Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Redes Definidas por Software Orientadas a Aplicação

Hemerson Aparecido da Costa Tacon

JUIZ DE FORA
NOVEMBRO, 2017

Redes Definidas por Software Orientadas a Aplicação

Hemerson Aparecido da Costa Tacon

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Alex Borges Vieira

JUIZ DE FORA

NOVEMBRO, 2017

Redes Definidas por Software Orientadas a Aplicação

Hemerson Aparecido da Costa Tacon

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Alex Borges Vieira
Doutorado em Ciência da Computação

Luciano Jerez Chaves
Doutorado em Ciência da Computação

Edelberto Franco Silva
Doutorado em Ciência da Computação

JUIZ DE FORA

21 DE NOVEMBRO, 2017

Resumo

Praticamente todos dispositivos tecnológicos atuais se relacionam com redes de alguma forma. Para conseguir um bom desempenho, o núcleo de redes é programado diretamente em *hardware*, dificultando assim a atualização da tecnologia utilizada. Além disso, a própria configuração e manutenção da mesma é dependente de administradores. Isso torna tais tarefas custosas e sujeitas a erros. Neste cenário surgiram as Redes Definidas por *Software* (RDS) tendo característica chave a separação do plano de dados do plano de controle. Dessa forma, RDS permitem que tarefas de configuração (controle) sejam realizadas independente do tráfego nas redes. Com isso, é possível programar as configurações como um *software*. Nesse contexto, o OpenFlow é a principal interface de programação de RDS. O protocolo OpenFlow desempenha um ótimo papel na comunicação entre plano de controle e plano de dados, porém, ainda não existe um padrão para a comunicação entre a aplicação e o controlador. Este trabalho tem por objetivo a elaboração de um arcabouço que demonstre a factibilidade de criação de uma interface de programação de aplicativos. Esse arcabouço permite que a rede se torne ciente da aplicação. Uma vez que esse arcabouço esteja criado, torna-se possível reencaminhar os pacotes da rede baseando-se no tipo de aplicação. A proposta foi avaliada em um cenário realista, onde uma aplicação de *streaming* utiliza o método criado.

Palavras-chave: RDS, API, redes de computadores, *streaming*.

Abstract

Practically all current technological devices relate to networks in some way. To achieve good performance, the network core is programmed directly in hardware, which turns it difficult to update the technology used. Moreover, the actual configuration and maintenance thereof is dependent on administrators. This makes such tasks costly and error prone. In this scenario, Software Defined Networking (SDN) arises, allowing easy configuration tasks with key characteristic separating the control plane from the data plane. In this way, SDN allow configuration tasks (control) are carried out independent of traffic over networks. With this, you can program the settings as a software. In this context OpenFlow is the main programming interface of SDN. The OpenFlow protocol plays a great role in the communication between control plane and data plane but there is still no standard for communication between the application and the controller. This work aims to develop a framework that demonstrates the feasibility of creating an application programming interface. This framework allows the network to become aware of the application. Once this framework is created, it becomes possible to forward the network packets based on application type. The proposal was evaluated in a realistic scenario, where a streaming application uses the method created.

Keywords: SDN, API, computer network, *streaming*.

Agradecimentos

A todos os meus parentes, pelo encorajamento e apoio.

Ao professor Alex pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"So long and thanks for all the fish".

Douglas Adams (The Hitchhiker's Guide to the Galaxy)

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

ABR	*Adaptative Bit Rate*
API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
CDN	*Content Delivery Network*
CLI	*Command Line Interface*
DCC	Departamento de Ciência da Computação
DPI	*Deep Packet Inspection*
Gbps	*Giga bits per second*
HD	*High Definition*
HTTP	*Hypertext Transfer Protocol*
IP	*Internet Protocol*
MAC	*Media Access Control*
NBI	*Northbound Interface*
ONF	*Open Network Foundation*
OSI	*Open Systems Interconnection*
QFF	*QoE Fairness Framework*
QoE	*Quality of Experience*
QoS	*Quality of Service*
RC	Redes de Computadores
RDS	Redes Definidas por *Software*
ROIA	*Real-Time Online Interactive Application*
SRT	*Statistics Retrieval Time*
SDN	*Software Defined Networking*
ssh	*Secure Shell*
TCP	*Transmission Control Procotol*
UFJF	Universidade Federal de Juiz de Fora
VoD	*Video on Demand*
VP	*Video Provider*

1 Introdução

O presente trabalho elabora um estudo para mostrar como redes definidas por *software* podem mudar o rumo das redes de computadores. O ponto atacado para alcançar tal finalidade baseia-se em deixar a rede ciente da aplicação. Isso é feito por meio da utilização de um controlador projetado que consegue realizar a finalidade supracitada. Inicialmente, será feita uma apresentação do tema, encaixando-o dentro da área de Redes de Computadores. Em sequência, o problema em questão é exposto. Uma justificativa dá continuidade ao capítulo, elucidando os benefícios que podem ser alcançados ao solucionar o problema mencionado. E, para finalizar o capítulo, os objetivos são identificados, sendo divididos em objetivo geral e objetivos específicos.

1.1 Apresentação do tema e contextualização do problema

Redes de computadores estão, há algum tempo, presentes em nosso cotidiano e conquistam cada vez mais espaço em diversas áreas. Essa constante expansão fez com que a área se tornasse de grande interesse, tanto para o mercado de trabalho, quanto para a área acadêmica (Guedes et al, 2012).

O paradigma de Redes Definidas por *Software* (RDS) emergiu com o objetivo de revolucionar o modo de implementação de redes de computadores, baseando-se em uma interface de programação simples. Neste âmbito é que foram criados a interface e protocolo OpenFlow (McKeown et al, 2008). Com OpenFlow, os elementos de encaminhamento oferecem uma interface de programação simples que lhes permite estender o acesso e controle da tabela de consulta utilizada pelo *hardware* para determinar o próximo passo de cada pacote recebido. Dessa forma, o encaminhamento continua sendo eficiente, pois a consulta à tabela de encaminhamento continua sendo tarefa do *hardware*, mas a decisão sobre como cada pacote deve ser processado pode ser transferida para um nível superior,

onde diferentes funcionalidades podem ser implementadas. Essa estrutura permite que a rede seja controlada de forma extensível através de aplicações expressas em *software* (Guedes et al, 2012).

O constante avanço da área de tecnologia fez com que redes de computadores fossem utilizadas cada vez mais. O foco atual se encontra nos dispositivos móveis e nas aplicações para os mesmos, que forçam cada vez mais as limitações encontradas atualmente nos serviços disponibilizados através de redes de computadores (ONF Solution Brief et al, 2013).

Os equipamentos utilizados no núcleo da infraestrutura das redes de computadores usam, em sua grande maioria, tecnologia proprietária e configurada diretamente no *hardware*. Essa dependência forte dos fabricantes fez com que os melhores equipamentos se tornassem caros. Outra desvantagem desses equipamentos é que não é possível modificá-los para que possam funcionar mais adequadamente para determinados propósitos como, por exemplo, alterar as políticas utilizadas para o encaminhamento de pacotes (Kreutz et al, 2015).

Vale ressaltar também que a maioria das redes de computadores utilizadas atualmente ainda utilizam a pilha de protocolos TCP/IP. Tais protocolos foram concebidos nas décadas de 1970 e 1980 nos primórdios da internet e não sofreram nenhuma evolução significativa desde então. Essas características fazem com que seja muito difícil ocorrer alguma atualização da tecnologia usada, uma vez que seria necessária uma transição entre os antigos equipamentos para os novos, o que também representa um grande custo para as empresas da área(Guedes et al, 2012).

Diante deste cenário surge então um problema, que pode ser formulado da seguinte maneira: "Como realizar a atualização da tecnologia usada no núcleo da rede, seja para prover uma melhoria no desempenho ou uma melhoria na qualidade de serviço, sem que isso gere altos custos?"

1.2 Justificativa

O paradigma de RDS surgiu com o intuito de resolver estes problemas, tendo como fundamento principal a separação do plano de dados do plano de controle, oferecendo assim

uma maior flexibilidade para programação de políticas e de elementos presentes na rede.

O paradigma RDS contempla três planos: aplicação, controle e dados (Rezende et al, 2016). A comunicação entre os planos de controle e dados é conhecida como API *Southbound* e possui uma implementação bem estabelecida, realizada pelo OpenFlow. O padrão OpenFlow como API *Southbound* foi estabelecido pela *Open Network Foundation*. A ONF é quem desenvolve e padroniza o protocolo OpenFlow. Já a comunicação entre os planos de controle e de aplicação é conhecida como API *Northbound*. Ao contrário da API *Southbound*, ainda não existe um padrão definido a se seguir quando o assunto é API *Northbound*. Nota-se uma resistência por parte dos desenvolvedores a criarem novas aplicações que tirem proveito das vantagens oferecidas por RDS pelo motivo de ausência de um padrão. A ONF ainda não se pronunciou sobre uma padronização para API *Northbound*. Diante deste cenário, se faz necessária a criação de uma API *Northbound* de qualidade e de fácil utilização para que tenha início a adoção de Redes Definidas por *Software* orientadas a aplicação. Uma vez que uma rede utilizando o paradigma RDS tenha conhecimento da aplicação que trafega por ela, são abertas novas possibilidades para a forma como os pacotes são encaminhados e também para a forma como as aplicações são desenvolvidas. Espera-se ainda que tal abordagem possa melhorar o desempenho das redes.

1.3 Objetivos

1.3.1 Objetivo Geral

O principal objetivo deste trabalho é criar um arcabouço que demonstre a factibilidade de criação de uma API *Northbound* a fim de se estabelecer uma comunicação efetiva entre as aplicações e o controlador, possibilitando assim uma interface de alto nível que facilite e incentive a criação de tais aplicações. Do ponto de vista de baixo nível, o controlador OpenFlow, através da API *Southbound*, já fornece nativamente algumas informações sobre o fluxo de dados: contadores de *bytes* e contadores de pacotes. Semelhante a esta API *Southbound*, por meio do arcabouço criado, é possível que o cliente forneça informações sobre seu tráfego, com a diferença de que as informações são obtidas de forma qualitativa.

Em contrapartida, o tráfego pode ser tratado de uma maneira mais inteligente.

1.3.2 Objetivos Específicos

Uma aplicação de *streaming* também foi elaborada para utilizar a arquitetura criada. O desempenho da arquitetura desenvolvida foi avaliado em conjunto com tal aplicação de *streaming*. Testes foram feitos em alguns cenários que propõe uma experiência similar à que um usuário se depara no dia-a-dia. Para avaliar o desempenho alcançado com o método proposto, os resultados foram analisados e comparados com o cenário onde nenhuma API desse tipo é utilizada, a fim de visualizar qual o impacto na qualidade de experiência (*Quality of Experience* - QoE) oferecida ao usuário.

1.4 Metodologia

A metodologia utilizada no trabalho constitui-se inicialmente de uma pesquisa na área, fazendo um levantamento dos mais recentes e principais trabalhos que contribuíram, de alguma forma, para crescimento e adoção de Redes Definidas por *Software*, com ênfase em deixar a rede ciente da aplicação. Um foco especial foi dado aos trabalhos que direcionam seus objetivos para a consolidação de uma API *Northbound* que execute, de maneira efetiva, a comunicação entre controlador e aplicação. Ao concluir essa investigação a respeito do tema, a revisão da literatura foi confeccionada utilizando-se dos trabalhos mais relevantes encontrados.

A partir desse ponto, o foco do trabalho voltou-se para a elaboração da comunicação entre controlador e aplicação, conhecida como API *Northbound*. Como ainda não existe um padrão estabelecido para API *Northbound*, o objetivo deste trabalho se alinhou na criação de algo diferente, tentando realizar uma contribuição relevante para o tema.

Antes da implementação do método, foi realizada a definição das mensagens a serem trocadas entre controlador e aplicação. Essas mensagens são recebidas na forma de pacotes. Para cada pacote recebido, uma ação foi associada. A ação normalmente é executada pelo controlador, que guia a conexão com o propósito de tornar a rede ciente da aplicação.

Tendo a proposta modelada, a implementação foi feita por meio do POX[1], que utiliza a linguagem de programação *Python*. Essa plataforma, juntamente com a linguagem, foram escolhidas por se integrarem facilmente ao ambiente virtual de testes utilizado, o *Mininet*[2]. Através do Mininet, é possível construir uma rede virtual realista, na qual um cenário de uso pode ser implantado para testar e validar o arcabouço criado. O cenário de uso basicamente consiste de usuários utilizando uma aplicação de *streaming* de vídeo. Em resumo, caberá ao controlador direcionar as solicitações de vídeo para os servidores da melhor forma possível. Tal aplicação de *streaming* também foi implementada, de forma a utilizar os recursos do método proposto. Para avaliar o desempenho, foram comparados os tempos de transmissão da aplicação de *streaming* utilizando o controlador projetado e da mesma aplicação de *streaming* sem fazer uso de tal controlador. A utilização ou não do controlador projetado implica em leves alterações no código e funcionamento da aplicação, porém as duas versões podem ser consideradas muito similares. As diferenças serão elucidadas e ficarão claras no capítulo 4.

O restante deste trabalho é dividido da seguinte forma: no capítulo 2 são apresentados, na forma de subtópicos, uma revisão sobre os temas e conceitos considerados pertinentes ao longo deste trabalho; a abordagem proposta é explicada com detalhes no capítulo 3; com o objetivo de demonstrar o desempenho alcançado pela solução proposta, no capítulo 4 são mostrados os resultados de alguns exprimentos; no capítulo 5 é desenvolvida uma pequena revisão da literatura, ressaltando as semelhanças e diferenças entre alguns trabalhos relevantes e este trabalho; por fim no capítulo 6 é apresentada a conclusão do trabalho bem como uma discussão sobre possíveis aprimoramentos para trabalhos futuros.

[1]Plataforma de código aberto (https://github.com/noxrepo/pox) para desenvolvimento de aplicações de controle para RDS

[2]http://mininet.org/

2 Fundamentação teórica

Neste capítulo, alguns conceitos essenciais para o desenvolvimento do trabalho são revisados. Os tópicos abordados variam desde os conceitos mais amplos como redes de computadores até os conceitos mais específicos relacionados ao tema aqui discutido, como o *Mininet*. O objetivo deste capítulo é ampliar a familiarização com os aspectos teóricos tratatos durante o decorrer do texto.

2.1 Redes de Computadores

Apesar da ampla utilização, as redes de computadores tradicionais baseadas no protocolo IP são complexas e difíceis de administrar (Feamster et al, 2014; Ramos et al, 2015; Kreutz et al, 2015). Alguns problemas surgem no momento de montar e configurar uma rede de computadores. Para explicitar as políticas de rede de alto nível desejadas, os operadores de rede têm que configurar cada equipamento de rede individualmente, utilizando comandos de baixo nível e muitas vezes específicos de cada fornecedor (Kreutz et al, 2015).

Estas redes possuem diversos equipamentos dos mais variados tipos e fabricantes: de roteadores a *switches* para *middleboxes*[3] tais como *firewalls*, tradutores de endereço de rede, balanceadores de carga de servidor e sistemas de detecção de intrusão. Dentre estes equipamentos, roteadores e *switches* utilizam *softwares* complexos e de controle distribuído, que normalmente são fechados e de licença proprietária. Durante o processo de criação de novos *softwares* que implementem os protocolos de rede é necessário passar por anos de padronização e testes de interoperabilidade para garantir o funcionamento com a vasta quantidade heterogênea de equipamentos no mercado (Feamster et al, 2014).

Nas redes de computadores existe uma sólida integração vertical no que diz respeito a sua infraestrutura, isto é caracterizado pelo forte acoplamento dos planos de controle e de dados nos equipamentos de rede (Ramos et al, 2015). Em uma era da computação em nuvem, este cenário faz com que a construção e manutenção de redes

[3]Filtros especiais de tráfego

eficientes se torne uma tarefa árdua, onde a quantidade de dados cresce cada vez mais, demandando um maior desempenho dos dispositivos (Ramos et al, 2015).

Para que o desempenho esperado seja atingido, uma das técnicas mais eficientes utilizadas é o encaminhamento dos dados de acordo com o seu tipo. Para isso é preciso realizar uma classificação destes dados. Uma vez identificado o tipo de determinado fluxo, ele poderá ser encaminhado utilizando-se da melhor política para tal tipo constatado. Nas redes tradicionais, a classificação da camada de aplicação é predominantemente realizada através de dispositivos específicos criados para suportar esta tarefa. Uma vez que diferentes fabricantes têm diferentes concepções, um mesmo fluxo pode ser classificado em diferentes tipos com diferentes dispositivos. Assim, é difícil determinar uma política coerente de gestão global da camada de aplicação para toda a rede. O conceito de separação do plano de controle e do plano de dados surgiu com a finalidade de solucionar os problemas citados e, juntamente com este conceito, emergiram as Redes Definidas por *Software* abrindo uma nova porta para o desenvolvimento das redes de computadores (Li et al, 2014).

2.2 Redes Definidas por *Software*

RDS é um paradigma de rede emergente que dá esperança para superar as limitações da infraestrutura das redes atuais (Kreutz et al, 2015). Esta nova tecnologia de rede divide o plano de controle do plano de dados dos dispositivos da rede e centraliza logicamente a gerência no controlador, além de oferecer interfaces abertas e programabilidade (Pham et al, 2016; Jarschel et al, 2014). Estas características trazem como benefícios principais as seguintes possibilidades: controle de redes é implementado em *software* através de programação, o que leva a equipamentos de rede mais simples uma vez que o *software* executado neles poderá ser programado, e também permite a virtualização de elementos de rede e automação da mesma. (Pham et al, 2016).

2.2.1 RDS Orientadas a Aplicação

A Internet, atualmente, não desfruta de uma transferência de informações direta entre aplicativos e a rede. Em consequência dos diversos requisitos das variadas aplicações que rodam em cima da rede, em certas situações, os usuários finais podem perceber uma má qualidade nas aplicações (Zinner et al, 2014). Para superar esta barreira, as aplicações devem tornar-se conscientes das redes e as redes devem se tornar sensíveis aos aplicativos (Zinner et al, 2014). Neste contexto, já existem algumas abordagens suportadas por RDS, como a moldagem de tráfego (Quinlan et al, 2015).

Outra possibilidade é a já citada classificação da camada de aplicação que, basicamente, pode ser feita de duas maneiras: inspeção profunda de pacotes (*Deep Packet Inspection* - DPI) e classificação de pacotes com base em aprendizado de máquina (*Machine Learning* - ML) (Li et al, 2014).

Em (Georgopoulos et al, 2014) é utilizada uma abordagem um pouco diferente baseada em *cache*: o OpenCache utiliza de RDS para proporcionar *cache* como um serviço para conteúdo de mídia de uma forma eficiente e transparente.

2.2.2 Interfaces de programação

O plano de controle de RDS realiza um controle direto sobre o estado dos equipamentos do plano de dados da rede (ou seja, roteadores, *switches* e outros *middleboxes*) através de uma interface de programação de aplicação (*Application Programming Interface* - API) bem definida. A API que realiza a comunicação entre o plano de controle e o plano de dados é chamada de API *Southbound*. O OpenFlow (McKeown et al, 2008) é um exemplo desta API e foi padronizado pela ONF, que é órgão dedicado a promoção e adoção de RDS (Feamster et al, 2014).

OpenFlow permite que experimentadores, pesquisadores, desenvolvedores de protocolo e administradores de rede explorem as verdadeiras capacidades de uma rede de um modo rápido, de fácil implementação e flexível. Além disso, eles têm a possibilidade de apresentar, em tempo real, uma nova funcionalidade sem ter que alterar especificamente qualquer um dos dispositivos de rede (Georgopoulos et al, 2013).

RDS também permite a troca de informações com as aplicações rodando em

cima da rede. Esta troca de informações é realizada através da API *Northbound* entre o controlador e a aplicação (Jarschel et al, 2014). A interface *Southbound* já tem uma proposta amplamente aceita (OpenFlow), mas uma interface *Northbound* comum que incentive o desenvolvimento de aplicações voltadas a RDS ainda é uma questão em aberto. Diante das inúmeras opções existentes, a ONF ainda não definiu uma API *Northbound* padrão (Ramos et al, 2015).

Uma visão simplificada da arquitetura RDS é mostrada na figura 2.1 a seguir encontrada em Kreutz et al (2015):

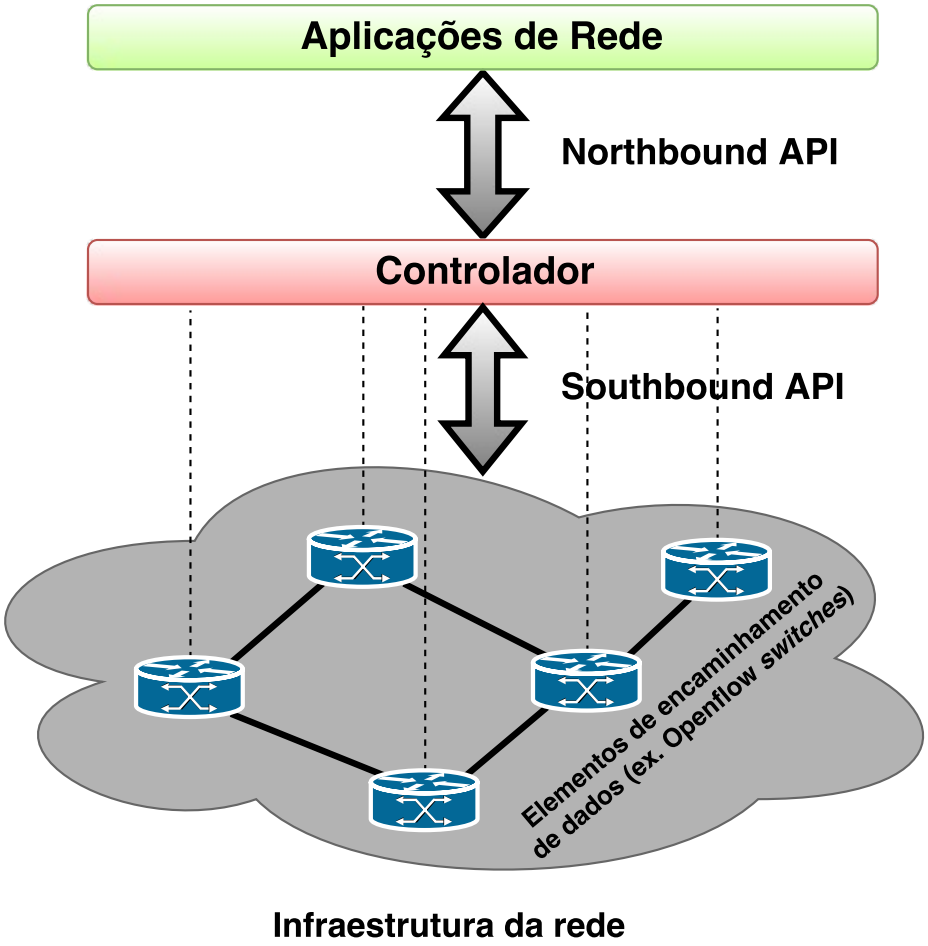

Figura 2.1: Visão simplificada da arquitetura RDS adaptada de Kreutz et al (2015)

2.3 API *Northbound*

Com a API *Northbound* do controlador RDS, o próprio aplicativo pode notificar a rede sobre suas propriedades e estado. Desta forma, o controlador de rede pode encaminhar os

fluxos de tráfego de forma a complementar ao invés de atrapalhar um ao outro (Jarschel et al, 2014), ou seja, o controlador torna-se programável e responde às mudanças na rede (Hue et al, 2015).

Via interfaces *Northbound* (*Northbound Interfaces* - NBI), aplicações podem fornecer informações para o plano de controle de rede, que pode então controlar o plano de dados. Particularmente, um plano de controle com reconhecimento de aplicativos pode ser empregado para aumentar o controle sobre a rede e melhorar qualidade de experiência (*Quality of Experience* - QoE) proporcionada ao usuário, alterando, por exemplo, o comportamento de encaminhamento de *switches*, ou através da concessão de mais recursos de rede para determinado fluxo (Zinner et al, 2014).

Controladores existentes, tais como Floodlight, Trema, NOX, Onix e OpenDaylight propõem e definem suas próprias APIs *Northbound*. Entretanto, cada uma das propostas tem as suas próprias definições específicas tornando difícil o desenvolvimento de aplicativos que funcionem adequadamente com todos equipamentos (Kreutz et al, 2015).

Na literatura atual existem algumas propostas de NBI para aplicações específicas. Em (Humernbrum et al, 2014) é elaborada uma NBI para as aplicações emergentes em tempo real na internet (*Real-Time Online Interactive Application* - ROIA). São definidas algumas funções que a API deve oferecer baseadas no envio de dados através da rede. A partir dessa comunicação entre aplicação e controlador é possível estabelecer um paralelo entre políticas de qualidade de serviço (*Quality of Service* - QoS) e QoE percebida pelo usuário.

Existe também uma proposta de NBI baseada em objetivos que explora as características modulares e reutilização dos micro serviços e arquiteturas orientadas a serviços. Mesmo diante das diversas opções apresentadas, as atuais NBIs estão com deficiência de algumas funcionalidades, o que inclui particionamento de serviços, verificadores de rede e contexto de serviço, funcionalidade de composição de serviços, entre outros (Pham et al, 2016).

2.4 Aplicações de *streaming*

Streaming de vídeo é uma forma cada vez mais difundida para consumir conteúdo multimídia (Georgopoulos et al, 2013). Com o aumento do crescimento de vídeo sob demanda (*Video on Demand* - VoD) e a popularidade de conteúdo em alta definição (*High Definition* - HD), um desafio preocupante para a infraestrutura de rede está se tornando evidente. A rede tem agora que transferir uma enorme quantidade de dados para o usuário final, e fazê-lo o mais rapidamente possível. (Georgopoulos et al, 2014).

Para aumentar a QoS oferecida, prestadores de serviços de vídeo usam Redes de Fornecimento de Conteúdo (*Content Delivery Networks* - CDNs) para acelerar a entrega de seu conteúdo aos clientes. Perante a isso, constata-se que o gargalo se encontra nos últimos enlaces da rede (Nam et al, 2014).

Existem algumas abordagens que tentam contornar estes problemas. Através da adaptação da taxa de *bits* de vídeo durante a reprodução, é possível minimizar a interrupção de vídeo e a quantidade de tempo gasto com armazenamento em *buffer*[4] (Georgopoulos et al, 2013). Se o *buffer* está vazio, a reprodução do vídeo é interrompida e ocorre uma estagnação do vídeo (Zinner et al, 2014).

Taxa de *bits* adaptativa (*adaptative bit rate* - ABR) de *streaming* sobre HTTP é tida como a abordagem de streaming padrão para muitos provedores de vídeo (*Video Providers* - VPs), como Netflix, Hulu e YouTube. Nos dias de hoje, *streaming* baseado em HTTP se tornou o protocolo de *streaming* de multimídia dominante. O VP gerencia a malha de controle de fluxo para aprimorar a QoE percebida pelo usuário, que é quantificada por diversas métricas, incluindo a latência de reprodução inicial, qualidade de vídeo recebida, e ausência de interrupções devido a estagnação de *streaming* (Quinlan et al, 2015). Baseada nessa abordagem, Georgopoulos et al (2013) traz *QoE Fairness Framework* (QFF) que maximiza consideravelmente a QoE dos usuários em redes multimídia.

Outras abordagens se baseiam em deixar a rede ciente do tráfego a fim de detectar fluxos de streaming. Isso pode ser alcançado a partir de uma NBI funcionando sobre RDS.

[4]Região de memória física utilizada para armazenar temporariamente os dados enquanto eles estão sendo movidos de um lugar para outro.

Por exemplo, Hue et al (2015) usa as estatísticas de fluxos em RDS para identificar cada fluxo. Como resultado é possível notar uma melhoria de desempenho em relação ao já citado método baseado em DPI. Os testes são feitos com vídeos do YouTube.

2.5 Mininet

Mininet pode ser definido a grosso modo como um sistema para emulação de redes. Esse sistema fornece uma maneira rápida de modelar grandes redes utilizando para tal apenas um simples computador e seus limitados recursos (Lantz et al, 2010). O Mininet foi criado com o próposito de possibilitar pesquisas na área de RDS em conjunto da utilização do OpenFlow (Kaur et al, 2014). Alguns atributos foram considerados requisitos indispensáveis durante o processo de concepção do Mininet (Lantz et al, 2010). Dentre eles se destacam:

- **Flexibilidade**: novas topologias e funcionalidades são definidos por *software*
- **Interatividade**: a execução e o gerenciamento da rede ocorre em tempo real
- **Escalabilidade**: o ambiente de modelagem da rede suporta redes em grande escala, isto é, redes com centenas ou milhares de *switches* e um único computador
- **Realismo**: o comportamento da rede modelada representa o comportamento real com um alto grau de confiança

Essas características fazem do Mininet um excelente ambiente de testes, além de ter custo baixíssimo e garantir a reprodutibilidade de experimentos.

Utilizando-se de uma abordagem de virtualização leve, o Mininet faz com que um único sistema se pareça com uma rede completa, executando o mesmo *kernel*, sistema e código de usuário. O Mininet se comporta como uma máquina real rodando Linux, fazendo com que seja possível realizar o acesso ao mesmo via ssh[5]. Com o Mininet não é necessário modelar os nós da rede individualmente, pois é possível criar topologias em larga escala apenas definindo sua estrutura e a quantidade de cada elemento a ser utilizado, dentre os quais estão disponívies *hosts*, *switches*, conexões e controladores (Kaur et al,

[5]https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

2014). A criação das topologias é realizada através de simples ferramentas na forma de linha de comando e também pode ser feita por meio de uma API. Uma vez que a rede tenha sido inicializada é possível interagir com a mesma através de uma interface de linha de comando (*Command Line Interface* - CLI) ciente da rede. A CLI disponibiliza uma variedade de comandos úteis além de permitir a invocação de janelas individuais para os hosts (*xterm windows* por meio de *X11 forwarding*). Como a CLI tem conhecimento dos nomes dos nós da rede e da própria configuração da rede, ela é capaz de substituir automaticamente os endereços IP dos *hosts* por seus nomes (Lantz et al, 2010). O Mininet fornece ainda uma API em *Python* para criação de experimentos e topologias, bem como a criação dos nós da rede.

O Mininet está disponível em forma de código livre e gratuito e pode ser acessado através do Github[6]. Juntamente com o Mininet são disponibilizados diversos exemplos de código que implementam alguns controladores simples com a finalidade de demonstrar os diferentes usos da API Mininet. Esses exemplos também podem ser usados como ponto de partida para a criação de funcionalidades mais complexas.

[6]https://github.com/mininet

3 Abordagem proposta

Todo o desenvolvimento e experimentos foram realizados dentro da máquina virtual do Mininet[7]. Como mencionado na seção 2.5, alguns códigos são disponibilizados em conjunto com o Mininet. Um desses códigos foi utilizado como ponto de partida para o desenvolvimento de um controlador. A ideia é que o controlador desenvolvido consiga oferecer uma maneira de realizar a comunicação com aplicações cliente. Dessa forma, em um cenário no qual existam alguns servidores disponíveis, estes poderão ter suas bandas de rede melhor gerenciadas ao realizar um reencaminhamento inteligente das conexões oriundas de diversos clientes.

3.1 Aplicações cliente e servidora

Foram desenvolvidas aplicações cliente e servidora apenas para fins de teste do arcabouço elaborado. Inicialmente foram criadas versões básicas, porém foi necessário adaptar a aplicação cliente para a utilização do controlador projetado. A aplicação cliente solicita um vídeo de determinada qualidade ao servidor. O servidor por sua vez tem a função de receber as requisições dos clientes e responde-las com o envio do arquivo correspondente.

As conexões entre cliente e servidor são feitas por *sockets* TCP. A aplicação cliente apresenta uma linha de comando ao usuário para que este possa informar o nome do arquivo e a qualidade separados por vírgula. Ao pressionar *enter*, uma conexão é estabelecida com o servidor para que a transmissão do arquivo seja realizada. O servidor aguarda por clientes e, ao receber uma nova conexão, é disparada uma *thread* que lida com tal conexão e assim o servidor volta a aguardar por novas conexões. Essa nova *thread* irá verificar se o nome do arquivo requisitado é válido, ou seja, se este arquivo existe no servidor bem como se a qualidade requisitada também é válida. Caso não haja nenhuma irregularidade com a requisição, o arquivo é enviado ao cliente. Enquanto isso o cliente aguarda por dados vindos do *socket*. A transmissão termina e a conexão por sua vez é

[7]http://mininet.org/download/

encerrada quando não são recebidos mais dados pelo *socket*. Nesse momento a aplicação cliente verifica se a quantidade de dados recebida tem tamanho igual a zero. Em caso afirmativo, significa que algo foi digitado incorretamente pelo usuário ou que o arquivo e/ou qualidade requisitados não existem no servidor em questão. No caso contrário, a transmissão do vídeo foi realizada com sucesso. Seja qual for o caso, a aplicação cliente fica preparada para uma nova solicitação do usuário.

A alteração necessária na aplicação cliente para utilizar o controlador desenvolvido constitui-se em realizar duas conexões *socket* por requisição ao invés de uma, pois uma conexão será usada para descobrir qual é o tipo de requisição do cliente e a outra será usada para de fato realizar a transmissão do arquivo requisitado. O motivo disso ficará mais claro posteriormente na seção 3.3. Além disso, a conexão inicialmente será enviada para um "servidor em um endereço virtual", uma vez que o controlador é quem decidirá de fato para qual servidor real a conexão será reencaminhada. Para o caso de uso idealizado isso não se torna um empecilho. Visto que o cenário de utilização imaginado é um no qual existem pelo menos um servidor com a função de fornecer vídeos de alta qualidade e outro com a função fornecer vídeos de qualidade média e baixa, a versão básica da aplicação cliente deve saber previamente quais são os endereços de cada servidor e também a função de cada um deles. Essa se torna uma evidente desvantagem da versão da aplicação cliente que não utiliza o controlador proposto, uma vez que ao utilizar tal controlador só será necessário ter conhecimento de um "endereço virtual" para o qual todas requisições, a princípio, serão enviadas. A figura 3.1 mostra a topologia de um típico cenário de uso das aplicações cliente e servidora.

3.2 Controlador básico

O código incial escolhido como ponto de partida para elaboração do controlador foi o "*l2_learning.py*"[8]. O próposito do controlador é funcionar como o "cérebro" do *switch* e, nesse código em específico, isso é feito através da catalogação de pares de endereço e porta. Ao receber um novo pacote, o *switch* irá verificar em suas regras de encaminhamento se existe alguma regra correspondente a esse pacote. Esses casamentos entre fluxos e regras

[8]https://github.com/att/pox/blob/master/pox/forwarding/l2_learning.py

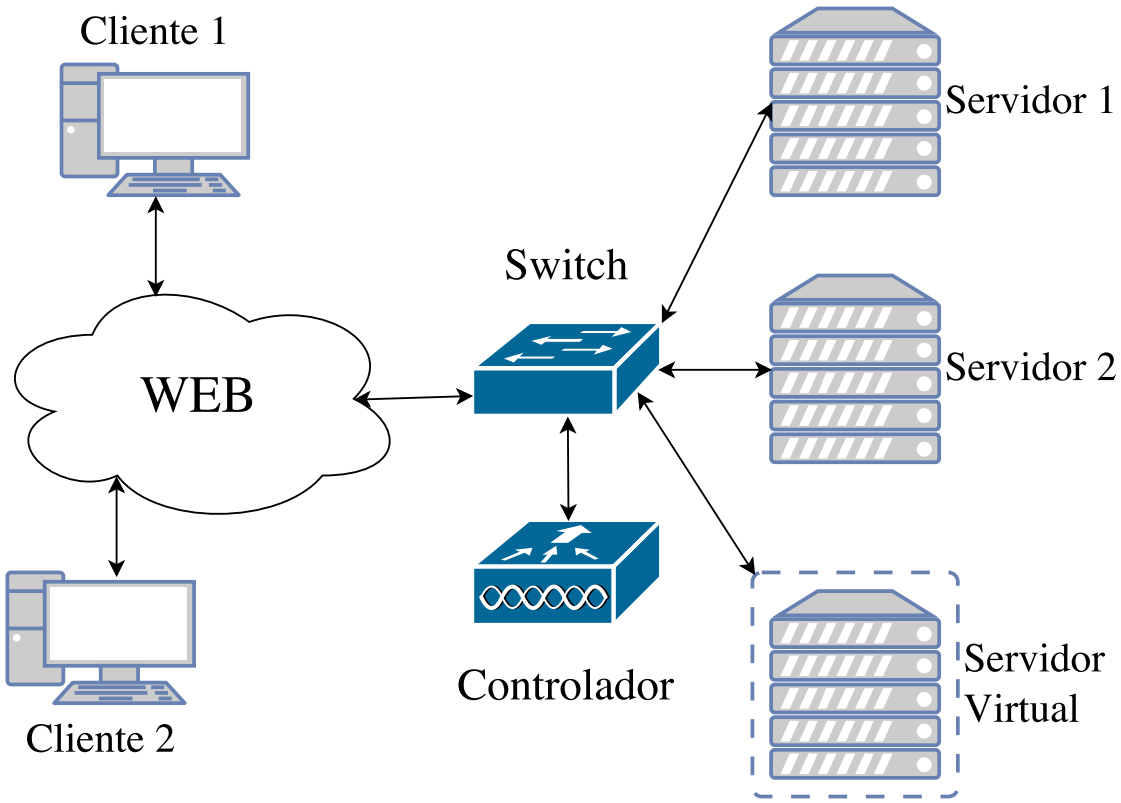

Figura 3.1: Topologia de um típico cenário de uso já incluindo o servidor virtual

são conhecidos como *match*. A verificação é feita ao comparar os valores dos campos do cabeçalho do pacote com os valores estabelecidos em cada *match*. Ao encontrar uma regra correspondente, o pacote é encaminhado normalmente. No caso negativo, o *switch* disparará um evento de entrada de pacote no controlador juntamente com o envio pacote em questão. O controlador por sua vez pode informar ao *switch* qual ação ele deve tomar em relação a este pacote. É nesse momento em que o código do controlador básico começa a ser executado.

Ao receber um novo pacote, o controlador verifica por qual porta do *switch* ele foi recebido e qual é seu endereço MAC de origem. Digamos que o controlador tenha recebido um pacote pela porta A e tenha endereço MAC de origem X. Então, através dessas informações, o controlador passa a ter conhecimento de que o *host* que possui o endereço MAC X pode ser alcançado através da porta A do *switch*, uma vez que foi recebido um pacote de tal *host* por tal porta. Esse conhecimento é armazenado em um dicionário cujas chaves são os endereços MAC aprendidos e os valores são as portas correspondentes. Em seguida, o controlador verifica se o endereço de destino desse pacote recebido já está presente como chave no dicionário. Isso é possível devido ao aprendizado

dos pares de MAC e porta de pacotes previamente recebidos. Suponhamos que o endereço MAC de destino do pacote seja Y. Ao encontrar Y como chave do dicionário, o controlador tem conhecimento de que aquele host, que possui endereço MAC Y, pode ser alcançado pela porta do *switch* presente no dicionário correspondente a este MAC. Denotaremos a porta correspondente ao MAC Y por B. O conhecimento relativo a esse par endereço-porta é portanto repassado ao *switch* por meio de uma nova regra de correspondência que modifica o fluxo do pacote. Essa nova regra diz ao *switch* que, a partir de agora, ao receber um pacote que tenha endereço de destino Y ele deve ser encaminhado para sua porta B. Dessa forma o *switch* não necessita contactar o controlador ao receber novos pacotes com destino Y.

Não obstante, ocorrem eventos de entrada de pacote no controlador quando o dicionário ainda está vazio e também quando o endereço de destino do pacote não está presente como chave do dicionário. Nesses casos o controlador instala um *match* no *switch* que diz a ele para enviar esse pacote específico para todas suas portas com exceção da porta de origem. Isso é conhecido como *flooding*. No caso em que o destino do pacote é *multicast* (requisições ARP por exemplo), o mesmo comportamento de *flooding* recém mencionado é realizado. Alguns outros casos relacionados a segurança, como ignorar pacotes com endereço MAC de origem e destino iguais, também são tratadas por esse código inicial, porém fogem do escopo deste trabalho e não serão explicados. A figura 3.2 mostra como seria o estabelecimento de uma conexão entre as aplicações cliente (endereço IP 10.0.0.4) e servidora (endereço 10.0.0.2), mencionadas na seção 3.1, utilizando o controlador básico descrito na presente seção.

3.3 Controlador proposto

O controlador proposto foi projetado tendo em mente seu funcionamento em conjunto com a aplicação de *streaming* desenvolvida. Entretanto, o método utilizado para deixar a rede ciente da aplicação mostrou-se flexível e pode ser adaptado para outras aplicações. Essa possibilidade é discutida juntamente com os trabalhos futuros no capítulo 6. O controlador desenvolvido é essencialmente o código do controlador básico com algumas funcionalidades adicionais que garantem à rede a característica de ser ciente da aplicação. Para conseguir

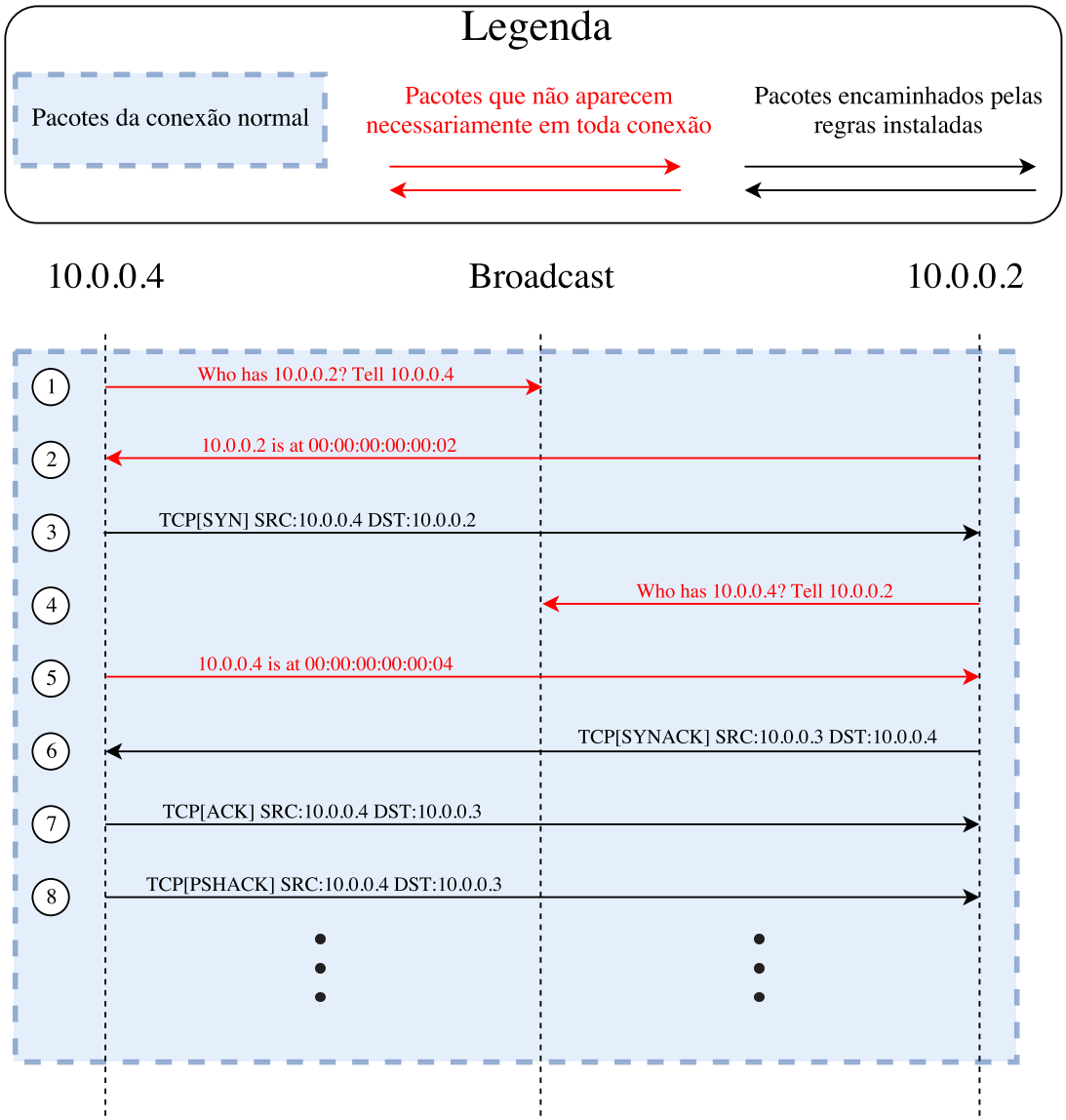

Figura 3.2: Início da transmissão de pacotes utilizando o controlador básico

essa característica, se faz necessário obter a informação do tipo de requisição almejada pelo cliente de alguma forma. Uma maneira de se obter isso é propor ao cliente que entregue essa informação explicitamente. A primeira vista essa ideia se parece trivial, porém sua execução não é tão simples. De certa forma, o que o controlador realiza pode ser resumido no algoritmo 1 abaixo.

Inicialmente foi estudada a possibilidade de sinalizar a conexão *socket* do cliente para o servidor por meio de alguma marca nos pacotes. Uma vez que as conexões que requisitassem um vídeo com qualidade melhor se diferenciassem de uma maneira conhecida das demais conexões, o controlador poderia identificar um pacote dessa conexão e dizer ao *switch* para reencaminha-lo para o servidor mais adequado. Contudo, como são utilizadas

Algoritmo 1: Estabelecimento da conexão entre cliente e servidor usando o controlador proposto

```
Entrada: pacote de um cliente ou servidor, (pacote).
1  início
2      se pacote.org = cliente e pacote.dst = serv_virtual então
3          se serv_selecionado = vazio então
4              criaConexãoAuxiliar(cliente)
5              tipo = extraiTipo(pacote)
6              serv_selecionado =selecionaServidor(tipo)
7              finalizaConexãoAuxiliar(cliente)
8          senão
9              instalaRegra(ALTERA_DST, pacote, serv_selecionado)
10             instalaRegra(REENCAMINHA, pacote)
11         fim se
12     senão
13         se pacote.org = serv_selecionado e pacote.dst = cliente então
14             instalaRegra(ALTERA_ORG, pacote, serv_virtual)
15             instalaRegra(REENCAMINHA, pacote)
16         fim se
17     fim se
18 fim
```

conexões TCP, existe a limitação relativa aos campos presentes no cabeçalho dos pacotes desse protocolo, o que impede a agregação de qualquer informação extra no formato de metadados. Tendo isso em mente, o único campo remanescente passível de personalização é o próprio campo de dados do pacote. Isso implica na utilização desse campo para que o cliente informe ao controlador qual sua requisição. Até o momento isso não acarreta em nenhuma alteração na aplicação cliente, uma vez que a informação do tipo de qualidade desejada pelo cliente já está presente no campo de dados.

O TCP é um protocolo orientado a conexão, isso significa que a conexão é inicialmente estabelecida seguindo uma sequência de passos bem definida. Essa sequência de passos para o estabelecimento da conexão é conhecida como "*3-way handshake*" (Postel et al, 1981). O *3-way handshake* funciona da seguinte forma: o cliente envia um pacote com a *flag* SYN (*synchronize*) ativa, o servidor por sua vez recebe esse pacote e o responde com um pacote com as *flags* SYN e ACK (*acknowledgement*) ativas, por fim o cliente responde esse pacote vindo do servidor com um pacote com a *flag* ACK ativa (Postel et al, 1981). Nesse momento o cliente pode iniciar o envio de pacotes com os dados. Já no primeiro pacote estará presente a informação da requisição do cliente.

Esses 3 pacotes trocados durante o estabelecimento da conexão, por meio do *3-way handshake*, não possuem os campos de dados. Isso resulta na impossibilidade de detectar qual é a qualidade do arquivo desejado pelo cliente logo no primeiro pacote, quando a conexão ainda não está de fato estabelecida. Dessa forma se faz necessário o estabelecimento da conexão com o cliente para em seguida ter acesso a sua requisição. O controlador pode dizer ao *switch* para responder ao primeiro pacote do *3-way handshake*, vindo do cliente, com um pacote que o cliente está esperando, ou seja, um pacote contendo as *flags* SYN e ACK ativas e, dessa forma, simular o estabelecimento da conexão entre o cliente e um suposto "servidor virtual". Nesse momento o cliente prosseguirá com o *3-way handshake* e enviará um pacote com a *flag* ACK ativa. O controlador pode dizer ao *switch* para simplesmente ignorar este pacote. Em sequência será enviado pelo cliente o pacote contendo sua requisição de fato. Esse pacote finalmente poderá ser analisado pelo controlador. Essa análise é bem simples: basta extrair do campo de dados o número que indica a qualidade do arquivo pretendido pelo cliente. Esse número é aquele mesmo número digitado pelo usuário na linha de comando da aplicação cliente. Uma vez que o controlador possui essa informação, ele pode finalmente definir para qual servidor real essa conexão será reencaminhada. Porém, um novo problema surge neste ponto.

Como mencionado, ao utilizar essa abordagem foi preciso que o *switch* forjasse uma conexão TCP com o cliente. Isso requer que o cliente defina um número de sequência para seus pacotes, bem como é indispensável que o *switch* estabeleça um número de sequência para o seu próprio pacote, que foi usado durante o estabelecimento da conexão. O número de sequência em conjunto com a *flag* ACK são usados pelo protocolo TCP como uma das técnicas para garantir a entrega confiável de pacotes, assim sendo estes campos são obrigatórios no cabeçalho dos pacotes (Postel et al, 1981). Portanto, o problema citado se resume em: como estabelecer uma conexão com um servidor real e unifica-la com a conexão já estabelecida com o cliente. Isso porque o número de sequência definido pelo servidor real será aleatório e é praticamente impossível que este seja igual ao número já definido pelo *switch* e já esperado também pelo cliente.

Este problema pode ser evitado simplesmente ao não unificar conexão alguma. Como o controlador já tem a informação da requisição do cliente, este pode dizer ao

switch para finalizar a conexão forjada com tal cliente. O cliente, por sua vez, pode logo em seguida enviar a mesma requisição para o estabelecimento de uma nova conexão que, desta vez, poderá ser estabelecida diretamente com o servidor real previamente escolhido pelo controlador. Esse artífício é que faz com que a aplicação cliente tenha de ser alterada. Ao invés de uma única conexão, o cliente agora terá que fazer duas conexões: a primeira conexão servirá apenas para que o controlador obtenha conhecimento do tipo de requisição almejada pelo cliente e, a segunda conexão será de fato usada para transmitir o arquivo na qualidade desejada. As duas conexões estabelecidas pelo cliente consistem do envio da mesma requisição. A figura 3.4 ilustra os pacotes trocados no início de uma nova conexão utilizando o controlador projetado. As figuras 3.2 e 3.4 foram esquematizadas baseando-se em observações das transmissões por meio do programa Wireshark[9].

Enfim, ao fazer uso do controlador projetado, o funcionamento da rede se dá da seguinte maneira: ao receber um pacote, o *switch*, quando não possuir uma regra que corresponda a tal pacote recebido, sempre irá disparar um evento de entrada de pacote para o controlador projetado (assim como acontece no uso do controlador básico). O controlador além de funcionar da maneira básica, catalogando os pares de endereço e porta, irá armazenar em uma lista um outro dicionário contendo endereço IP, endereço MAC e porta de onde o pacote se originou. Em adição, os dicionários dessa lista também possuem campos para armazenar as informações de IP, MAC e porta do servidor, que são preenchidos quando o controlador escolhe para qual servidor a requisição do cliente será enviada. Por último, mas não menos importante, existe um campo no dicionário que armazena o estado da conexão do cliente. Esse estado é o que guia o controlador ao receber um novo pacote do cliente. A máquina de estados correspondente às alterações desse estado está ilustrada na figura 3.3.

Ao receber pacotes que possuem endereço de destino corresponde ao endereço do "servidor virtual", o controlador toma ações baseando-se no estado atual registrado no item do dicionário cuja chave é igual ao endereço de origem do pacote. O estado é inicializado como "Esperando SYN". Ao receber um pacote com a *flag* SYN, o controlador instala um regra no *switch*, que diz a ele para responder esse pacote com um pacote com

[9]https://www.wireshark.org/

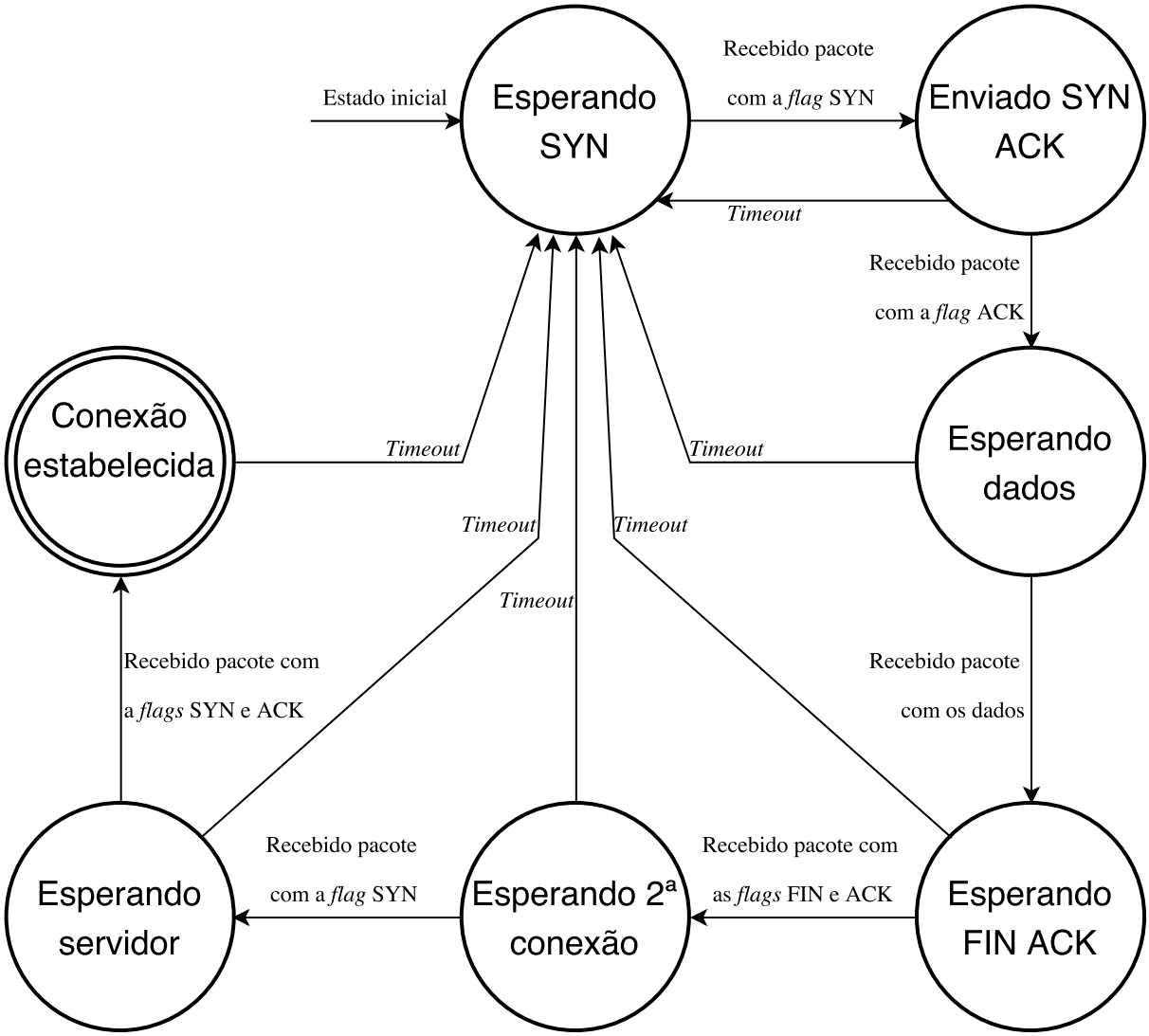

Figura 3.3: Máquina de estados correspondente aos possíveis estados da conexão entre cliente e servidor

as *flags* SYN e ACK e altera o estado para "Enviado SYN ACK". Em sequência, ao receber um pacote com a *flag* ACK, o controlador apenas altera o estado para "Esperando dados". Ao receber o pacote de dados do cliente, o controlador identifica o tipo de requisição e escolhe o servidor. Nesse momento, o item do dicionário relativo ao cliente que está enviando estes pacotes tem seus campos de MAC, IP e porta, correspondentes às informações do servidor, preenchidos. As informações dos servidores disponíveis, bem como suas funções, consistem de um conhecimento prévio codificados diretamente no algoritmo do controlador. Em adição, o controlador também diz ao *switch*, por meio de uma nova regra, que encerre a conexão forjada com o cliente enviando um pacote com as *flags* FIN e ACK ativas. O estado é alterado para "Esperando FIN ACK". Quando o *switch* receber um pacote também com as *flags* FIN e ACK ativas, o controlador apenas

irá alterar o estado para "Esperando 2ª conexão" e aguardará pelo estabelecimento da segunda conexão. Então, ao receber mais um pacote com a *flag* SYN ativa, o controlador instala uma regra no *switch* que diz a este para alterar o destino desse pacote e dos próximos pacotes semelhantes a este. O novo destino do pacote será aquele endereço do servidor escolhido pelo controlador. Essa regra também diz ao *switch* para reencaminhar o pacote pela sua porta que leva para tal servidor. O estado é alterado para "Esperando servidor". Finalmente, ao receber um pacote com as *flags* SYN e ACK tendo como endereço origem e endereço destino respectivamente os endereços de um servidor e um cliente que estejam registrados em um mesmo item da lista de dicionários, o controlador instala um regra no *switch* para que este modifique o endereço de origem do pacote e reencaminhe este pacote para a porta pela qual o cliente pode ser encontrado. Nesse momento, a conexão se estabelece e o *switch* contém as regras necessárias para lidar com esta. Ao chegar no estado final, "Conexão estabelecida", o item que contém as informações dessa conexão recém estabelcida é removido do dicionário.

Se durante o processo de estabelecimento da conexão houver uma demora maior que 5 segundos para o recebimento de um novo pacote, o estado é redefinido para o estado inicial novamente. Isso garante que o controlador seja capaz de lidar com encerramentos abruptos de conexão antes do estabelecimento da conexão entre um cliente e um servidor real. Todas as regras instaladas no *switch* também possuem tempo de expiração e, quando esse tempo é atingido, a regra é removida. Os tempos de expiração são de dois tipos: ocioso e rígido. A regra sempre é removida quando o tempo de expiração rígido é atingido. Já o tempo de expiração ocioso resulta na remoção da regra quando esta não é utilizada em tal intervalo de tempo. Os tempos de expiração rígido e ocioso para todas as regras foram, respectivamente, 30 segundos e 10 segundos. Os tempos de expiração garantem que a tabela de regras do *switch* não fique lotada com regras que não estão sendo mais utilizadas.

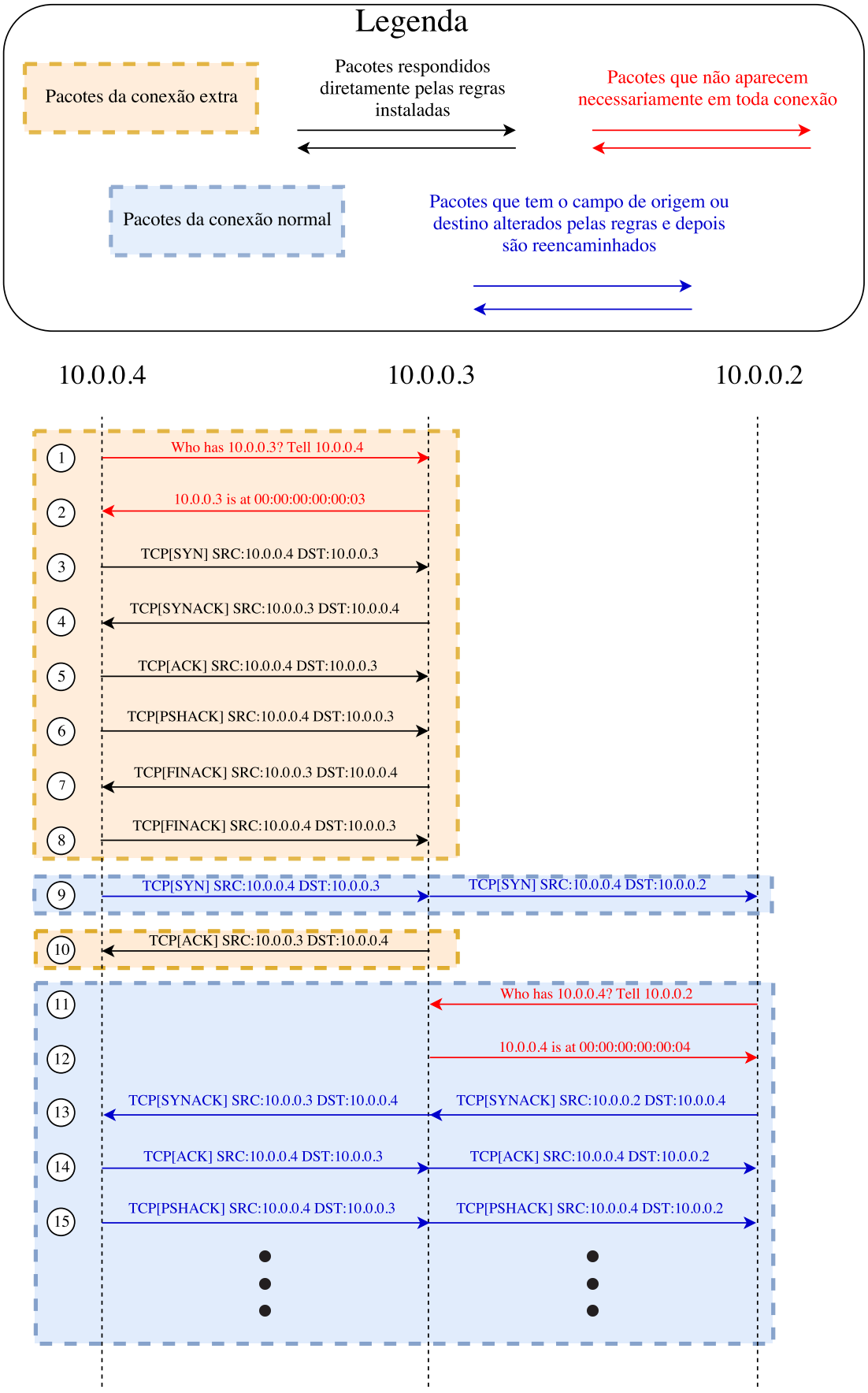

Figura 3.4: Início da transmissão de pacotes de uma conexão TCP. O cliente, o servidor virtual e o servidor real escolhido pelo controlador são representados respectivamente pelos endereços 10.0.0.4, 10.0.0.3 e 10.0.0.2.

4 Experimentos

Evidentemente existe uma sobrecarga extra na rede ao utilizar o controlador projetado, pois é necessária uma conexão adicional para tornar a rede ciente da aplicação. Para averiguar o impacto dessa sobrecarga, um simples teste foi realizado. Foi utilizada uma topologia contendo 12 *hosts*: 10 destes executando a aplicação cliente e 2 executando a aplicação servidora. Um desses servidores teve a função de responder as requisições de arquivos de qualidade alta, e o outro teve a função de responder as requisições de qualidade média e baixa. Foram executadas 30 rodadas de requisições. Em cada rodada, cada *host* cliente executou 3 requisições pelo mesmo arquivo, sendo uma de cada tipo de qualidade (alta, média e baixa). Dessa maneira, a cada rodada foram realizadas 30 requisições, sendo 10 de cada tipo. Ao final das 10 rodadas, 900 requisições haviam sido realizadas, sendo 300 de cada tipo. Esse teste foi realizado em um cenário utlizando o controlador básico e também em um cenário utilizando o controlador projetado. O primeiro cenário será chamado de "requisição normal" enquanto o segundo será chamado de "requisição ciente". O tempo de todas as transmissões foi registrado. Todas transmissões foram executadas sequencialmente, dessa forma nenhum tempo adicional de congestionamento contribuiu para o aumento do tempo registrado em cada transmissão.

Foi elaborado um *script* em *Python* para executar as requisições de forma automatizada. O *script* faz uso da interface do Mininet disponível para *Python* e assim realiza a configuração da topologia utilizada nos testes. A API do Mininet permite ainda que sejam executadas tarefas individalmente em cada *host*. Essa funcionalidade fez com que fosse possível a execução das aplicações cliente e servidora nos *hosts*.

Nos gráficos das figuras 4.1, 4.2 e 4.3 são mostradas respectivamente as informações relativas às transferências dos arquivos de qualidade baixa, média e alta. Os arquivos correspondentes a essas 3 qualidades possuem tamanho respectivamente igual a 43.9 MB, 82.3 MB e 127 MB. Em cada gráfico são ilustrados os tempos médios registrados das requisições normais e cientes. Também são mostrados os intervalos de confiança, para 95% de nível de confiança, extraídos a partir do desvio padrão do conjunto de amostras

obtido.

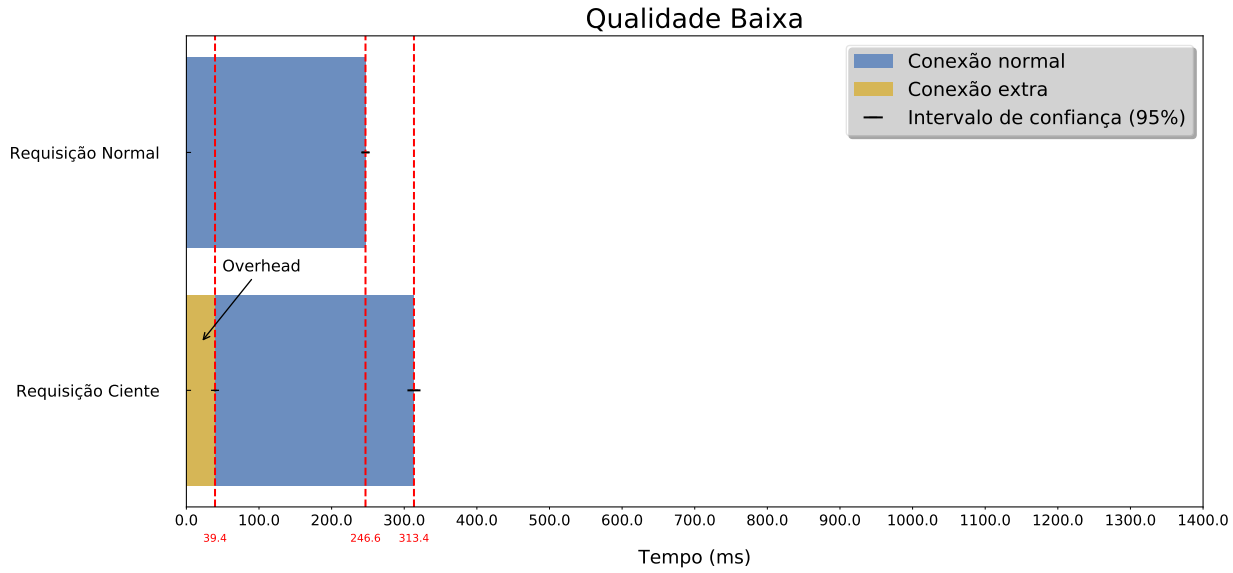

Figura 4.1: Média de tempo juntamente com o intervalo de confiança das requisições por qualidade baixa nos dois cenários propostos. "Conexão normal" e "Conexão extra" são referências às figuras 3.2 e 3.4

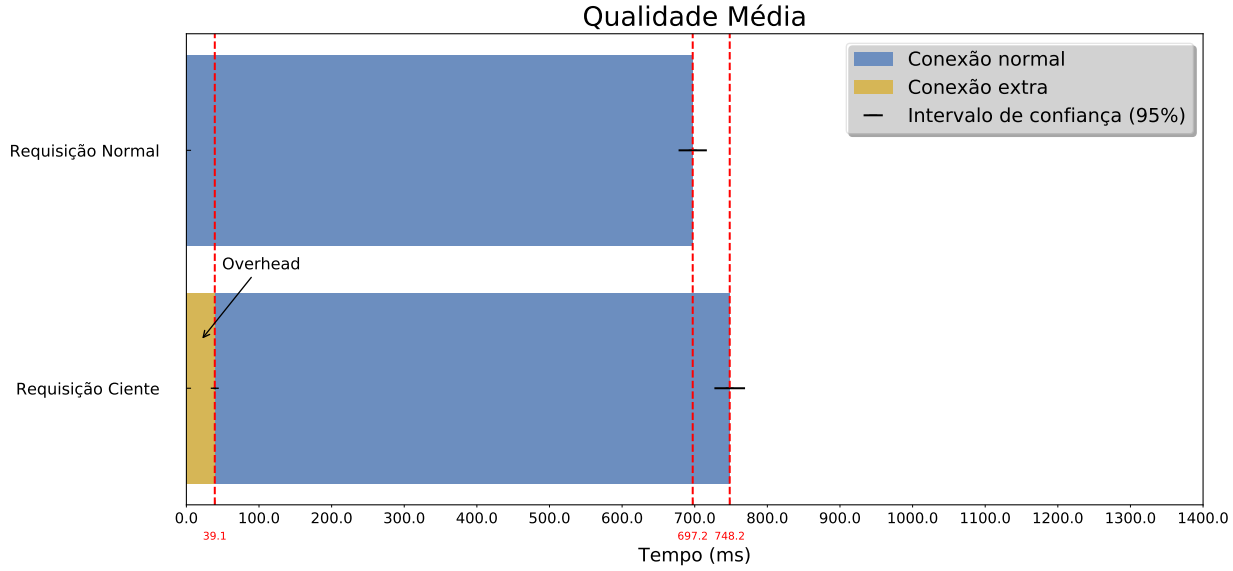

Figura 4.2: Média de tempo juntamente com o intervalo de confiança das requisições por qualidade média nos dois cenários propostos. "Conexão normal" e "Conexão extra" são referências às figuras 3.2 e 3.4

Em todos os gráficos, como era de se esperar, o tempo médio da requisição ciente é maior em comparação com a requisição normal. Além disso, os 3 gráficos também tem em comum o tempo de sobrecarga (*overhead*) devido a conexão extra (esse tempo de sobrecarga da qualidade alta está ligeiramente acima dos demais, porém a diferença é insignificante). Isso se deve ao fato de que a conexão extra, que serve para garantir que a rede fique ciente da aplicação, sempre consiste da mesma quantidade de pacotes.

Tabela 4.1: Intervalos aproximados de confiança das requisições para o nível de 95%

	Requisição normal	**Requisição ciente sem sobrecarga da conexão extra**
Qualidade alta	1208.0 ms a 1261.2 ms	1266.5 ms a 1339.2 ms
Qualidade Média	677.7 ms a 716.7 ms	688 ms a 730.2 ms
Qualidade baixa	241.0 ms a 252.2 ms	265.0 ms a 283.0 ms

Esse comportamento também já era previsível. Ademais é evidente o maior impacto que a sobrecarga da conexão extra exerce sobre uma requisição por qualidade baixa. Uma vez que o arquivo de baixa qualidade possui o menor tamanho, este será transmitido utilizando-se um número menor de pacotes em comparação com as outras duas qualidades. Dessa forma, percentualmente, o tempo de sobrecarga é maior em comparação com os demais.

Por fim, ao analisar os tempos registrados nos gráficos é possível chegar a conclusão de que existe uma outra sobrecarga no tempo da requisição ciente além da conexão extra. Em todas as 3 situações, ao subtrair do tempo médio da requisição ciente o tempo de sobrecarga da conexão extra, o resultado ainda é maior do que o tempo da requisição normal. É razoável acreditar que a alterações dos endereços de origem e destino, realizadas pelo *switch* no cenário no qual é utilizado o controlador projetado, acarreta em uma pequena sobrecarga nos tempos totais de transmissão de todas requisições. Entretanto, essa pequena sobrecarga é extremamente sútil e isso pode ser assegurado pelo intervalo de confiança, que demonstra que os tempos dos dois cenários ficam bastante próximos na maioria das vezes. Através da tabela 4.1 ainda é plausível realizar outra afirmação que reforça a insignificância que a diferença entre os tempos representa. Para o caso do arquivo com a qualidade média, ao desconsiderar o tempo de sobrecarga da conexão extra, as duas abordagens não são diferentes para o nível de confiança apresentado. Isso é justificado pela média de ambas requisições estarem contidas nos intervalos de suas contrapartes. Logo, a utilização do controlador proposto não acarreta na degradação da performance da rede e carrega consigo a vantagem da obtenção de informações sobre o tráfego de uma forma simples porém poderosa.

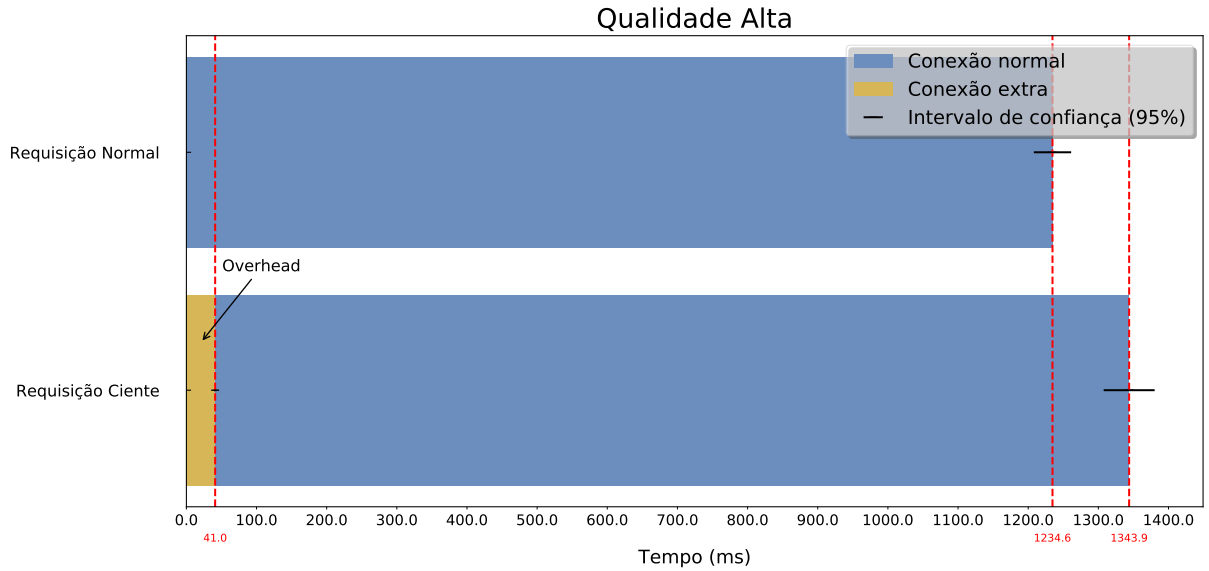

Figura 4.3: Média de tempo juntamente com o intervalo de confiança das requisições por qualidade alta nos dois cenários propostos. “Conexão normal” e “Conexão extra” são referências às figuras 3.2 e 3.4

5 Revisão da Literatura

Redes Definidas por Software não é um tema novo dentro do contexto de redes de computadores. Existem diversos trabalhos na área que fizeram contribuições interessantes nesse assunto. A partir da leitura de artigos atuais que tratam do mesmo tema estudado neste trabalho, foi elaborado um levantamento contendo os trabalhos que mais se relacionam de alguma forma com a proposta e objetivo deste trabalho. Os aspectos semelhantes e divergentes em relação a abordagem desenvolvida foram expostos e comentados. Esse levantamento tem a finalidade de localizar espacialmente esse trabalho na literatura.

5.1 Trabalhos Relacionados

As técnicas utilizadas para realizar a comunicação entre o controlador e a aplicação variam desde a informação direta enviada pela aplicação a inferência do tipo de fluxo de dados. Tal tipo de fluxo de dados pode ser inferido a partir da observação de características típicas de *streaming* de vídeo, uma vez esta possui características bem peculiares a serem ressaltadas mais a frente. Conteúdo de vídeo, principalmente vídeos de alta qualidade em tempo real tem ocupado uma grande fatia do tráfego da Internet. Tratar essas cargas de trabalho se tornou uma difícil tarefa, mesmo para uma grande CDN (Chen et al, 2016). Para solucionar este problema os algoritmos de balanceamento de carga são fundamentais (Costa et al, 2016). No entanto, os algoritmos tradicionais de balanceamento de carga, como *Round-Robin* e randomização, não têm conhecimento das exigências do lado do usuário. Portanto, não é incomum que os pedidos de vídeos de alta qualidade em tempo real não sejam satisfeitos. Neste contexto, o trabalho de Chen et al (2016) tenta atender a esses pedidos integrando a tecnologia de rede definida por *software* com a infraestrutura CDN.

Com um controlador RDS e um CDN baseado em RDS, o balanceador de carga é capaz de coletar estatísticas de largura de banda em tempo real de todos os *links* no CDN. Por exemplo, o balanceador de carga recebe um pedido de largura de banda de

1,2 Gbps. Então, controlador analisa dentre todos os *links* disponíveis, quais atendem a exigência de vídeo do cliente. O balanceador de carga responde ao cliente com um ou vários servidores para garantir a largura de banda de 1,2 Gbps. Com a arquitetura de rede flexível e encaminhamento baseado em fluxo, é fácil para o balanceador de carga recuperar servidores que atendam às solicitações dos usuários (Chen et al, 2016).

Conforme ilustrado na figura 5.1 (Chen et al, 2016), um cliente solicita um conteúdo de vídeo em tempo real que requer largura de banda de 2.9 Gbps. Existem três servidores disponíveis e sua largura de banda não utilizada são 2.1, 1.5 e 1.9 Gbps respectivamente. É assumido que o *downlink*[10] do cliente tem a capacidade de suportar esta exigência de largura de banda. No balanceamento de carga tradicional, o pedido pode ser redirecionado para o servidor 1, uma vez que o servidor 1 tem a carga de trabalho mínima. Contudo, a balanceador de carga não tem conhecimento de que esta atribuição não será capaz de satisfazer o pedido. No trabalho proposto por Chen, o balanceador de carga é integrado a um controlador RDS, que permite o balanceador conhecer a taxa de link de cada servidor. Portanto, o balanceador pode atribuir a solicitação com dois servidores: servidor 2 e servidor 3. Nesse caso, a exigência de largura de banda mínima é alcançada (Chen et al, 2016).

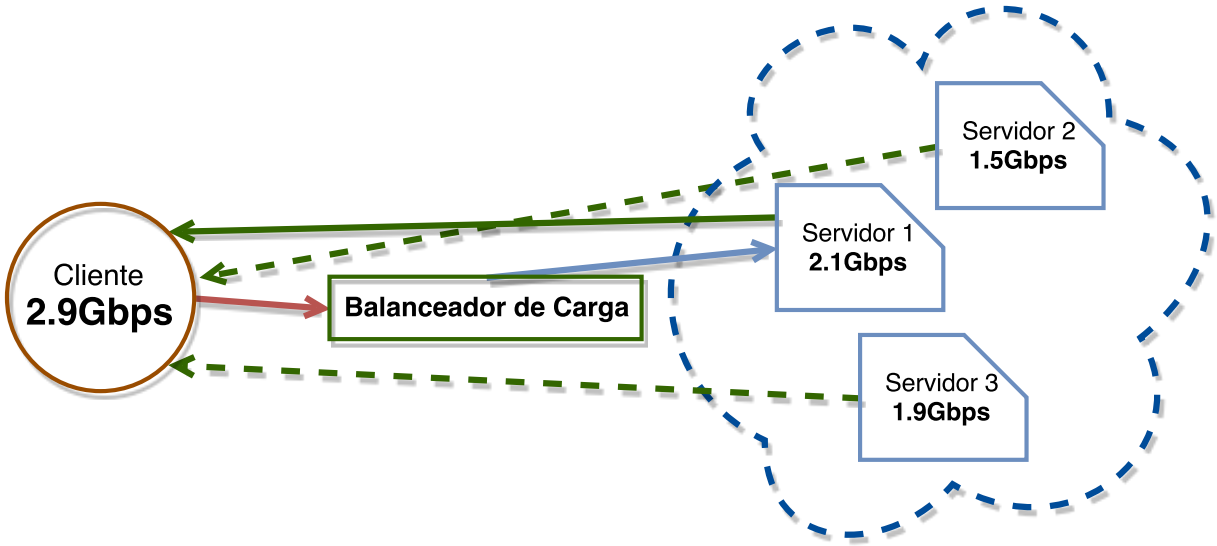

Figura 5.1: Exemplo motivacional adaptado de Chen et al (2016)

Essa lógica utilizada para a escolha do melhor servidor que atenda às exigências

[10]Parte da conexão de rede responsável por receber dados de um servidor remoto

do fluxo de vídeo, é transformada em um algoritmo que por sua vez é implementado diretamente no controlador RDS que então, trabalha em conjunto com o CDN a fim de deixa-lo ciente da aplicação. No presente trabalho, uma verificação semelhante a esta foi elaborada para atribuir eficientemente cada fluxo de vídeo a um servidor que possua o conteúdo desejado. Entretanto, algumas diferenças existem. Apesar de ser possível utilizar o controlador projetado como um balanceador de carga, essa característica não foi explorada neste trabalho mas será analisada nos trabalhos futuros como discutido no capítulo 6. Outra característica divergente é a diferença no tipo de informação passada pelo cliente. Enquanto que no trabalho de Chen isso é passado de forma quantitativa, na abordagem proposta isso é alcançado de forma qualitativa. A forma qualitativa possui a vantagem de simplificar a elaboração da aplicação cliente, pois muitas vezes não há como o cliente prever de forma exata qual será a carga atribuída a requisição.

No trabalho de Chen et al (2016), os esforços foram concentrados tanto na carga de trabalho do servidor quanto na satisfação do usuário. A motivação principal fundamenta-se no fato de que as abordagens de balanceamento de carga tradicionais não conseguem acompanhar o rápido crescimento da qualidade de vídeo. Por meio de simulações, foi observado que o trabalho proposto por Chen tem o potencial de satisfazer a exigência de largura de banda de *streaming* de vídeo em tempo real e utilizar totalmente os *links* em CDNs.

Para conceber o principal objetivo deste trabalho que, como já mencionado, se trata de demonstrar a factibilidade da criação de uma API *Northbound*, é necessário especificar suas possíveis funcionalidades. Alguns trabalhos na literatura realizam essa especificação analisando a demanda de QoS que desejam atingir. Um exemplo disso é visto no trabalho de (Humernbrum et al, 2014), que obtém como resultado da análise, uma lista de características desejadas na API. Mesmo que estas características mencionadas tenham sido obtidas a partir da análise dentro do escopo de ROIA (*Real-Time Online Interactive Application*), elas também são pertinentes para o escopo das aplicações de *streaming* de vídeo, uma vez que os objetivos de ambos os escopos se alinham na questão de melhoria de desempenho.

As funcionalidades da API *Northbound* especificadas para ROIA, que também

são de interessantes e podem ser usadas pela a aplicação de *streaming* desenvolvida neste trabalho, são listadas a seguir juntamente com uma possível utilização:

1. **Atualizar requisitos da aplicação:** quando o usuário altera a qualidade de vídeo que deseja assistir, o controlador deve ser capaz que alterar a largura de banda reservada para o cliente.

2. **Permitir especificação de requisitos antecipadamente:** ao término da reprodução de um vídeo de uma *playlist*, o seguinte vídeo pode necessitar de requisitos diferentes do anterior

3. **Definir requisitos baseados em tempo:** na mesma situação do item anterior, o início do próximo vídeo deve ser armazenado em *buffer* antes que o vídeo anterior acabe, a fim de obter uma transição entre os vídeos sem ocorrer estagnação

A aplicação de *streaming* desenvolvida para testar o controlador proposto no presente trabalho não conta com um reprodutor de vídeo, pois o foco do trabalho está na transmissão dos arquivos. Contudo, algumas dessas funcionalidades acima enumeradas já foram concebidas e podem ser utilizadas por uma aplicação de *streaming* com um reprodutor de vídeo integrado. A funcionalidade 1 é entregue ao tratar cada requisição de vídeo de um cliente como uma conexão diferente. Dessa forma, diferentes conexões de um mesmo cliente podem ser encaminhadas para servidores diferentes a assim prover arquivos de qualidades também diferentes. A funcionalidade 2 é possível pois o *socket* utilizado na aplicação é não-bloqueante, permitindo então realizar mais de uma conexão ao mesmo tempo. Uma vez que a funcionalidade 1 e 2 são oferecidas, a funcionalidade 3 depende apenas de modificar a aplicação cliente, pois a mesma pode ser derivada através das duas primeiras funcionalidades.

Através dessas funcionalidades mencionadas e, juntamente com as demais funcionalidades específicas ao contexto de ROIA, esse trabalho mencionado consegue adicionar um nível a mais de abstração através da API *Northbound*. Este novo nível de abstração atingido garante que a comunicação entre controlador e aplicação seja de alto nível, facilitando sua utilização e atraindo mais desenvolvedores para a área.

Com RDS, um método particularmente interessante para permitir gestão de

tráfego em redes tornou-se disponível. No trabalho de Jarschel et al (2013) é implementada e investigada uma abordagem baseada em *Deep Packet Inspection* (DPI) e uma abordagem com base na informação direta fornecida pelo aplicativo. Essas abordagens são utilizadas em um cenário de uso com OpenFlow, tendo como principal objetivo mostrar como estes tipos diferentes de informações sobre a aplicação, podem ser explorados para melhorar a QoE.

Esta segunda abordagem é muito semelhante com a que é utilizada pelo controlador proposto neste trabalho. Ainda em relação a essa segunda abordagem, outra diferença se encontra no caso de uso. Enquanto que neste trabalho presente na literatura é utilizado como cenário de uso o YouTube, o cenário de uso aqui proposto foi construído a partir de uma aplicação de *streaming* desenvolvida especificamente para se beneficiar do arcabouço desenvolvido, também elaborado neste trabalho. Em suma, a comparação é válida uma vez que o YouTube também possui características de *streaming* de vídeo.

A abordagem com base na informação direta fornecida pelo aplicativo consiste em informar ao controlador quando o *buffer* ficar abaixo de determinado de limite e, o controlador por sua vez, deve executar alguma ação em relação ao fluxo de dados, para que a QoE percebida pelo usuário não seja prejudicada. A ação tomada pelo controlador, quando o limite mínimo de dados em *buffer* é atingido, consiste em redirecionar o tráfego para um *link* menos sobrecarregado. Dessa forma, as solicitações desse fluxo serão atendidas de forma mais rápida. Uma vez que o *buffer* deste vídeo atingir um limiar satisfatório de dados, o fluxo de dados deste vídeo pode retornar ao *link* anterior, já que a QoE do usuário não correrá mais riscos de ser prejudicada.

Em comparação com a abordagem baseada em DPI, também usada neste trabalho da literatura, a abordagem utilizando a rede ciente da aplicação tem a vantagem de não necessitar de um *link* dedicado para o *streaming* de vídeo, uma vez que o *link* menos sobrecarregado não precisa ser exclusivamente utilizado por apenas um fluxo de dados. Entretanto, a abordagem onde a rede fica ciente da aplicação gera uma sobrecarga adicional na rede em relação à abordagem baseada em DPI, pois requer uma instância de computação adicional para receber e filtrar as informações da aplicação para o controlador. No trabalho aqui apresentado, foi realizada uma tentativa de evitar a sobrecarga da

rede por meio do pré-estabelecimento de perfis de fluxo diminuindo a necessidade de processamento dos dados. Embora a abordagem apresentada ainda tenha feito a transmissão sofrer de sobrecarga, esta ocorreu em uma escala muito pequena.

Diferente do método que utiliza informação fornecida diretamente pela aplicação, outra abordagem encontrada na literatura tenta inferir o tipo de fluxo de dados. A partir da identificação do fluxo, o trabalho de Hue et al (2015) realiza o roteamento dos dados chegando a diminuir a latência em até 75% e aumenta a taxa de identificação correta de fluxo em até 138% quando comparada com a já mencionada técnica que utiliza de DPI.

A técnica empregada pela maioria dos serviços de transmissão de vídeo provoca um padrão de tráfego de ciclo *ON-OFF* com uma duração fixa que é diferente de outras atividades (por exemplo, navegação na *web*). Como esses períodos tem sempre uma mesma duração, é possível identificá-los de maneira simples. Na técnica proposta por Hue et al (2015), as estatísticas da rede definida por *software* são solicitadas a cada tempo de recuperação de estatísticas (*statistics retrieval time* - SRT), que é uma variável manipulada no estudo dele. Ao monitorar o tamanho dos dados transmitidos em SRT mostrados nas estatísticas de RDS, um mecanismo é projetado com a função de sinalizar os fluxos que parecem corresponder ao padrão de tráfego dos serviços de *streaming* de vídeo.

A partir disso é possível chegar à conclusão de que diminuir o SRT tanto quanto possível trará uma melhoria maior nos resultados, porque os dados de tráfego coletados se aproximarão de uma situação real. No entanto, ao abaixar o SRT a frequência com a qual o controlador solicitará as estatísticas de RDS aumentará proporcionalmente e, portanto, deixará o controlador e o *switch* mais lentos. Então, existe uma preocupação a respeito do *trade-off*[11] entre a precisão de acerto nos padrões de tráfego e a carga no controlador e no *switch*.

Como resultado, os experimentos de tal trabalho mostram que, com determinado SRT, a técnica que foi proposta pode reduzir a latência para reconhecer o fluxo de vídeo e alcançar uma maior taxa de sucesso. A taxa de sucesso alcançada mostra que, em comparação com DPI, o método concebido no trabalho de Hue et al (2015) é um

[11]Relação entre ganho e perda em uma escolha conflitante

forte concorrente para identificar dados de tráfego. Outra vantagem é sua complexidade baixíssima em comparação com DPI, tornando-se mais fácil de ser estendida e implantada. Tal simplicidade será também o foco na elaboração da API *Northbound* proposta em meu trabalho.

Diversos trabalhos na área de redes também utilizam do Mininet como ambiente de testes principalmente por facilitar a reprodutibilidade de experimentos. Em Handigo et al (2012) é proposta ainda uma extensão para o Mininet, que é batizada de Mininet Hi-Fi. Essa nova implementação tem como características principais o isolamento de performance, providenciamento de recursos e monitoramento para garantir fidelidade de performance. O Mininet por si só já apresenta uma maneira simples e eficiente para ser utilizado como *testbed*[12]. Isso pode ser visto no trabalho de Lantz et al (2010) no qual o Mininet é usado para criar *testbeds* virtuais durante o funcionamento da rede. Nesse mesmo trabalho mencionado é criada uma classe servidor que estende a classe *host* do Mininet, com a intenção de deixar o *testbed* virtual mais convincente. A capacidade de criação de *testbeds* também foi explorada por meu trabalho durante a realização dos testes de desempenho da nova lógica do controlador apresentado, porém de forma mais simples. A distinção entre cliente e servidor está presente em meio a execução das aplicações com os mesmos nomes. Isso garante uma maior flexibilidade dentro de uma mesma topologia. Entretanto, da forma como foi implementado, é necessário editar o código do controlador para que o mesmo possa ter conhecimento de quais nós são os servidores. Na tabela 5.1 são mostradas, de forma resumida, as principais semelhanças e diferenças relacionadas a RDS entre este trabalho e aos trabalhos mencionados.

Conforme foi mencionado em um dos trabalhos da literatura, antes de qualquer desenvolvimento de uma API *Northbound* é necessário estabelecer suas funcionalidades de forma qualitativa, para então estabelecer os parâmetros a serem comunicados entre controlador e aplicação. As funcionalidades devem fornecer no mínimo uma troca de informações que permita que a rede fique ciente da aplicação. Como podem ser notadas, as abordagens que se baseiam em deixar a rede ciente da aplicação podem aumentar o desempenho da transmissão de dados, resultando em uma melhoria significativa na

[12]Plataforma para conduzir rigorosos testes de novas tecnologias além de ser transparente e garantir a replicabilidade

Tabela 5.1: Comparação entre os trabalhos da literatura a este trabalho

	Características dos trabalhos da literatura	**Semelhanças e diferenças em relação a este trabalho**
Chen et al (2016)	Rede tem acesso a informação da requisição de forma quantitativa	Rede tem acesso a informação da requisição de forma qualitativa
	Controlador funciona como balanceador de carga	Existe a possibilidade de ser usado como controlador, bastaria recuperar as estatísticas das bandas disponíveis em cada servidor
Humernbrum et al (2014)	Atualizar requisitos da aplicação	Possibilitado por meio de uma nova conexão
	Especificar requisitos antecipadamente	Possibilitado ao permitir mais de uma conexão ao mesmo tempo por cliente
	Definir requisitos baseados em tempo	Derivado a partir das duas funcionalidades anteriores
Jarschel et al (2013)	Uso de DPI	Uso do campo de dados explicitamente
	Abordagem tem um custo computacional considerável	Custo computacional insignificante

QoE do observada pelo usuário. Este ganho é observado a partir da comparação com métodos tradicionais, como DPI, que gasta um tempo considerável realizando a filtragem de pacotes. Entretanto, alguns métodos também apresentam algumas desvantagens no tempo de computação por conta da verificação feita para detectar o tipo de fluxo. Na técnica que foi empregada nesse trabalho, foi feita uma tentativa de contornar a sobrecarga nos elementos da rede a partir do preestabelecimento de perfis de fluxo. Tirando proveito de uma forma simplificada para o estabelecimento de tais perfis, apenas um caractere foi utilizado na indicação da qualidade requisitada pela aplicação. Dessa forma, se torna trivial a verificação de quais aplicações necessitam de servidores com alta QoS: a aplicação envia o valor 0 quando necessita de alta QoS e o valor 1 ou 2 caso contrário. Assim, é possível que o controlador faça distinção entre fluxos de dados, identificando os que demandam uma largura de banda maior dos que demandam uma largura de banda menor, podendo reencaminhar os pacotes da conexão para o servidor mais adequado.

6 Conclusão e trabalhos futuros

No presente trabalho foi realizado um estudo sobre o paradigma de Redes Definidas por Software. De forma resumida, esse paradigma busca solucionar problemas relacionados com a dificuldade existente em propor e implantar novas tecnologias para as redes de computadores atuais. Uma maneira de ampliar a adoção de RDS é através de propostas de APIs *Northbound*. APIs desse tipo objetivam estabelecer a comunicação entre o plano de aplicação e o plano de controle. A falta de padrão para tal API desincentiva a adesão de grandes desenvolvedores a tirar proveito das vantagens oferecidas por RDS durante a criação de novas aplicações. Dentro desse contexto, o objetivo desse trabalho foi mostrar a factibilidade da criação de uma API que explore a característica de deixar a rede ciente da aplicação. Esse objetivo principal foi alcançado ao fornecer uma forma de recebimento de mensagens vindas da aplicação e, a partir delas, o controlador pode modificar a forma como os pacotes são encaminhados na rede de acordo com que é "avisado" pela aplicação. Além disso, foram mostradas na prática, por meio da implementação de aplicações, algumas vantagens que podem ser obtidas por meio dessa característica. Dentre elas, destaca-se a identificação do tipo de requisição de fluxos na rede, fazendo assim com que a rede fique ciente da aplicação.

A utilização da informação do tipo de fluxo foi feita de maneira simples através do redirecionamento dos pacotes para servidores, que foram previamente definidos para cada tipo de fluxo. Entretanto, o arcabouço criado permite a utilização do método elaborado de diversas maneiras nas mais diferentes aplicações. Em trabalhos futuros, espera-se fazer uso dessa informação juntamente com outras informações da rede para atuar de maneira mais proativa. Uma dessas informações adicionais que pode ser obtida facilmente é a estatística de utlização das conexões entre o *switch* e servidores. Uma vez que o controlador saiba qual conexão está mais livre, este pode dizer ao *switch* para reencaminhar os pacotes por tal conexão, podendo balancear melhor a carga em situações de congestionamento da rede. Através do método proposto torna possível também a agregação de mais funções ao controlador, pois outros tipos de requisições oriundas de aplicações clientes poderiam ser

passadas ao mesmo utilizando o campo de dados de um pacote TCP. O controlador por sua vez poderia dizer ao *switch* que fornecesse alguma informação à aplicação por meio de uma resposta direta utilizando um outro pacote TCP. Reencaminhamentos de transmissão semelhante ao reencaminhamento demonstrado também podem ser feitos utilizando-se de informações diferentes da informação sobre qualidade utilizada.

Foram apresentados experimentos que garantem que a abordagem proposta, no qual o *switch* faz a alteração nos endereços de origem e destino dos pacotes, não gera sobrecarga no tempo de transmisão de maneira significante a impactar negativamente na qualidade de serviço percebida pelo usuário. Em um dos casos pode-se afirmar ainda que o controlador projetado em comparação com o básico não se diferenciou. Após esses testes foi possível elucidar a real contribuição deste trabalho para as Redes Definidas por *Software* Orientadas à Aplicação. Nos trabalhos futuros, mais testes serão propostos com o objetivo de diversificar os cenários de uso para averiguar se a abordagem se mantém interessante. A reprodutibilidade dos experimentos deste trabalho é assegurada pela acessibilidade e fácil configuração do ambiente de testes (Mininet) em conjunto da disponibilização, no sistema de monografias[13], dos códigos elaborados.

[13]http://www.monografias.ice.ufjf.br/tcc-web/

Bibliografia

Chen, L.; Qiu, M.; Dai, W. ; Jiang, N. Supporting high-quality video streaming with sdn-based cdns. **The Journal of Supercomputing**, p. 1–15, 2016.

Costa, L. C.; Vieira, A. B.; de Britto, E.; Silva, D. F.; Gomes, G.; Correia, L. H. ; Vieira, L. F. Avaliaç ao de desempenho de planos de dados openflow. **34o. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2016**.

Feamster, N.; Rexford, J. ; Zegura, E. The road to sdn: an intellectual history of programmable networks. **ACM SIGCOMM Computer Communication Review**, v.44, n.2, p. 87–98, 2014.

Georgopoulos, P.; Elkhatib, Y.; Broadbent, M.; Mu, M. ; Race, N. **Towards network-wide qoe fairness using openflow-assisted adaptive video streaming**. In: Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking, p. 15–20. ACM, 2013.

Georgopoulos, P.; Broadbent, M.; Plattner, B. ; Race, N. **Cache as a service: Leveraging sdn to efficiently and transparently support video-on-demand on the last mile**. In: 2014 23rd International Conference on Computer Communication and Networks (ICCCN), p. 1–9. IEEE, 2014.

Guedes, D.; Vieira, L.; Vieira, M.; Rodrigues, H. ; Nunes, R. V. Redes definidas por software: uma abordagem sistêmica para o desenvolvimento de pesquisas em redes de computadores. **Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2012**, v.30, n.4, p. 160–210, 2012.

Handigol, N.; Heller, B.; Jeyakumar, V.; Lantz, B. ; McKeown, N. **Reproducible network experiments using container-based emulation**. In: Proceedings of the 8th international conference on Emerging networking experiments and technologies, p. 253–264. ACM, 2012.

Hue, C.; Chen, Y.-J. ; Wang, L.-C. **Traffic-aware networking for video streaming service using sdn**. In: 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC), p. 1–5. IEEE, 2015.

Humernbrum, T.; Glinka, F. ; Gorlatch, S. A northbound api for qos management in real-time interactive applications on software-defined networks. **Journal of Communications**, v.9, n.8, 2014.

Jarschel, M.; Wamser, F.; Hohn, T.; Zinner, T. ; Tran-Gia, P. **Sdn-based application-aware networking on the example of youtube video streaming**. In: 2013 Second European Workshop on Software Defined Networks, p. 87–92. IEEE, 2013.

Jarschel, M.; Zinner, T.; Hoßfeld, T.; Tran-Gia, P. ; Kellerer, W. Interfaces, attributes, and use cases: A compass for sdn. **IEEE Communications Magazine**, v.52, n.6, p. 210–217, 2014.

Kaur, K.; Singh, J. ; Ghumman, N. S. **Mininet as software defined networking testing platform**. In: International Conference on Communication, Computing & Systems (ICCCS), p. 139–42, 2014.

Kreutz, D.; Ramos, F. M.; Verissimo, P. E.; Rothenberg, C. E.; Azodolmolky, S. ; Uhlig, S. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v.103, n.1, p. 14–76, 2015.

Lantz, B.; Heller, B. ; McKeown, N. **A network in a laptop: rapid prototyping for software-defined networks**. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p. 19. ACM, 2010.

Li, Y.; Li, J. **Multiclassifier: A combination of dpi and ml for application-layer classification in sdn**. In: Systems and Informatics (ICSAI), 2014 2nd International Conference on, p. 682–686. IEEE, 2014.

McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S. ; Turner, J. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, v.38, n.2, p. 69–74, 2008.

Nam, H.; Kim, K.-H.; Kim, J. Y. ; Schulzrinne, H. **Towards qoe-aware video streaming using sdn**. In: 2014 IEEE Global Communications Conference, p. 1317–1322. IEEE, 2014.

Brief, O. S. Openflow^{TM}-enabled mobile and wireless networks. **white paper**, 2013.

Pham, M.; Hoang, D. B. **Sdn applications-the intent-based northbound interface realisation for extended applications**. In: NetSoft Conference and Workshops (NetSoft), 2016 IEEE, p. 372–377. IEEE, 2016.

Postel, J. Transmission control protocol. 1981.

Quinlan, J. J.; Zahran, A. H.; Ramakrishnan, K. ; Sreenan, C. J. **Delivery of adaptive bit rate video: balancing fairness, efficiency and quality**. In: The 21st IEEE International Workshop on Local and Metropolitan Area Networks, p. 1–6. IEEE, 2015.

Ramos, F. M.; Kreutz, D. ; Verissimo, P. Software-defined networks: On the road to the softwarization of networking. **Cutter IT journal**, 2015.

Rezende, P. H.; Faina, L. F.; Camargos, L. ; Pasquini, R. Roteamento multicaminhos em redes definidas por software. **Simpósio Brasileiro de Redes de Computadores**, 2016.

Zinner, T.; Jarschel, M.; Blenk, A.; Wamser, F. ; Kellerer, W. **Dynamic application-aware resource management using software-defined networking: Implementation prospects and challenges**. In: 2014 IEEE Network Operations and Management Symposium (NOMS), p. 1–6. IEEE, 2014.