor), que são uma sequência de bytes. Campos de cabeçalhos codificados são representados ou como um valor numérico (índice) ou como um valor literal. Na representação de índice, existe uma referência para uma entrada na tabela estática ou na tabela dinâmica. Na representação literal, existe um par (nome, valor) onde um nome ou é representado literalmente ou é uma referência para uma entrada na tabela estática ou na tabela dinâmica e um valor é representado literalmente.

Um bloco de cabeçalho é uma concatenação de representações de campos de cabeçalhos descritos no parágrafo anterior. Um decodificador pode processar um bloco de cabeçalho sequencialmente para reconstruir a lista de cabeçalhos original.

Uma tabela estática é uma estrutura de dados do tipo lista e é predefinida em [23] com uma lista estática de campos de cabeçalhos, podendo ser referenciada por um índice cujos valores são pares (nome do cabeçalho, valor do cabeçalho). Por exemplo, o índice 2 possui o par (":method", "GET") e o índice 19 possui o par ("accept", ""), sem valor do cabeçalho. A tabela estática associa estaticamente campos de cabeçalhos que ocorrem frequentemente para índices. Trata-se de uma tabela ordenada, de apenas leitura (*read-only*), é sempre acessível e compartilhada em todos os contextos de codificação e decodificação.

Uma tabela dinâmica é uma estrutura de dados similar à tabela estática, com a diferença de que as entradas são criadas dinamicamente. Ela tem um limite superior ao seu tamanho e se a criação de novas entradas ultrapassar esse tamanho, as entradas mais antigas são removidas. Clientes e servidores mantêm uma tabela dinâmica para o envio e outra para o recebimento de campos de cabeçalhos (correspondem a contextos de codificação e decodificação, respectivamente). Elas operam seguindo o princípio FIFO (*First In, First Out*) e associam campos de cabeçalhos para índices e é específica para um contexto de codificação/decodificação. As tabelas dinâmicas podem ser usadas pelo codificador para indexar campos de cabeçalhos redundantes e é permitido que ela contenha entradas duplicadas.

O codificador é implementado com o auxílio das tabelas estáticas e dinâmicas que mapeiam campos de cabeçalhos para índices. Sua função é atualizar a tabela dinâmica e ordenar os campos de cabeçalho no bloco de cabeçalho de acordo com a ordem original da lista de cabeçalho. Na verdade, existem três métodos de compressão utilizados pelo codificador para codificar campos de cabeçalhos: tabelas estáticas, tabelas dinâmicas e código de Huffman estático. Já o decodificador compartilha as modificações prescritas pelo codificador e mantém uma tabela dinâmica como um contexto de decodificação para descomprimir blocos de cabeçalhos.

2.1.5 Priorização de Requisições

Priorizar uma requisição é permitir que clientes informem a quantidade de recursos (como CPU, memória e taxas de transmissão de dados) que devem ser alocados a essa requisição em relação a outras requisições. Visto que vários fluxos carregando requisições podem estar concorrentemente abertos em uma comunicação HTTP/2, clientes podem atribuir informações de priorização para um fluxo específico quando ele é aberto (enviando um quadro do tipo *HEADERS*) e em qualquer momento da comunicação essa prioridade pode ser alterada enviando um quadro do tipo *PRIORITY*.

Para isso, o HTTP/2 permite que dependências por um fluxo sejam associados a um outro fluxo que depende do seu término. Caso dois ou mais fluxos sejam dependentes do término do mesmo fluxo, um peso (um número inteiro entre 1 e 256) pode ser atribuído para cada fluxo dependente para informar que os recursos disponíveis devem ser alocados em proporção aos seus pesos.

O mecanismo de dependências e pesos de fluxos para permitir priorização pode ser ilustrado através de uma árvore de dependências. Por exemplo, considere a árvore de dependências da Figura 2.4. Cada nó representa um fluxo, sendo que o número fora dos parênteses é o seu identificador e o número dentro dos parênteses é o seu peso.

Todo fluxo que não tem dependência declarada depende do fluxo 0 (conforme

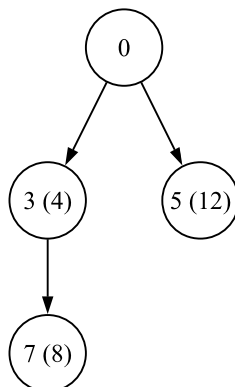


Figura 2.4: Dependências e pesos de fluxos HTTP/2 como uma árvore de dependências.

visto na subseção 2.1.3, um fluxo de valor 0 é reservado para a conexão), que forma a raiz da árvore. Sendo assim, os fluxos 3 e 5 dependem do fluxo 0 e o fluxo 7 depende do fluxo 3. Como os fluxos 3 e 5 são dependentes do mesmo fluxo, a alocação de recursos é distribuída proporcionalmente em relação aos seus pesos: o fluxo 3 recebe $1/4$ ($4/16$) e o fluxo 5 recebe $3/4$ ($12/16$) dos recursos disponíveis.

2.1.6 Push de Requisição/Resposta do Servidor

Uma página Web típica inclui vários recursos independentes e que precisam ser requisitados separadamente por um cliente HTTP. Por exemplo, o arquivo HTML padrão de um servidor (comumente referido como *index.html*) pode incluir recursos dinâmicos como arquivos JavaScript, CSS e imagens que são necessários para exibir a página corretamente. No HTTP, o cliente precisa requisitar cada um desses recursos separadamente.

No HTTP/2, um servidor pode enviar várias respostas para uma única requisição de um cliente, ou seja, um servidor pode fazer um *push* (forçar) o envio de duas ou mais respostas contendo recursos associados a requisição original por um recurso, sem que clientes tenham que requisitar cada recurso separadamente. Assim, o servidor pode enviar antecipadamente todos os recursos dinâmicos contidos em uma página, sem que o cliente precise requisitar cada um.

O *push* do servidor é dividido em *push* de requisição e *push* de resposta. No *push* de requisição, o servidor envia um quadro do tipo *PUSH_PROMISE* junto com campos de cabeçalhos (assim como nos cabeçalhos de uma requisição normal de um cliente) no estado ocioso de um fluxo, indicando que esse fluxo será reservado para um futuro *push* de resposta. Depois de enviado esse quadro, o servidor começa a enviar respostas assim que uma requisição tenha sido explicitamente feita pelo cliente.

É importante observar que o cliente pode desabilitar o *push* do servidor e que,

por medidas de segurança, o servidor precisa ser autoritativo pelas respostas enviadas. O cliente também pode optar por recusar um *push* do servidor enviando um quadro do tipo *RST_STREAM*. Assim, o cliente tem completo controle sobre como o *push* do servidor é usado.

2.2 Lua

A linguagem de programação Lua [12, 13, 15] foi criada no início dos anos 1990 por Roberto Ierusalimschy, Waldemar Celes e Luis Henrique de Figueiredo na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). É uma linguagem de *scripting* multiparadigma que oferece suporte principalmente ao paradigma de programação procedural, embora também seja usada em programação concorrente, programação orientada a objetos, programação funcional, entre outros paradigmas. Lua também oferece suporte à descrição de dados, no estilo de JSON ou XML.

Atualmente, Lua é utilizada em várias aplicações industriais, com uma ênfase em sistemas embarcados e jogos, sendo a linguagem de *scripting* mais utilizada em jogos do mundo. Ela também é considerada uma das linguagens de *scripting* de melhor desempenho do mundo, devido a sua implementação eficiente.

Lua tem tipagem dinâmica, estruturas de dados dinâmicas, gerenciamento automático de memória e outras construções de linguagem similares às de outras linguagens de *scripting* dinamicamente tipadas. O que separa Lua unicamente dessas linguagens é o seu projeto, que foi definido em função dos seguintes objetivos:

- Extensibilidade: Lua é uma biblioteca escrita em C e possui uma API C usada tanto para ser estendida com novas características quanto para estender outras aplicações escritas em C. De fato, a extensibilidade impactou fortemente o projeto de Lua [14].
- Simplicidade: Lua é uma linguagem simples e pequena. Ao invés de ter incontáveis construções de linguagem feitas para resolver um determinado problema, Lua possui poucos (mas flexíveis) conceitos que atendem diferentes necessidades.
- Eficiência: A implementação de Lua é pequena em tamanho. Isso a torna uma implementação muito eficiente e portátil. Como Lua é leve e eficiente, uma aplicação que embute Lua não é sobrecarregada.
- Portabilidade: Lua pode ser executada em praticamente todas as plataformas existentes por ser escrita em ANSI (ISO) C. Por ser pequena, Lua executa facilmente nessas plataformas sem que elas sejam negativamente impactadas.

2.2.1 Tabelas e Funções

Lua é bastante reconhecida por sua simplicidade em harmonia com sua flexibilidade de programação. Muitas construções da linguagem são criadas a partir de duas construções mais básicas: tabelas e funções. Ambas construções permeiam toda a linguagem e formam a base para mecanismos mais complexos como módulos e programação orientada a objetos.

A tabela é uma estrutura de dados dinâmica e constitui a única estrutura de dados de Lua. As tabelas de Lua são semelhantes aos vetores associativos de Perl e PHP, que são compostas por pares (chave, valor). É fácil pensar em uma tabela como um *map* ou como uma tabela *hash* por associarem uma chave a um valor. Tabelas são usadas para implementar as mais diversas estruturas de dados (tais como vetores, registros, listas, filas, pilhas e conjuntos) de modo eficiente e são as construções que permitem criar outras construções mais complexas da linguagem. Logo, tabelas permeiam toda a linguagem Lua.

Lua suporta diferentes mecanismos para construir tabelas. A expressão `t["x"] = 1` cria uma tabela `t` onde a chave "x" é associada ao valor 1 e a expressão `t[1] = "x"` cria uma tabela `t` associando a chave 1 ao valor "x". Utilizando o chamado “açúcar sintático” de Lua, a primeira expressão pode ser escrita como `t.x = 1`. Utilizando a sintaxe de construtor {}, o construtor `{x = 1, y = 2}` cria uma tabela com duas entradas, uma associando "x" ao valor 1 e outra associando "y" ao valor 2 [15].

As funções de Lua têm funcionalidades tradicionais às de outras linguagens de programação, mas com o diferencial de que elas são valores de primeira classe, isto é, uma função pode ser armazenada em variáveis, passada como argumento para parâmetros de outras funções e ser retornada como resultado dessas outras funções. Lua também oferece suporte a funções com escopo léxico (*closures*), habilitando Lua a ser utilizada para programação funcional. O código abaixo mostra uma função anônima sendo atribuída a variável `mod`:

```
mod = function(x, y)
    return x % y
end
```

Tabelas e funções de primeira classe podem ser usados para implementar módulos em Lua (semelhantes a pacotes em Java e Perl, ou *namespaces* em C++), que servem como bibliotecas ou como um espaço de nomes para a linguagem. Por exemplo, considere o Código 2.1, que mostra como uma biblioteca simples em Lua que permite criar pontos (coordenadas) e calcular a distância entre dois pontos pode ser criada. Um programador pode (re)usar essa biblioteca usando a função `require` da biblioteca padrão (Código 2.2).

Código 2.1 Biblioteca "ponto"

```
1  local ponto = {}  
2  
3  function ponto.new(x, y)  
4      return {x = x, y = y}  
5  end  
6  
7  function ponto.dist(p1, p2)  
8      local dx = (p2.x - p1.x)^2  
9      local dy = (p2.y - p1.y)^2  
10     local dist = math.sqrt(dx + dy)  
11     return dist  
12 end  
13  
14 return ponto
```

Código 2.2 Utilizando a biblioteca "ponto"

```
1  local ponto = require "ponto"  
2  local p1 = ponto.new(1, 1)  
3  local p2 = ponto.new(1, 4)  
4  print(ponto.dist(p1, p2)) --> 3.0
```

A função `require` mostrada na linha 1 do Código 2.2 é um exemplo de uma das funções fornecidas pela biblioteca padrão de Lua. Além dessa função, a função `math.sqrt` é exibida na linha 10 do Código 2.1, que faz parte da biblioteca padrão `math` de Lua. A biblioteca “ponto” é um exemplo de uma biblioteca externa que pode ser criada por programadores Lua como forma de estender as funcionalidades da linguagem. Elas não vêm inclusas no conjunto de bibliotecas padrões de Lua e devem ser incluídas manualmente no código caso o programador deseje usá-la.

Tabelas e funções também podem ser utilizadas para simular programação orientada a objetos. Conceitos de orientação a objetos como classes, objetos, métodos e herança podem ser representados por tabelas e funções, seguindo o padrão de utilizar mecanismos básicos da linguagem para construir mecanismos mais complexos.

Por exemplo, considere o Código 2.3, onde uma classe nomeada *class* é implementada em Lua. Essa classe faz o uso de dois métodos: `set_value(value)` define um novo valor `value` e `get_value()` recupera esse valor. Um construtor `new` também é definido para instanciar objetos dessa classe. Lua possibilita a criação de classes com o auxílio da função `setmetatable` para instanciar tabelas a partir de outras tabelas e do

metamétodo `__index` para acessar valores dessas tabelas. Além disso, Lua provê o “açúcar sintático” de dois-pontos para criar um parâmetro de autoreferência (normalmente chamado de *this* ou *self* em outras linguagens de programação orientadas a objetos).

O Código 2.4 mostra como um objeto pode ser instanciado a partir de uma classe (representada por um módulo). Na linha 3, o código cria um novo objeto a partir da classe `class` utilizando seu construtor `new`, configura o valor 2 para esse objeto e depois recupera esse mesmo valor chamando os métodos da classe `class`.

Código 2.3 Classe

```
1  local class = {__index = {}}
2
3  local function new(object)
4      local instance = object or {value = 1}
5      local self = setmetatable(instance, class)
6      return self
7  end
8
9  function class.__index:set_value(v)
10     self.value = v
11 end
12
13 function class.__index:get_value()
14     return self.value
15 end
16
17 local class = {new = new}
18
19 return class
```

Código 2.4 Objeto

```
1  local class = require "class"
2
3  local object = class.new()
4  object:set_value(2)
5  local value = object:get_value()
6  print(value)
```

2.2.2 Co-rotinas

O suporte à programação concorrente em Lua é fornecido por co-rotinas e funcionam como *multithreading* cooperativa: apenas uma co-rotina (*thread*) está em execução por vez, e a troca de contexto entre as co-rotinas é feita de forma explícita. Diferentemente de *multithreading* preemptiva, onde a troca de contexto entre as *threads* ocorre de forma implícita, na cooperativa existem funções explícitas para a transferência de controle.

Co-rotinas são, assim como funções, valores de primeira classe. Outra característica de co-rotinas em Lua é que elas são assimétricas, ou seja, existe uma função para suspender uma co-rotina e outra função para executar uma co-rotina. Isso as distingue de co-rotinas simétricas, onde apenas uma função exerce esse papel.

Uma co-rotina é criada através da biblioteca padrão `coroutine` de Lua. Assim como a biblioteca padrão `table` de Lua, a biblioteca `coroutine` é um módulo nativo da linguagem Lua, consistindo de tabelas e funções. As principais funções que fazem parte do módulo `coroutine` são:

- `coroutine.create`: cria uma co-rotina.
- `coroutine.yield`: suspende uma co-rotina.
- `coroutine.resume`: coloca uma co-rotina em execução.
- `coroutine.status`: verifica o estado de uma co-rotina.

O funcionamento de uma co-rotina pode ser ilustrado através de um exemplo de código. No Código 2.5, que foi adaptado de [15], uma co-rotina é criada na linha 1 pela função `coroutine.create`, passando uma função anônima como seu parâmetro. Essa função apenas cria a co-rotina, sem executá-la. Na linha 2, o valor de `x` passado como parâmetro será 1 porque a função `coroutine.resume` na linha 8 passou esse valor como argumento. Na linha 3, a co-rotina suspende a execução, passando o valor 2 como argumento de `coroutine.yield` para `coroutine.resume` na linha 8 e armazenando esse valor na variável `y`. Depois, o programa retorna a executar a co-rotina, dessa vez passando o valor 3 como argumento de `coroutine.resume` na linha 9 para o retorno de `coroutine.yield` na linha 3. Por fim, a co-rotina causa o término de `coroutine.resume` na linha 9 retornando o valor 4 e armazenando-o na variável `y`.

Código 2.5 Co-rotina

```
1 co = coroutine.create(function(x)
2     --> x == 1
3     x = coroutine.yield(2)
4     --> x == 3
5     return 4
6 end)
7
8 y = coroutine.resume(co, 1) --> y == 2
9 y = coroutine.resume(co, 3) --> y == 4
```

Uma desvantagem em utilizar co-rotinas é que gerenciá-las pode se tornar uma tarefa complexa quando o número de suspensões e retomadas aumentam. Por exemplo, se uma co-rotina invoca uma operação bloqueantes de rede como `receive` e bloqueia, todo o programa é bloqueado até que a operação termine. Esse comportamento pode ser evitado fornecendo um conjunto de funções não bloqueantes que iniciam uma operação de rede e suspende a co-rotina ativa quando a operação não puder ser imediatamente completada.

Jerusalismchy [13] mostra um exemplo de uma aplicação concorrente que utiliza co-rotinas com funções não bloqueantes da API *socket* do LuaSocket [19]. Uma outra forma de evitar esse comportamento indesejável em aplicações concorrentes é criando um escalonador de co-rotinas, que gerencia a tarefa de suspender e executar co-rotinas quando as operações de rede bloquearem. Uma biblioteca em Lua que proporciona essa funcionalidade é chamada Copas, descrita na seção a seguir.

2.2.3 Copas

Copas (*Coroutine Oriented Portable Asynchronous Services for Lua*) [5] é uma biblioteca que oferece suporte a operações assíncronas para aplicações de rede escritas em Lua e seu foco é na criação de clientes e servidores. Funções assíncronas são fornecidas pela biblioteca Copas para impedir que a aplicação de rede fique bloqueada em uma chamada de rede tipicamente bloqueante (por exemplo, na função de recebimento de dados `receive`). Para isso, Copas faz uso da característica de *multithreading* cooperativa de Lua, atuando como um escalonador de co-rotinas.

Copas utiliza a biblioteca LuaSocket [19] para interagir com sua API *socket*. Com funcionamento semelhante às funções `socket.send` e `socket.receive` do LuaSocket, a função `copas.send` envia dados pelo *socket* e a função `copas.receive` recebe dados pelo *socket*. A diferença entre essas funções em ambas as bibliotecas é que em Copas não existem chamadas bloqueantes. O uso de *multithreading* cooperativa de Lua

permite que os envios e recebimentos de dados pelo *socket* sejam tratados de forma simultânea em várias *threads*, desde que elas façam chamadas às funções `copas.send` e `copas.receive` de Copas.

A tarefa de escalonar co-rotinas é realizada pelo escalonador do Copas, onde um *loop* principal fica responsável principalmente por empregar um mecanismo de consulta para verificar se o resultado da computação de uma função não bloqueante de um *socket* (por exemplo, recebimento de dados ou *time out*) já está disponível. Quando o resultado é recebido, o escalonador utiliza co-rotinas para processá-lo concorrentemente com outras co-rotinas contendo resultados de computação de outras funções não bloqueantes. Com isso, a execução assíncrona de uma aplicação de rede é alcançada, pois o uso de funções não bloqueantes de um *socket* em combinação com a facilidade de executá-las concorrentemente com várias *threads* nunca travam o *loop* principal da aplicação. As seguintes funções de Copas implementam esse gerenciamento de co-rotinas:

- `copas.loop`: inicia o *loop* principal do Copas.
- `copas.addthread`: adiciona uma nova co-rotina ao escalonador para que ele possa escalonar essa co-rotina quando o resultado de uma computação assíncrona de rede for recebida.
- `copas.sleep`: assim como a função `coroutine.yield` vista na Subseção 2.2.2, suspende a execução da co-rotina atual (apenas uma pode estar em execução) criada por `copas.addthread`.
- `copas.wakeup`: volta a executar a co-rotina criada por `copas.addthread` e passada como parâmetro para essa função.

Para ilustrar um caso de uso do Copas, considere o Código 2.6, que demonstra um simples uso de um repetidor que transmite de volta os dados recebidos e termina quando recebe `quit`. Para simplificar, o código para criação do *socket* que utiliza a API do LuaSocket foi omitido. Primeiramente, nas linhas 19 e 20, as *threads* `on_quit` e `despachante` são adicionadas ao escalonador do Copas, respectivamente. A *thread* `despachante` é simplesmente um *loop* que recebe dados provenientes de um *socket* e verifica se esses dados correspondem a `quit`, sinalizando o término do envio de dados. A *thread* `on_quit` é uma tratadora dessa sinalização de `quit`, que é despachada quando os dados param de ser enviados.

Depois de adicionadas as *threads*, o *loop* principal do Copas começa a executar, na linha 22. A primeira *thread* que ele escalona é `on_quit`, pois foi a primeira a ser adicionada. Em seguida, na linha 14, essa *thread* é colocada em modo suspenso porque ela só poderá ser despachada quando os dados pararem de serem enviados pelo *socket*, ou seja, quando `quit` é recebido. Assim que a *thread* `on_quit` é suspensa, a *thread* `despachante` é escalonada pelo *loop* principal, que começa a verificar pela sinalização do

término do envio de dados. Caso os dados ainda estejam sendo enviados pelo remetente, esses mesmos dados são enviados para o mesmo, repetindo o ciclo. Quando os dados correspondentes a `quit` são recebidos, a *thread* `on_quit` é despachada na linha 5 e o programa é finalizado, pois não há mais nenhuma co-rotina e nenhum *socket* a serem escalonados pelo *loop* do Copas.

Código 2.6 Repetidor Copas

```
1  function despachante(socket)
2      while true do
3          dados = copas.receive(socket, 6)
4          if dados == "quit" then
5              copas.wakeup(on_quit)
6          else
7              copas.send(socket, dados)
8          end
9      end
10 end
11
12 function quit()
13     -- suspende essa thread até que ela seja escalonada
14     copas.sleep(-1)
15     -- termina o programa
16     os.exit()
17 end
18
19 on_quit = copas.addthread(quit)
20 copas.addthread(despachante, socket)
21
22 copas.loop()
```

2.3 Trabalhos Relacionados

Muitos esforços vêm sendo realizados para implementar bibliotecas clientes HTTP/2. O principal problema enfrentado pelos trabalhos que se propõem a implementar bibliotecas do lado cliente do HTTP/2 é o de construir um escalonador de fluxos. A exigência do HTTP/2 é de que fluxos concorrentes devem ser processados fora de ordem, porém a ordem com que os quadros são transmitidos dentro desses fluxos é relevante [3]. Por esses motivos, os trabalhos de implementação de bibliotecas clientes HTTP/2 necessitam intercalar quadros provenientes de múltiplos fluxos em ordem, finalizar o

processamento de um fluxo e decidir qual será o próximo fluxo a ser processado. Assim, todos os trabalhos revisados neste estudo empregam o modelo de programação concorrente, apesar das diferentes construções de linguagens de programação envolvidas para realizá-lo.

Um segundo problema que as bibliotecas tentam resolver é o modo de implementar o ciclo de vida de um fluxo. Parece que existe um consenso de que o modelo de programação mais apropriado para representar esse ciclo é o de orientação a eventos. Esse modelo de fato traz vantagens sobre a estruturação tanto do código de API de alto nível quanto do código HTTP/2 de baixo nível, pois os estados e as transições do ciclo de vida de um fluxo podem ser programadas em máquinas de estados, um mecanismo muito semelhante à programação orientada a eventos. Nesse contexto, com exceção de uma biblioteca, todos os outros trabalhos revisados neste estudo empregam o modelo de programação orientada a eventos, apesar das diferentes construções de linguagens de programação envolvidas para realizá-la.

A biblioteca HTTP/2 atualmente em uso para Lua (lua-http) [17] depende de uma biblioteca externa denominada cqueues para obter a facilidade de tratar *threads*, sinais e notificações para implementarem concorrência e eventos associados aos fluxos. Ela também faz uso extenso do *loop* principal de cqueues, que é modularizável e que pode ser facilmente controlado externamente. No entanto, o problema da biblioteca cqueues é que ela faz uso de primitivas ao nível do *kernel* do sistema operacional, o que impõe limitações de portabilidade na biblioteca lua-http. Como consequência, essa biblioteca apenas funciona em sistemas operacionais UNIX.

A biblioteca cliente HTTP/2 para Node.js (http2) [21], assim como lua-http, também é completa em termos das especificações de um cliente HTTP/2. Diferentemente de Lua, Node.js é uma tecnologia com um *loop* principal nativo embutido em todo código JavaScript que seja interpretado por Node.js. Essa característica o torna ideal para modelar a programação orientada a eventos e consequentemente implementar o ciclo de vida um fluxo. Além disso, a biblioteca cliente HTTP/2 de Node.js faz uso extenso de chamadas assíncronas para oferecer concorrência.

Por fim, a biblioteca cliente HTTP/2 para C e C++ (nghttp2) [20] mais reconhecida atualmente é uma biblioteca constantemente atualizada, desde o começo da concepção do protocolo HTTP/2. Por isso, trata-se de uma biblioteca usada como referência de implementação por ser muito completa, rápida e compatível com as exigências de um cliente HTTP/2. A biblioteca de alto nível libnghttp2_asio, construída sobre a biblioteca nativa libnghttp2, oferece abstrações em uma API para construir clientes HTTP/2. Também nessas bibliotecas é empregado o modelo de orientação a eventos e de programação concorrente, que depende da biblioteca externa *Boost* de C++ para proporcionar o gerenciamento de *threads* e operações assíncronas de rede.

Implementação

Neste capítulo apresentamos a implementação da biblioteca cliente HTTP/2, denominada `http2`, em termos da capacidade exigida pela especificação do HTTP/2 e com as características dos modelos de programação orientada a eventos e programação concorrente em Lua, realizando os conceitos e ideias apresentados no Capítulo 2. Serão identificadas as restrições da implementação que foram importantes em orientar as decisões de projeto durante o processo de desenvolvimento e o impacto dessas restrições. A implementação também será descrita em um nível maior de detalhes, no qual códigos que são críticos para a operação da biblioteca serão apresentados. Também mostraremos os problemas que surgiram durante a implementação e como lidamos com eles.

3.1 Modelo de Eventos

O modelo de programação orientado a eventos foi adotado na implementação da biblioteca `http2`. Nesse modelo, um *loop* principal denominado *loop de eventos* fica responsável por receber eventos e despachá-los para seus respectivos tratadores, normalmente representadas por funções *callbacks*. Assim que um evento é alcançado, o fluxo de execução é transferido para a *callback*, fazendo com que a execução só retorne ao *loop* quando o processamento da *callback* chegar ao fim ou quando o fluxo da execução ser explicitamente devolvido. Nenhum novo evento será recebido e processado enquanto o *loop* não estiver executando novamente, ou seja, enquanto a *callback* não transferir o controle de volta para o *loop*.

Essa decisão foi tomada porque o diagrama de transição de estados da Figura 2.3 se assemelha bastante com a programação orientada a eventos. Internamente pode existir uma constante mudança no estado do cliente HTTP/2 com base nos fluxos recebidos e processados. Um fluxo se encontra em um estado, que é modificado com a chegada de um novo evento. Esse evento pode ser, por exemplo, o recebimento de um quadro do tipo *HEADERS* ou o envio desse mesmo quadro. Como resultado, o modelo de programação orientada a eventos foi empregado para implementar o ciclo de vida de um fluxo.

Em reflexo do funcionamento do ciclo de vida de um fluxo e pelo fato de cada requisição enviada por um cliente HTTP/2 ser associada a um fluxo, toda a implementação da biblioteca `http2` segue o modelo orientado a eventos, não só para as características internas de baixo nível do protocolo HTTP/2 mas também para a exposição e controle da API disponibilizada junto com a biblioteca (ver Seção 3.4). Apesar disso, um programador que utiliza a biblioteca `http2` não precisa saber sobre o ciclo de vida de fluxos porque o trabalho de controlar esse ciclo é realizado internamente pela biblioteca.

Nesse contexto, as características de *multithreading* cooperativa de Lua descritas na Seção 2.2.2 simulam bem o modelo de orientação a eventos. As *callbacks* podem ser implementadas como *threads* cooperativas e têm como opção devolver o fluxo de execução ao *loop* sem necessariamente terminar o tratamento completo do evento. Durante o processamento das *callbacks*, elas conseguem transferir o controle de volta ao *loop* e esperar a ocorrência de um evento requerido para que a execução venha a ser retomada.

Esse modelo de eventos é comumente escolhido no cenário de programação de aplicações de rede. Em Lua, poderíamos utilizar co-rotinas diretamente para implementar esse modelo, em que teríamos que chamar funções da biblioteca padrão *coroutine* como forma alternativa de implementar o modelo de eventos. Porém, durante a implementação, focamos em utilizar todas as facilidades de escalonamento de co-rotinas da biblioteca Copas [5] de forma a gerenciar co-rotinas mais facilmente.

A biblioteca Copas descrita na Seção 2.2.3 pode cumprir o papel de programar eventos porque ela possui funções que permitem escalonar *threads*. O diagrama de sequência UML na Figura 3.1 mostra como Copas foi aproveitada como a entidade responsável por escalonar *threads* e foi feita com base no comportamento de tratamento de eventos com *multithreading* cooperativa mostrado por Silvestre [24].

Uma nova *thread* cooperativa é criada para tratar o evento A e transfere o controle imediatamente para a mesma. Após algum processamento, a *thread* termina o processamento do evento A, finaliza sua execução e o controle é devolvido ao `copas.loop()`, que por sua vez recebe um novo evento B. Mas o evento B não é processado por completo, devolvendo o controle para o `copas.loop()` imediatamente após a chamada de `copas.sleep(-1)`. Novamente com o controle, o `copas.loop()` recebe um terceiro evento C, que continua a processar o evento B através da *thread* B. Observe que a execução do `copas.loop()` não é necessariamente interrompida após a chamada de `copas.wakeup()`. Dependendo das próximas instruções a serem executadas pelo `copas.loop()`, ele continuará executando até que alguma chamada de função bloqueante (como `copas.sleep()` ou uma operação de rede) seja executada. No entanto, o evento B é concluído, a *thread* B termina sua execução e retorna a execução ao `copas.loop()`, que logo finaliza a execução.

Notamos que poderíamos usar co-rotinas simples de Lua para implementar esse modelo de programação a eventos, fazendo com que a biblioteca usasse co-rotinas para

empregar um mecanismo de notificação para obtenção do resultado do processamento de um evento. Esse mecanismo é implementado através de funções *callback*, que são passadas como parâmetro no ato da chamada das funções da API (ver Seção 3.4). Uma função da API invoca a *callback* após o processamento de um evento estar completo (operação concluída).

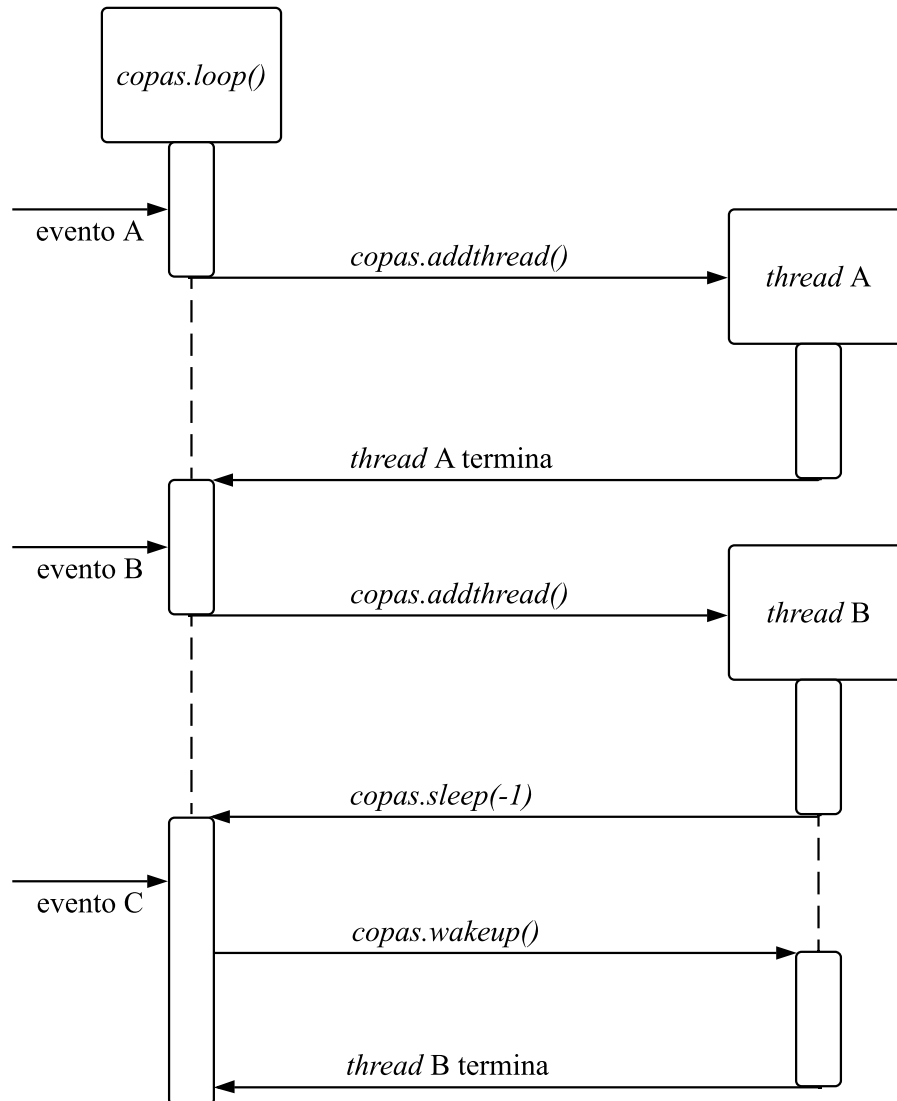


Figura 3.1: Diagrama de sequência UML que modela a programação orientada a eventos utilizando a biblioteca Copas.

3.2 Modelo de Concorrência

O modelo de programação concorrente também foi necessário para atender aos requisitos de um cliente HTTP/2. Conforme visto na Seção 2.1.3, um servidor pode multiplexar fluxos em uma única conexão HTTP/2. Isso significa que múltiplos fluxos podem estar sendo enviados pelo servidor de modo concorrente em qualquer momento

durante uma conexão HTTP/2. Precisamos então de uma espécie de escalonador de fluxos para que possamos lidar com múltiplos fluxos sendo enviados concorrentemente por um servidor HTTP/2.

Para recebermos e processarmos fluxos concorrentes, também precisamos de operações assíncronas ao nível de *sockets* TCP porque o modelo clássico de bloquear operações de rede não resolve o problema de tratar fluxos concorrentes na conexão TCP (que na verdade são apenas uma cadeia de bytes dentro do *buffer* TCP), além de ser uma abordagem ineficiente porque o cliente passaria a maior parte do tempo bloqueado na chamada de rede *receive* do *socket*.

Visando obter tanto concorrência para processar fluxos quanto operações de rede assíncronas, também utilizamos a biblioteca Copas. Nesse caso, o objetivo de Copas é escalonar múltiplos fluxos que podem estar presentes dentro da conexão TCP. As funcionalidades de Copas de prover operações assíncronas, descritas na Seção 2.2.3, atendem bem o papel de escalonar os fluxos no nível de *sockets* TCP por combinar as características de *multithreading* cooperativa de Lua com as operações assíncronas de rede da biblioteca LuaSocket [19].

Sempre que o *dispatcher* de Copas verifica que uma *thread* ainda está realizando alguma operação de rede *receive*, mas ainda não a concluiu, o *status* identificado por “*timeout*” de LuaSocket é sinalizado, o que significa que a operação retornou por incompleto. Nesse caso, a *thread* é suspensa por Copas. Internamente, o *dispatcher* de Copas implementa um *loop* despachante que emprega um mecanismo de notificação para obter o resultado do processamento de alguma operação de rede assíncrona contida em *threads* distintas, chamando uma por uma.

Assim, antes de detalharmos a implementação de ambos os modelos vistos e como eles foram empregados para implementar o lado cliente do HTTP/2, a Figura 3.2 formaliza as dependências da biblioteca http2 através de um diagrama de componentes UML, em que cada biblioteca é representada por um componente. A biblioteca http2 depende da biblioteca LuaSocket para obter suporte tanto ao protocolo TCP quanto a funcionalidades de construção e processamento de URLs; de Copas para construirmos um escalonador de fluxos e obtermos operações de rede assíncronas e de LuaSec [25] para fazer uso de conexões TLS seguras através do protocolo ALPN [10]. Na próxima seção mostramos outro diagrama de componentes UML que ilustra a estrutura interna da biblioteca http2.

3.3 Modelagem do Código-fonte

O diagrama de componentes UML retratado na Figura 3.2 mostra como o código-fonte da biblioteca http2 foi modelado. O programador apenas interage com o

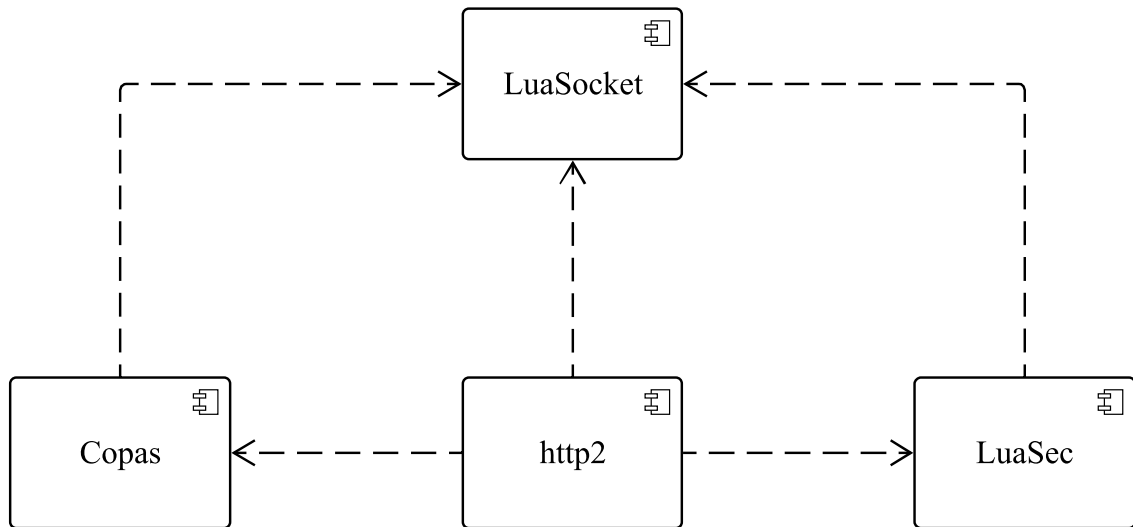


Figura 3.2: Dependências da biblioteca `http2` por outras bibliotecas de Lua.

módulo `http2` diretamente, sendo que os demais servem para oferecer interfaces projetadas especificamente em torno do suporte para as características de baixo nível do protocolo HTTP/2, além de permitir uma melhor organização de código através da modularização de funcionalidades.

- *connection*: encapsula o gerenciamento de uma conexão HTTP/2.
- *stream*: encapsula o gerenciamento de fluxos HTTP/2.
- *framer*: realiza codificação e decodificação de quadros HTTP/2.
- *hpack*: realiza a codificação e decodificação de campos de cabeçalhos.
- *http2*: oferece a interface necessária para a criação de um cliente HTTP/2.

3.3.1 Módulo *connection*

Alguns quadros HTTP/2 como `WINDOW_UPDATE`, `SETTINGS` e `GOAWAY` são enviados a nível de uma conexão HTTP/2 e são pertinentes à toda sessão de comunicação entre clientes e servidores. No HTTP/2, é adotado a convenção de que esses quadros são enviados em um fluxo com identificador 0, indicando que esse fluxo é reservado para controlar a conexão e não pode ser utilizado para criar outros fluxos. Para tanto, o módulo *connection* foi criado para abstrair a responsabilidade de controlar uma conexão do módulo de fluxos *stream* (ver Seção 3.3.2).

Essencialmente, o módulo *connection* oferece toda a lógica necessária para um cliente HTTP/2 se comportar com as exigências especificadas de um cliente HTTP/2 em

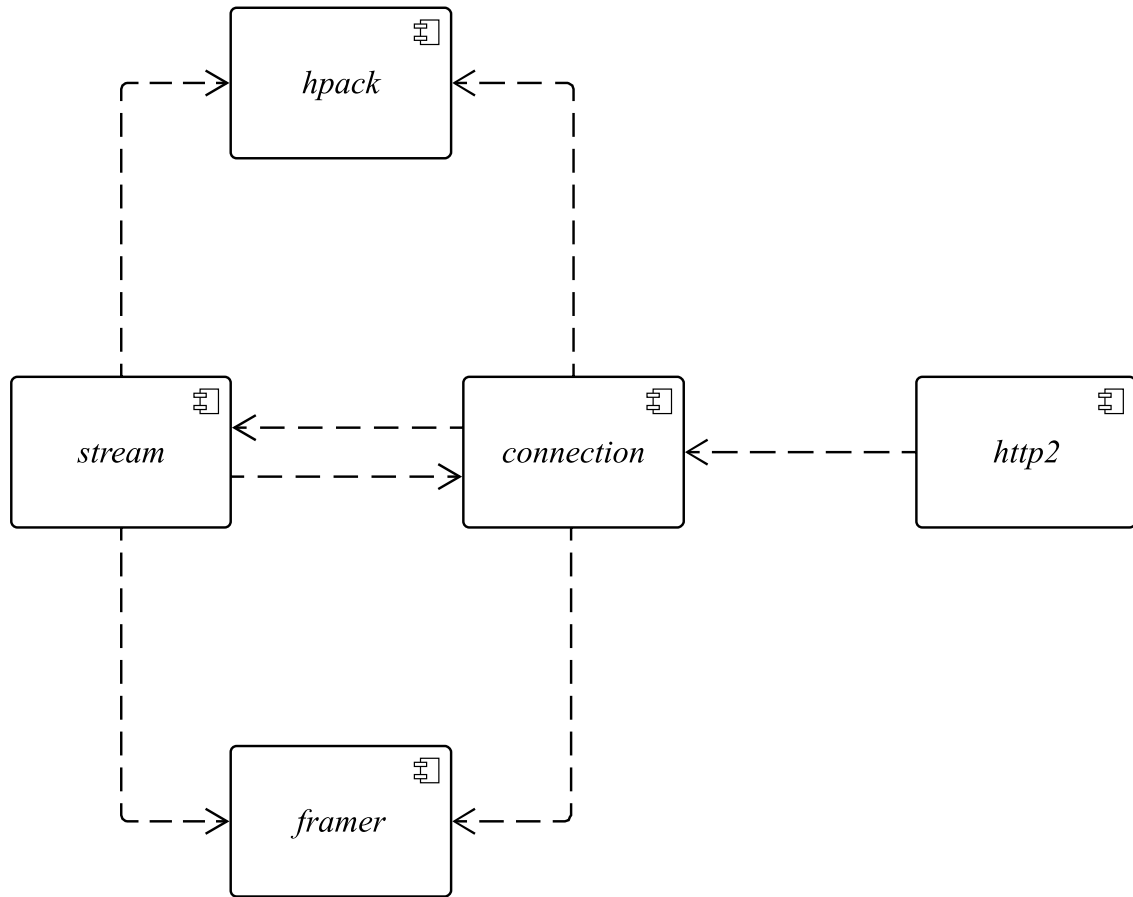


Figura 3.3: Componentes da biblioteca *http2*.

termos do uso que ele faz da conexão HTTP/2 ¹. As funções disponibilizadas por esse módulo são:

- `connection:window_update(payload)`: envia um quadro *WINDOW_UPDATE* com o seu respectivo corpo (`payload`) para o servidor.
- `connection:settings(payload)`: envia um quadro *SETTINGS* com o seu respectivo corpo (`payload`) para o servidor.
- `connection:goaway(payload)`: envia um quadro *GOAWAY* com o seu respectivo corpo (`payload`) para o servidor.
- `connection:parse_stream()`: encapsula toda a funcionalidade requerida para decodificar e processar um fluxo HTTP/2 contido no *buffer* do TCP.
- `connection:new_stream(identifier)`: aloca e retorna um novo fluxo (`stream`) para a conexão atual atribuindo-lhe o identificador `identifier`.
- `connection:new()`: construtor para uma nova conexão HTTP/2.

¹Na verdade, esse comportamento é simétrico entre clientes e servidores, ou seja, as mesmas funcionalidades do módulo *connection* podem ser aplicadas tanto para clientes quanto para servidores HTTP/2.

3.3.2 Módulo *stream*

Uma única conexão HTTP/2 pode multiplexar múltiplos fluxos concorrentes: múltiplas requisições e respostas podem ser enviadas simultaneamente e os dados (quadros) do fluxo podem ser intercalados e priorizados. O módulo *stream* encapsula todo o gerenciamento de estados, transições, controle de fluxo e gerenciamento de erros no nível de fluxos conforme definidos pela especificação do HTTP/2. As funções disponibilizadas por esse módulo são:

- `stream:parse(frame)`: processa um quadro (*frame*) proveniente de um fluxo HTTP/2.
- `stream:headers(payload)`: envia um quadro *HEADERS* com o seu respectivo corpo (*payload*) para o servidor.
- `stream>window_update(payload)`: envia um quadro *WINDOW_UPDATE* com o seu respectivo corpo (*payload*) para o servidor.
- `stream:new()`: construtor para um novo fluxo HTTP/2.

3.3.3 Módulo *framer*

Como visto na Seção 2.1.2, uma das melhorias trazidas pelo HTTP/2 em relação ao HTTP/1.1 é a forma como as mensagens HTTP são codificadas e decodificadas para serem enviadas na conexão TCP. No HTTP/2, cada mensagem HTTP/1 é mapeada para um quadro, a unidade de comunicação básica do HTTP/2, com o intuito de introduzir um novo enquadramento binário de mensagens. O módulo *framer* (“enquadrador”) implementa esse novo enquadramento, cuja função é descrever a maneira que quadros HTTP/2 são estruturados para serem formados em fluxos multiplexados. As funções auxiliares providas por esse módulo são:

- `encode(frame)`: gera um quadro HTTP/2 codificado em binário a partir de suas características desejáveis (cabeçalho e corpo) passadas como parâmetro (*frame*).
- `decode(buffer)`: decodifica um quadro HTTP/2 completo a partir de um *buffer* TCP passado como parâmetro e retorna os campos referentes ao cabeçalho e ao corpo desse quadro.

3.3.4 Módulo *hpack*

A implementação do formato de compressão de cabeçalhos para o HTTP/2 (HPACK) [23] para representar os cabeçalhos HTTP mais eficientemente é fornecido pelo módulo *hpack*. Apenas as funções utilitárias de codificação e decodificação de cabeçalhos foram implementadas nesse módulo, que são:

- `encode(header_list)`: responsável por codificar listas de pares chave-valor (`header_list`) utilizando o algoritmo HPACK.
- `decode(header_block)`: processa um bloco de cabeçalhos (`header_block`).
- `new(HEADER_TABLE_SIZE)`: cria e retorna um novo contexto HPACK a partir do parâmetro do quadro *SETTINGS*, `HEADER_TABLE_SIZE`.

Código 3.1 Cliente HTTP/2

```
1  local http2 = require "http2"
2
3  local url = "https://www.example.org/"
4
5  http2.on_connect(url, function(session)
6    local req = session.request()
7
8    req.on_response(function(headers)
9      for _, field in ipairs(headers) do
10        for name, value in pairs(field) do
11          print(name, value)
12        end
13      end
14    end)
15
16    req.on_data(function(data)
17      print(data)
18    end)
19  end)
```

3.3.5 Módulo *http2*

Talvez o módulo mais importante da implementação seja o *http2*, que merece atenção especial nesta seção. Trata-se de um módulo responsável por prover ao programador operações específicas de um cliente HTTP/2, bem como controlar o fluxo de execução do *loop* de eventos do Copas. Como o modelo de programação da biblioteca *http2* é orientado a eventos, o código do programador é encarregado de tratar um evento emitido durante a sessão de comunicação com um servidor HTTP/2.

Para melhor expor o funcionamento desse módulo, considere um exemplo de um código cliente HTTP/2 que faz uso da biblioteca *http2*, ilustrado no Código 3.1. Nesse código, o programador registrou algumas funções anônimas atuando como funções de *callback* para tratarem de determinados eventos.

- Na linha 5, uma função é registrada como tratadora do evento *on_connect*, que é emitido quando uma conexão HTTP/2 com o servidor é estabelecida. Essa função de *callback* é invocada com o argumento *session*, que representa uma sessão de comunicação HTTP/2 com o servidor.
- Na linha 6, uma requisição é submetida ao servidor conectado através da função *request*. Essa requisição é realizada enviando cabeçalhos padrões definidos pela biblioteca, pois nenhum argumento foi passado para a função *request*.
- Na linha 8, uma função é registrada como tratadora do evento *on_response*, que é emitido quando um quadro *HEADERS* é recebido do servidor HTTP/2 conectado do fluxo correspondente à requisição *req*. Essa função de *callback* é invocada com o argumento *headers*, que contém o objeto que representa cabeçalhos HTTP/2 (uma tabela bidimensional de listas de cabeçalhos).
- Na linha 16, uma função é registrada como tratadora do evento *on_data*, que é emitido quando o servidor HTTP/2 conectado termina de enviar dados (através de quadros *DATA*). Essa função de *callback* é invocada com o argumento *data*, que é uma variável do tipo *string* contendo os dados transmitidos pelo servidor.

O Código 3.1 pode ser facilmente estendido para realizar mais de uma requisição, bastando apenas repetir as chamadas de funções e trocando os nomes das variáveis associadas a requisição. Nesse contexto, o papel do módulo *http2* é abstrair o enquadramento de mensagens e multiplexação de fluxos de baixo nível realizados pela biblioteca, permitindo que o programador se concentre apenas na lógica de sua aplicação ou de seu cliente específico desde que as funções da API sejam corretamente chamadas.

As funções de alto nível fornecidas pelo módulo *http2* representam eventos que são emitidos quando computações específicas sobre dados transmitidos pelo servidor são enviados. Esses dados são passados como argumentos para as funções de *callback* registradas pelo programador para que ele possa tratá-los apropriadamente. Efetivamente, essas funções compõem a API da biblioteca *http2* e são explicadas na Seção 3.4. Uma função de baixo nível que faz parte da estrutura interna do módulo *http2* é chamada de *dispatcher* e é explicada em seguida.

Dispatcher

O *dispatcher* é uma função interna do módulo *http2* e é uma parte importante na biblioteca *http2*, pois ele é o responsável por realizar a tarefa de intercalar os quadros contidos em diferentes fluxos concorrentemente abertos em uma conexão TCP, bem como despachar *threads* associadas a esses fluxos. Ele foi construído baseando-se no modelo de programação concorrente com as características descritas nas Seções 2.2.2 e 2.2.3 e

também no modelo de programação orientado a eventos com as características descritas no início deste capítulo.

Essencialmente, o *dispatcher* é um simples *loop* que recebe um quadro de um *buffer* TCP, determina a que fluxo esse quadro pertence e processa o seu conteúdo para que ele possa ser adequadamente intercalado. Como os fluxos HTTP/2 são concorrentes entre si, fica a cargo do *dispatcher* administrar esse nível de concorrência de modo a evitar problemas de coordenação e sincronização. Intercalado um quadro, o *dispatcher* verifica se alguma *thread* pode ser acordada em resposta ao término do processamento de determinados quadros de um fluxo.

É mais fácil entender o *dispatcher* visualizando seu código. O Código 3.2 mostra como a implementação do *dispatcher* foi feita em função do gerenciamento de *threads* do Copas e do ciclo de vida um fluxo. Com o auxílio da função `connection:parse_stream` (ver Seção 3.3, intercalamos quadros HTTP/2 contidos em um fluxo específico. Através do *dispatcher*, o ciclo de vida de um fluxo (mostrado na Figura 2.3) é efetivamente implementado. Observamos que esse ciclo de vida foi implementado utilizando o modelo de programação orientada a eventos: quando um determinado estado de um fluxo é alcançado, despachamos a *thread* (nesse caso, a *callback*) que trata desse estado com a função `copas.wakeup`. O *loop* termina quando o estado *closed* é alcançado.

3.4 API

A API da biblioteca `http2` é definida em termos do modelo de programação orientada a eventos com as características descritas no início deste capítulo. O Código 3.1 serve como um exemplo da maneira que o programador pode utilizar a biblioteca chamando funções da API disponibilizada. Todas as funções da API são definidas da seguinte forma:

- `http2.on_connect(url, callback)`: dispara a função de resposta *callback* quando uma conexão HTTP/2 é estabelecida com o servidor identificado por *url*. Retorna uma tabela representando uma sessão de comunicação HTTP/2.
- `session.request([headers, body])`: submete uma requisição para o servidor HTTP/2 conectado na sessão *session* e retorna uma tabela utilizada para invocar funções de *callback* associadas a essa requisição (vê-las a seguir). Caso estejam presentes, *headers* é uma tabela bidimensional contendo listas de cabeçalhos HTTP/2 e *body* é o corpo de uma requisição POST. Caso contrário, uma requisição GET é submetida com campos de cabeçalhos extraídos da URL passada como argumento da função `on_connect`.
- `req.on_response(callback)`: dispara a função de resposta *callback* quando um quadro *HEADERS* é recebido do servidor HTTP/2 conectado do fluxo corres-

pendente à requisição `req`. A função `callback` é invocada com uma tabela bidimensional contendo listas de cabeçalhos HTTP/2.

- `req.on_data(callback)`: dispara a função de resposta `callback` quando o servidor HTTP/2 conectado termina de enviar dados (através de quadros *DATA*). A função `callback` é invocada com uma string contendo os valores dos dados recebidos.

Código 3.2 Dispatcher

```
1 local function dispatcher()
2     local stream
3     while true do
4         stream = conn:parse_stream()
5         if stream.state == "open" then
6             local on_response = callbacks.on_response[stream.id]
7             if on_response then
8                 copas.wakeup(on_response)
9             end
10        end
11        if stream.state == "closed" then
12            local on_data = callbacks.on_data[stream.id]
13            if on_data then
14                copas.wakeup(on_data)
15            end
16            break
17        end
18    end
19 end
```

3.5 Considerações Finais

O presente capítulo descreveu a implementação da biblioteca `http2`. O código-fonte, juntamente com o cliente mostrado na Seção 3.3 podem ser encontrados em [6]. Requisições simples com os métodos de requisição *GET* e *POST* estão funcionando e foram testadas (outros métodos como *HEAD* e *PUT* podem ser simulados a partir desses dois), embora somente o HTTP/2 executado sobre uma conexão TLS seja suportado por padrão. Essa decisão foi tomada em reflexo da tendência atual de habilitar TLS por padrão tanto em navegadores quanto servidores e devido à pouca utilização do protocolo HTTP/2 sobre o TCP puro.

Como a implementação foi inteiramente escrita na linguagem Lua, alcançamos a capacidade de multiplataforma da biblioteca http2 porque Lua é uma linguagem multiplataforma. Com isso, a biblioteca segue a portabilidade de Lua. Uma restrição importante da implementação é que a biblioteca http2 não é compatível com versões anteriores à versão 5.3 de Lua. Essa decisão foi tomada sobretudo com base nas facilidades de manipulação de bits dessa versão e também visando a simplicidade da implementação.

Visto que o HTTP/2 é um protocolo extenso e complexo em funcionalidades, a biblioteca http2 oferece um suporte mínimo necessário para implementar as características básicas de um cliente HTTP/2. O programador não precisa ter total conhecimento do ciclo de vida de um fluxo porque atualmente apenas requisições simples podem ser feitas, ou seja, os estados e as transições que ocorrem na visão do programador são controlados internamente pela biblioteca. Na visão da biblioteca, contudo, implementamos os estados ocioso, aberto, semifechado (remoto) e fechado.

Como resultado, as principais características de um cliente HTTP/2 foram implementadas, como a habilidade de realizar multiplexação de fluxos e de fazer compressão e descompressão de campos de cabeçalhos. Em termos das outras novas características do HTTP/2, o *push* de requisições/respostas do servidor e priorizações de requisições não foram implementados, uma vez que o HTTP/2 não obriga o suporte para ambos esses mecanismos e também devido à limitação de tempo disponível para implementar essas novidades. Consequentemente, quadros *PRIORITY* e *PUSH_PROMISE* não foram implementados. O quadro *PING* também não foi implementado porque ainda não mantemos verificações periódicas sobre o estado da conexão. Os outros quadros foram implementados e são suportados por padrão.

Resultados Experimentais

Neste capítulo vamos apresentar alguns testes feitos para verificar o desempenho de um cliente escrito utilizando a biblioteca `http2` quando submetido a diferentes fluxos de dados. Para efeito de análise, vamos utilizá-la para criar um cliente HTTP/2, chamado `http2`, a fim de compará-lo com outros clientes escritos em suas respectivas bibliotecas que também implementam o lado cliente do protocolo HTTP/2.

Os testes foram conduzidos da seguinte forma: selecionamos três clientes HTTP/2 populares para realizarem 10 requisições para um mesmo servidor HTTP/2 e para processarem as respostas enviadas por esse servidor simultaneamente na mesma conexão TCP. Essas 10 requisições serão feitas 10 vezes seguidas através de 10 diferentes execuções do cliente em particular. O servidor escolhido foi o servidor `nghttpd` da biblioteca `nghttp2` (versão 1.30.0) [20]. Os clientes escolhidos foram:

- o cliente da biblioteca `nghttp2` (versão 1.30.0): `nghttp` [20].
- um cliente que utiliza a biblioteca “lua-http” de Lua (versão 0.2) [17]: `lua-http`.
- um cliente que utiliza a biblioteca “http2” de Node.js (versão 8.10.0) [21]: `nodejs`.

O computador que hospedou o cliente HTTP/2 tem como especificações técnicas um processador Intel Core i3-4340 com 4 núcleos, cada um com 3.6 GHz; 6 GB de memória RAM; Ethernet 100 Mb/s e executa o sistema operacional GNU/Linux de 64 bits, *kernel* 4.4.0-17134-Microsoft. O computador que hospedou o servidor HTTP/2 tem como especificações técnicas um processador AMD Dual Core E1-500 com 2 núcleos, cada um com 1.48 GHz; 4 GB de memória RAM; Ethernet 100 Mb/s e executa o sistema operacional GNU/Linux de 64 bits, *kernel* 4.15.0-38-generic.

Tentamos diversificar a escolha dos clientes de modo a testarmos como diferentes métodos para implementá-los podem impactar em seus desempenhos nos cenários de teste deste trabalho. O cliente `nghttp` foi escolhido por ter sido implementado na linguagem compilada C, na qual a expectativa é que ele obtenha um desempenho superior em relação aos demais, que são implementados em linguagens de *scripting*. O cliente `lua-http` foi selecionado por se assemelhar com o cliente `http2` em termos da linguagem Lua, apesar

de ambos empregarem diferentes mecanismos de implementação. O cliente *nodejs* foi escolhido por também ter sido implementado em uma linguagem de *scripting*.

Durante a execução dos testes, conectamos o computador cliente e o computador servidor na mesma rede LAN, bem como suspendemos quaisquer intervenções externas da rede que possam impactar negativamente a vazão de dados nos enlaces Ethernet. Apesar de estudos como [7, 22] mostrarem que o HTTP/2 apenas traz vantagens em certas condições de rede, é importante conduzir testes para comparar o desempenho da implementação do cliente *http2* com outros clientes HTTP/2 quando múltiplas requisições simultâneas são feitas na mesma conexão TCP, ao mesmo tempo evitando uma fuga ao escopo deste trabalho.

Nas Figuras 4.1, 4.2 e 4.3, medimos o tempo médio de execução de três testes feitos como descritos a seguir. Em cada teste, realizamos 10 execuções de cada cliente, cada execução submetendo um total de 10 requisições simultâneas na mesma conexão TCP (e não sequenciais ou em diferentes conexões TCP para cada requisição, como no HTTP/1.1) para o servidor.

O primeiro retrata melhor o cenário atual de uma requisição por uma página Web de tamanho típico de 1,5 megabytes [11]. Pelos resultados mostrados na Figura 4.1, observamos que o cliente da biblioteca *http2* se sobressaiu no cenário de uma curta troca de dados e, nesse caso, com tempo de 1,08 segundos para baixar 1 megabyte. Em comparação, para baixar 1 megabyte, o cliente *lua-http* precisou de 1,13 segundos; o cliente *nodejs*, 1,38 segundos e o *nghttp*, 1,1 segundos.

O segundo teste fez com que os clientes realizassem requisições para um arquivo de 10 MB. Pelos registros resumidos na Figura 4.2, o cliente *http2* em questão ainda mostra uma boa vazão ao levar 1,11 segundos para baixar 1 megabyte nessa situação de um médio fluxo de dados. Em comparação, para baixar 1 megabyte, o cliente *lua-http* precisou de 1,13 segundos; o cliente *nodejs*, 1,17 segundos e o cliente *nghttp*, 1,10 segundos.

No último teste, os clientes submeteram requisições para um arquivo de 100 MB. Depois, medimos o tempo médio de execução (em segundos) e obtivemos o satisfatório resultado de que o cliente *http2* teve um tempo comparável com cliente *nghttp*. Em média, para baixar 1 megabyte nesse cenário, o cliente *http2* levou 0,89 segundos. Em comparação, para baixar 1 megabyte, o cliente *lua-http* precisou de 0,94 segundos; o cliente *nodejs*, 0,89 segundos e o cliente *nghttp*, também 0,89 segundos.

É importante observar que o cliente *http2* foi implementado visando funcionalidades mínimas necessárias para um cliente HTTP/2, pois ela somente cumpre com os requisitos mínimos do protocolo HTTP/2. Em contrapartida, as outras implementações, por serem mais maduras, podem ter uma lógica de processamento mais bem elaborada internamente, podendo impactar mais na vazão dos dados transmitidos pelo servidor.

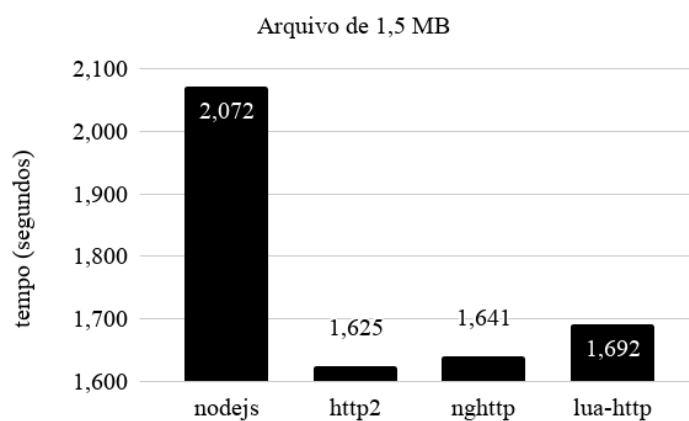


Figura 4.1: Tempo médio de execução de quatro clientes HTTP/2, onde cada um fez 10 requisições de um arquivo de 1,5 megabytes.

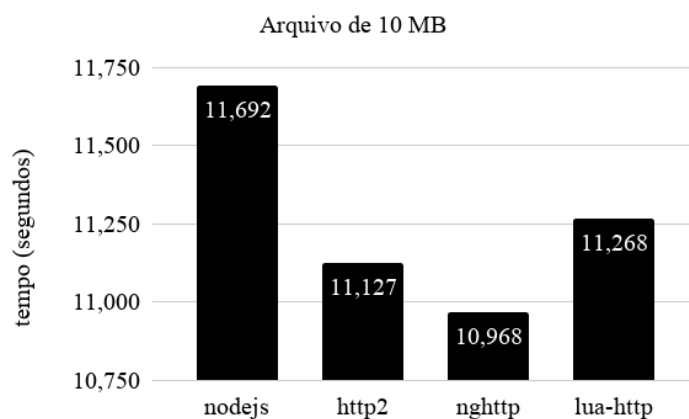


Figura 4.2: Tempo médio de execução de quatro clientes HTTP/2, onde cada um fez 10 requisições de um arquivo de 10 megabytes.

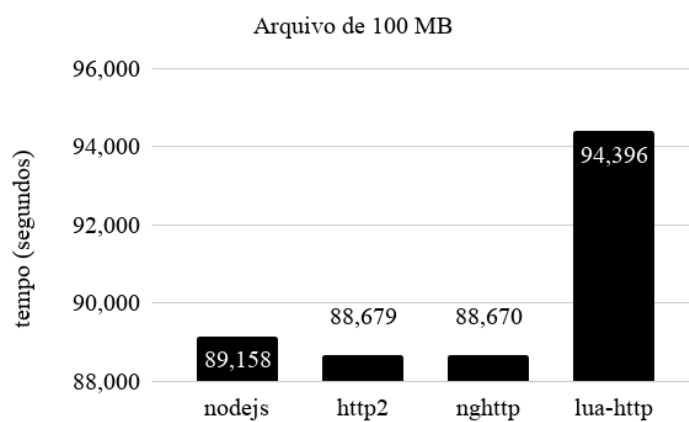


Figura 4.3: Tempo médio de execução de quatro clientes HTTP/2, onde cada um fez 10 requisições de um arquivo de 100 megabytes.

Considerações Finais

Este trabalho apresentou as características teóricas e práticas da implementação de uma biblioteca cliente HTTP/2 multiplataforma em Lua, denominada `http2`. A metodologia empregada para desenvolver a biblioteca `http2` com a característica de ser multiplataforma tem por base o modelo de programação orientado a eventos o modelo de programação concorrente de Lua. O modelo orientado a eventos foi adotado para implementar os estados de um fluxo HTTP/2, bem como para fornecer uma API baseada em *callbacks*. O modelo concorrente permitiu que fluxos HTTP/2 multiplexados pudessem ser recebidos e processados de forma concorrente e assíncrona.

A programação de clientes HTTP/2 é permitida através da presente biblioteca, que é multiplataforma por ser escrita inteiramente em Lua, oferecendo aos programadores uma simples API para realizarem requisições HTTP/2 simultâneas na mesma conexão TCP, aproveitando os benefícios de desempenho trazidos pelo protocolo HTTP/2. A construção de clientes HTTP/2 e de aplicações Web que executam sobre o HTTP/2 é amplamente encorajada, principalmente devido as vantagens de desempenho em relação ao HTTP/1.1.

No entanto, o HTTP/2 é um protocolo complexo. É difícil implementar o lado cliente desse protocolo com todas as suas características em escala de produção. Com ciência disso, decidimos implementar apenas as funcionalidades essenciais que permitem a criação de clientes HTTP/2 em conformidade com a especificação do HTTP/2, incluindo as duas principais características do HTTP/2: a capacidade de realizar múltiplas requisições simultâneas na mesma conexão TCP e de realizar a compressão de campos de cabeçalhos utilizando o HPACK.

Em termos do suporte às características do HTTP/2: permitimos multiplexação de fluxos na mesma conexão TCP; implementamos o algoritmo de compressão de campos de cabeçalhos do HTTP/2, o HPACK; implementamos os quadros *DATA*, *HEADERS*, *RST_STREAM*, *SETTINGS*, *GOAWAY*, *WINDOW_UPDATE* e *CONTINUATION*; implementamos os estados de um fluxo ocioso, aberto, semifechado (remoto) e fechado; requisições *GET* e *POST* triviais foram implementadas, possibilitando que outros métodos de requisições que são derivados desses dois (como *HEAD* e *PUT*) sejam utilizados.

5.1 Trabalhos Futuros

Uma primeira proposta para trabalhos futuros é a implementação das características de um cliente HTTP/2 que não foram implementadas neste projeto. Priorização de requisições podem trazer benefícios de desempenho para a aplicação quando vários fluxos são abertos. *Push* de requisição/resposta do servidor podem ser implementados e aceitos quando um servidor solicita esse mecanismo, podendo melhorar significativamente o desempenho da aplicação dependendo da política de *push* do servidor.

Como o protocolo HTTP/2 não alterou a semântica existente do HTTP/1.1, seria interessante dar suporte a mais características especificadas do HTTP/1.1, como *lookups* DNS e gerenciamento de *cookies*.

Uma outra forma de melhorar a implementação seria adicionando mais tratamentos de erros e, conseqüentemente, melhorando a conformidade de clientes escritos utilizando a biblioteca http2 com os tratamentos de erros do HTTP/2.

Por outro lado, mudanças feitas nesses sentidos precisam ser feitas de forma transparente com o código do programador, ajustando a API para que as mudanças internas sejam refletidas externamente na utilização da biblioteca.

Outra abordagem mais avançada seria reavaliar a forma de funcionamento da biblioteca Copas, levando em consideração que ela possui um único *loop* de estado global. Seria interessante alterar Copas para, por exemplo, não dependermos de um único *loop* global, de forma que possamos controlar mais facilmente a programação de clientes HTTP/2.

Referências Bibliográficas

- [1] BELSHE, M.; PEON, R. **A 2x Faster Web**. <https://blog.chromium.org/2009/11/2x-faster-web.html>, 2009. Online; acessado em 13/11/2018.
- [2] BELSHE, M.; PEON, R. **SPDY: An experimental protocol for a faster web**. <https://www.chromium.org/spdy/spdy-whitepaper>, 2009. Online; acessado em 30/10/2018.
- [3] BELSHE, M.; PEON, R.; THOMSON, M. **Hypertext Transfer Protocol Version 2 (HTTP/2)**. RFC 7540, RFC Editor, May 2015.
- [4] BRYLINSKI, A.-S.; BHATTACHARJYA, A. **Overview of HTTP/2**. In: *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing, ICC '17*, p. 114:1–114:6, New York, NY, USA, 2017. ACM.
- [5] COPAS. **Copas — Coroutine Oriented Portable Asynchronous Services for Lua**. <http://keplerproject.github.io/copas>. Online; acessado em 13/11/2018.
- [6] DE PAULA, M. R. **http2 — Multi-platform HTTP/2 client library in Lua**. <https://github.com/murillow/http2>. Online; acessado em 29/11/2018.
- [7] DE SAXCÉ, H.; OPRESCU, I.; CHEN, Y. **Is HTTP/2 really faster than HTTP/1.1?** In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, p. 293–299, April 2015.
- [8] FIELDING, R. T.; RESCHKE, J. F. **Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing**. RFC 7230, RFC Editor, June 2014.
- [9] FIELDING, R. T.; RESCHKE, J. F. **Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content**. RFC 7231, RFC Editor, June 2014.
- [10] FRIEDL, S.; POPOV, A.; LANGLEY, A.; STEPHAN, E. **Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension**. RFC 7301, RFC Editor, July 2014.

- [11] HTTP ARCHIVE. **Report: State of the Web.** <http://httparchive.org/reports/state-of-the-web?start=earliest&end=latest>, 2018. Online; acessado em 13/11/2018.
- [12] IERUSALIMSKY, R. **Programming with multiple paradigms in Lua.** In: Escobar, S., editor, *Proceedings of the 18th International Conference on Functional and (Constraint) Logic Programming*, volume 5579 de **LNCS**, p. 5–13, Heidelberg, Germany, 2009. Springer.
- [13] IERUSALIMSKY, R. **Programming in Lua, Fourth Edition.** Lua.Org, 2016.
- [14] IERUSALIMSKY, R.; DE FIGUEIREDO, L. H.; CELES, W. **Passing a language through the eye of a needle.** *Communications of the ACM*, 54(7):38–43, 2011.
- [15] IERUSALIMSKY, R.; DE FIGUEIREDO, L. H.; CELES, W. **A look at the design of lua.** *Communications of the ACM*, 61(11):114–123, Oct. 2018.
- [16] KUROSE, J. F.; ROSS, K. W. **Computer Networking: A Top-Down Approach.** Pearson, 6th edition, 2012.
- [17] LUA-HTTP. **lua-http — HTTP library for Lua.** <https://daurnimator.github.io/lua-http/0.2>, 2018. Online; acessado em 13/11/2018.
- [18] MANZOOR, J.; DRAGO, I.; SADRE, R. **The curious case of parallel connections in HTTP/2.** In: *2016 12th International Conference on Network and Service Management (CNSM)*, p. 174–180, Oct 2016.
- [19] NEHAB, D. **LuaSocket — Network support for the Lua language.** <http://w3.impa.br/~diego/software/luasocket>, 2007. Online; acessado em 13/11/2018.
- [20] NGHTTP2. **Nghttp2: HTTP/2 C Library.** <https://nghttp2.org/>, 2018. Online; acessado em 26/11/2018.
- [21] NODE.JS. **HTTP/2.** https://nodejs.org/api/http2.html#http2_http_2, 2018. Online; acessado em 26/11/2018.
- [22] ODA, N.; YAMAGUCHI, S. **HTTP/2 performance evaluation with latency and packet losses.** In: *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, p. 1–2, Jan 2018.
- [23] PEON, R.; RUELLAN, H. **HPACK: Header Compression for HTTP/2.** RFC 7541, RFC Editor, May 2015.

- [24] SILVESTRE, B. O. **Modelos de Concorrência e Coordenação para o Desenvolvimento de Aplicações Orientadas a Eventos em Lua**. PhD thesis, Pontífica Universidade Católica do Rio de Janeiro – PUC-Rio, 2009.
- [25] SILVESTRE, B. O. **LuaSec**. <https://github.com/brunoos/luasec/wiki>, 2018. Online; acessado em 27/11/2018.
- [26] WANG, X. S.; BALASUBRAMANIAN, A.; KRISHNAMURTHY, A.; WETHERALL, D. **How Speedy is SPDY?** In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, p. 387–399, Berkeley, CA, USA, 2014. USENIX Association.
- [27] YAMAMOTO, K.; TSUJIKAWA, T.; OKU, K. **Exploring HTTP/2 Header Compression**. In: *Proceedings of the 12th International Conference on Future Internet Technologies*, CFI'17, p. 1:1–1:5, New York, NY, USA, 2017. ACM.