Estimando métricas de serviço através de In-band Network Telemetry

Leandro C. de Almeida[1,2], **Fábio L. Verdi**[2], **Rafael Pasquini**[3]

[1]Unidade Acadêmica de Informática – Instituto Federal da Paraíba (IFPB)
João Pessoa – PB – Brasil

[2]Departamento de Computação (Dcomp) – Universidade Federal de São Carlos (UFSCar)
Sorocaba – SP – Brasil.

[3]Faculdade de Computação (FACOM) – Universidade Federal de Uberlândia (UFU)
Uberlância, MG – Brasil

leandro.almeida@ifpb.edu.br, verdi@ufscar.br, rafael.pasquini@ufu.br

Abstract. *Recently, new approaches of fine telemetry in the network, through In-band Network Telemetry in programmable devices, have delivered new and accurate information about the status of the network. In this context, this work presents indications that it is possible to use fine metrics of network telemetry in a programmable data plan, in conjunction with machine learning methods, to estimate quality of service metrics. A minimalist proof of concept was carried out and preliminary results indicate that, with the aid of machine learning algorithms, it is possible to estimate metrics for a video service from fine metrics related to the buffers of the switches.*

Resumo. *Recentemente, novas abordagens de telemetria fina na rede, através de In-band network telemetry em equipamentos programáveis, têm entregue novas e precisas informações sobre o estado da rede. Neste contexto, este trabalho apresenta indícios de que é possível utilizar métricas finas de telemetria de rede em plano de dados programável, em conjunto com métodos de aprendizado de máquina, para estimar métricas de qualidade de serviço. Um prova de conceito minimalista foi realizada e resultados preliminares indicam que, com o auxílio de algoritmos de aprendizado de máquina, é possível estimar métricas de um serviço de vídeo a partir de métricas finas relacionadas aos buffers dos switches.*

1. Introdução

Tradicionalmente, o monitoramento e o gerenciamento das redes de computadores é baseado na análise de um conjunto de métricas de desempenho, tais como: o consumo de CPU de um determinado equipamento, a quantidade de pacotes enviados por uma interface de rede, ou ainda o atraso em uma determinada conexão. Essas métricas representam o estado da rede, e são afetadas por um conjunto de outros fatores. Por exemplo, espera-se que exista uma correlação positiva entre o número de conexões (fator) em um servidor *web* e o seu consumo de memória (métrica afetada).

No contexto de redes, essa relação de causa e efeito não é novidade, sendo amplamente utilizada por protocolos de transporte para inferir congestionamentos. O protocolo TCP (*Transmission Control Protocol*), por exemplo, utiliza o não recebimento de

confirmações para inferir congestionamentos na rede e ajustar a sua taxa de envio. Já o protocolo BBR (*Bottleneck Bandwidth and Round-trip*) utiliza o RTT (*Round-trip time*) para inferir congestionamentos [Cardwell et al. 2016]. Acontece que, tanto no caso do TCP quanto no BBR, não é possível conhecer com precisão em que ponto da rede o congestionamento acontece. Além disso, as informações obtidas representam um estado passado da rede, ou seja, o congestionamento inferido aconteceu em um instante anterior do instante detectado, de modo que as alternativas são reativas e não pró-ativas.

Conjectura-se que os serviços que executam sob uma rede, também poderiam utilizar métricas para avaliar o desempenho e a qualidade de uma aplicação. Em um serviço de transmissão de vídeo, por exemplo, poderia-se utilizar a quantidade de quadros/segundo (FPS - *Frames per second*) recebidos por um cliente para avaliar se a reprodução do vídeo está sendo executada em uma qualidade satisfatória. Neste caso, as métricas de qualidade de um serviço (*QoS - Quality of Service*) poderiam refletir o estado atual da rede na qual o serviço está sendo executado [Stadler et al. 2017]. Entretanto, obter métricas de serviço em tempo real não é algo trivial para a maioria dos provedores de rede [Calasans 2020].

Em switches, *buffers* são uma importante fonte de informação, pois podem indicar o nível de congestionamento da rede [Kim et al. 2018]. Além disso, o nível de ocupação do *buffer* pode afetar outras métricas usadas pelos sistemas de monitoramento, como RTT, atraso e *jitter*. Coletar dados de *buffers* em tempo real, no entanto, sempre foi uma tarefa muito cara, devido à necessidade de bits adicionais nos cabeçalhos e o aumento do consumo de energia dos dispositivos [Arslan and McKeown 2019].

Esse cenário mudou nos dispositivos de rede modernos, que com um aumento insignificante de potência ou perda de capacidade, é possível obter informações sobre os *buffers* em tempo real [Arslan and McKeown 2019]. Além disso, com um plano de dados programável e telemetria de rede, é possível observar dados mais precisos da rede.

Inserir inteligência nesse contexto é um passo natural, visto que pesquisas no campo de redes de computadores têm avançado no uso de algoritmos de aprendizado de máquina (*ML - Machine Learning*) [Boutaba et al. 2018]. Na literatura, alguns trabalhos têm feito uso de ML em conjunto com plano de dados programável [Xiong and Zilberman 2019], entretanto, não se tem conhecimento de um trabalho que faça uso da telemetria em plano de dados programável para estimar métricas de serviço.

Neste contexto, este trabalho apresenta indícios de que é possível utilizar métricas finas (relacionadas aos *buffers*) de telemetria de rede em plano de dados programável, em conjunto com métodos de aprendizado de máquina, para estimar métricas de QoS. Uma prova de conceito (*PoC - Proof of Concept*) minimalista foi realizada com o objetivo de avaliar a hipótese de que existe uma correlação entre dados de telemetria de rede e métricas de um serviço de vídeo. Resultados preliminares indicam que, com o auxílio de algoritmos de ML, é possível estimar métricas de um serviço de vídeo a partir de métricas finas relacionadas aos *buffers* dos switches. Essa hipótese poderia auxiliar a compreenssão da qualidade de um serviço executado sob redes de *datacenter* ou ainda em porções de redes de ISPs, nas quais o hardware programável poderia ser utilizado.

As principais contribuições deste trabalho são: 1) demonstração através de uma PoC minimalista de que é possível estimar métricas de um serviço de vídeo a partir de métricas finas relacionadas aos *buffers* dos switches, utilizando algoritmos de ML; 2)

indícios estatísticos de que o erro absoluto médio normalizado (*NMAE - Normalized Mean Absolute Error*) é menor em relação aos trabalhos relacionados, que não utilizam métricas finas de rede; 3) toda a infraestrutura da PoC, incluíndo o *dataset* utilizado para treinamento e avaliação, disponibilizada em um ambiente *Vagrant* sob a perspectiva de infraestrutura como código (*IaC - Infrastructure as Code*), para fins de replicação e comparação.

Este trabalho está organizado conforme descrito a seguir. A Seção 2 apresenta o problema de pesquisa, detalhando como podemos estimar métricas de serviço a partir de métricas finas de rede. A Seção 3 apresenta resumidamente a linguagem P4 e a especificação INT (*In-band Network Telemetry*), os quais são a base para a produção e coleta de métricas finas de rede. A Seção 4 apresenta os conceitos relacionados aos métodos de ML. A PoC é descrita na Seção 5, de modo que a infraestrutura necessária para a execução do experimento e detalhes de configuração estão descritos. Na Seção 6 estão detalhados os principais resultados da PoC. Por fim, na Seção 7, têm-se as conclusões e as propostas para trabalhos futuros.

2. Descrição do problema

Em um ambiente distribuído como a Internet, coletar métricas de qualidade de um serviço fim-a-fim não é algo trivial, visto que existe uma complexidade intríseca na arquitetura da rede. Além disso, em um ambiente multi-domínio, o acesso as métricas ainda é um desafio. Nesse contexto, este trabalho investiga a viabilidade de se estimar métricas de QoS de vídeo adaptativo (*ABS - Adaptive Bitrate Streaming*), a partir de um conjunto de métricas finas de rede.

Essa estimativa pode ser útil no contexto de provedores de serviço de rede, os quais, poderiam ajustar a infraestrutura de acordo com as estimativas preditas pelos algoritmos de ML. Em nosso sistema de estudo, analisamos como métricas finas de rede (denominadas por conjunto de dados *X*) se correlacionam com métricas de QoS (denominadas por conjunto de dados *Y*). De um modo geral, as métricas finas de rede *X*, são dados relativos aos *buffers* das portas dos nós na rede. Já as métricas de QoS *Y* são dados relativos a execução de componentes de vídeo e áudio em um cliente de vídeo.

Assim como no trabalho [Stadler et al. 2017], consideramos um relógio global (*global clock*), que pode ser lido por todos os dispositivos da rede, de modo que todos os relógios estejam sincronizados corretamente. Além disso, a evolução das métricas *X* e *Y* é tratada como uma série temporal $\{X_t\}$, $\{ Y_t \}$ e $\{ (X_t, \; Y_t \;) \}$, na qual cada instante de tempo *t* representa o estado da rede e do serviço de vídeo. O problema de estimar métricas de serviço Y_t no tempo *t* de um cliente, baseado no conhecimento das métricas finas de telemetria de rede X_t, pode ser modelado como $M : X_t \rightarrow \hat{Y}_t$, onde $\hat{Y}_t$ é uma função aproximadora da função Y_t, para um dado X_t. Este é um problema de regressão que pode ser resolvido com aprendizado supervisionado [James et al. 2013].

A hipótese utilizada neste trabalho é de que variações nas métricas de serviço (por exemplo, quadros por segundo) possuem correlação positiva com métricas finas de rede. Neste sentido, conjecturamos que com o auxílio de algoritmos de ML existem indícios de que a hipótese é verdadeira.

3. Programabilidade de redes e linguagem P4

Com o surgimento das redes definidas por software (*SDN - Software Defined Networking*) [McKeown et al. 2008], vieram os primeiros esforços na direção da programabilidade da rede [Hauser et al. 2021]. A separação dos planos de dados e controle, permitiu uma maior flexibilidade e abriu novos horizontes de pesquisa nas redes de computadores.

Neste contexto, o protocolo *openflow* foi pioneiro em fornecer uma camada de abstração da rede física para o elemento de controle, permitindo assim que o plano de dados da rede fosse configurado ou manipulado através de programação via software, entretanto ainda não oferecia flexibilidade suficiente para se adicionar novos cabeçalhos e definir novas ações após um *flow matching* [Garcia et al. 2018].

A linguagem P4[1] [Bosshart et al. 2014] surgiu em 2014 como uma alternativa e evolução natural do paradigma de programabilidade de redes, trazendo a possibilidade de se programar o plano de dados de um elemento de rede ou em uma placa de rede inteligente (*SmartNics*).

Em um elemento de rede, o plano de dados é responsável pela função de processamento de pacotes orientado pela lógica definida pelo plano de controle. A programabilidade do plano de dados significa que algoritmos podem ser definidos para manipular pacotes no nível mais próximo do hardware, sem precisar recorrer ao plano de controle. A linguagem P4 se enquadra neste contexto, permitindo que programadores definam algoritmos para o processamento de pacotes no plano de dados.

O código compilado em linguagem P4 pode executar em diversas arquiteturas, por exemplo: PISA (*Protocol-Independent Switching Architecture*), PSA (*Portable Switch Architecture*), *V1model*, *Simple Sume Architecture* e TNA (*Tofino Native Architecture*). Neste trabalho foi escolhida a arquitetura *V1model*, pois é a única completamente suportada pelo software switch BMv2 (*Behavioral Model version 2*) utilizado na PoC.

De acordo com a Figura 1, a arquitetura *V1model* é composta pelos componentes programáveis: *Parser*, *Ingress/Egress match-action pipelines* e *deparser*. O *traffic manager* não é um componente programável via código P4 na arquitetura *v1model*.

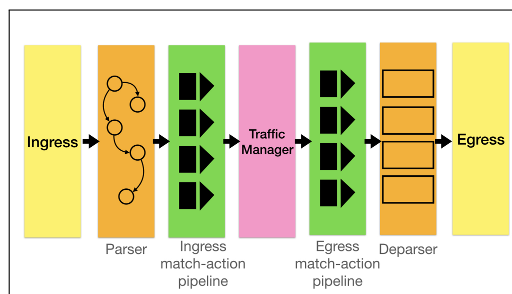


Figura 1. Arquitetura *V1model*.

3.1. *In-band Network telemetry*

Graças aos avanços recentes em hardware programável e a linguagem P4, os dispositivos de rede podem informar o estado da rede sem intervenção do plano de controle

[1] *Programming Protocol-independent Packet Processors.*

[Arslan and McKeown 2019]. Nesse caso, os pacotes contêm campos de cabeçalho que são instruções de telemetria, os quais podem ser usados para coletar e gravar dados de rede com maior granularidade. As instruções de telementria estão descritas na especificação de plano de dados INT [P4 2020], a qual define três modos de operação: INT-XD (*eXport Data*), INT-MX (*eMbed Instructions*) e INT-MD (*eMbed Data*).

No modo INT-XD, cada dispositivo exporta os metadados usando suas instruções INT configuradas nas tabelas de fluxo diretamente do plano de dados para o sistema de monitoramento. Neste modo, nenhuma modificação nos pacotes do tráfego dos clientes é realizada.

No modo INT-MX, o nó de origem cria instruções INT no cabeçalho do pacote, de modo que em cada nó de trânsito as intruções INT são lidas e os respectivos metadados são transmitidos diretamente para o sistema de monitoramento. Neste modo, pequenas modificações são realizadas nos pacotes do tráfego dos clientes, visto que apenas instruções de telemetria são inseridas nos cabeçalho dos pacotes.

No modo INT-MD, instruções INT e metadados são inseridos nos pacotes a cada salto na rede. Este é o modo de operação padrão definido pela especificação INT, e utilizado neste trabalho. Os metadados utilizados na PoC foram os seguintes:

- *Node ID*: identificador único do dispositivo na rede P4.
- *Ingress Port*: número da porta em que o pacote entrou no dispositivo P4.
- *Egress Spec*[2]: número da porta em que o pacote irá sair do dispositivo P4.
- *Egress Port*: número da porta em que o pacote saiu do dispositivo P4.
- *Ingress Global Timestamp*: o *timestamp*, em microsegundos, de quando o pacote entrou no bloco *ingress*.
- *Egress Global Timestamp*: o *timestamp*, em microsegundos, de quando o pacote iniciou o processamento no bloco *egress*.
- *Enq Timestamp*: o *timestamp*, em microsegundos, de quando o pacote foi inicialmente colocado na fila para processamento.
- *Enq Qdepth*: a profundidade da fila quando o pacote foi enfileirado, em unidades de número de pacotes.
- *Deq Timedelta*: o tempo, em microsegundos, em que o pacote permaneceu na fila.
- *Deq Qdepth*: a profundidade da fila quando o pacote foi desenfileirado, em unidades de número de pacotes.

4. Aprendizado de máquina

Para realizar comparações com os trabalhos relacionados atuais, [Stadler et al. 2017] e [Calasans 2020], consideramos o método floresta aleatória (*Random Forest*) uma vez que tais trabalhos usaram este método. Para avaliar o desempenho do método, foram utilizadas as seguintes métricas: o Erro Absoluto Médio Normalizado (NMAE) e o tempo de treinamento do modelo. A validação cruzada foi utilizada para treinar e avaliar os modelos, buscando minimizar problemas de sobreajuste (*overfitting*) nos dados.

A floresta aleatória é um método que estende as árvores de regressão, utilizando uma estratégia de combinação de diversas árvores. A estimativa final é calculada a partir

[2]Metadado escrito no *ingress*, para indicar qual a porta de saída esperada. Já o *egress port* é o metadado lido no *egress*, para indicar qual a porta de saída já definida.

da média das estimativas resultantes de cada árvore. Cada árvore é construída utilizando uma fração dos atributos X de entrada de maneira aleatória, alterando o formato de cada árvore [Stadler et al. 2017]. A árvore de regressão é um método baseado na estratégia de dividir para conquistar, buscando resolver um problema complexo pela decomposição em subproblemas menores. Neste caso, as soluções dos subproblemas são combinadas em um formato de árvore, produzindo a solução do problema original [Faceli et al. 2011].

O objetivo do algoritmo da árvore de regressão é minimizar a soma dos quadrados residuais (*RSS - Residual Sum Square*), descrito na Equação 1, onde $\hat{y}_{R_j} = \sum_{i \in R_j} \frac{Y_i}{|R_j|}$.

$$\sum_{j=1}^{J} \sum_{i \in R_k} (y_i - \hat{y}_{R_j})^2 \quad (1)$$

Nesse caso, o espaço *X* de *n* características é dividido em *J* regiões distintas e não sobrepostas, R_1, R_2,...,R_j. Para cada observação que cair numa dada região R_j, uma predição é realizada, a qual consiste na média de valores de resposta para as observações de treino em R_j. Além disso, $\hat{y}_{R_j}$ é a resposta média para as $\mid R_j \mid$ observações da j-ésima região [Calasans 2020].

4.1. Métricas de avaliação

Em linhas gerais, não existe um algoritmo melhor que os demais em todas as situações. Por este motivo, faz-se necessário utilizar métricas de avaliação de desempenho para se obter conclusões a partir de seus resultados. Em problemas de regressão, a avaliação de um algoritmo de ML é normalmente realizada por meio da análise do regressor gerado por ele na estimação de novos objetos, não apresentados previamente em seu treinamento [Faceli et al. 2011].

O NMAE representa a média normalizada dos erros absolutos das predições efetuadas, descrito na Equação 2, onde m é o tamanho do conjunto de testes; y_i corresponde à amostra *i* do conjunto de teste; $\hat{y}_i$ refere-se ao valor estimado para a amostra *i*; e $\bar{y}$ é a média das respostas das amostras do conjunto de teste.

$$NMAE(y, \hat{y}) = \frac{1}{\bar{y}} \left(\frac{1}{m} \sum_{i=0}^{m-1} |y_i - \hat{y}_i| \right) \quad (2)$$

O tempo de treinamento é a medida de tempo em segundos que durou o processo de treinamento do modelo no conjunto de treino, conforme a Equação 3.

$$t_{treino} = t_{final} - t_{inicial} \quad (3)$$

4.2. Validação cruzada

Sobreajustamento (*overfitting*), é o problema no qual o regressor obtem um bom resultado na estimativa da função objetivo $\hat{Y}$, uma vez que ele se ajustou quase perfeitamente aos dados de entrada X [Faceli et al. 2011]. Este problema acontece quando o regressor foi treinado com todos os dados de entrada, ou com uma partição considerável dos dados.

Para minimizar este problema, utilizou-se a estratégia de validação cruzada com k partes (*k-fold cross-validation*). Esta estratégia permite a divisão dos dados em k partições, as quais são utilizadas para treinamento e validação do regressor de forma

independente. Conforme a Figura 2, pode-se observar que em cada k partição existem dois subconjuntos, treinamento (80%) e teste (20%), que não se repetem. Dessa forma, o modelo é treinado e avaliado k vezes.

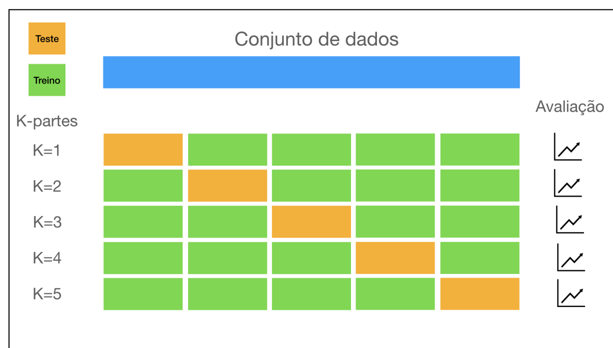


Figura 2. Validação cruzada com K=5 partições.

5. Prova de conceito

A PoC deste trabalho foi realizada a partir de experimentos que permitiram avaliar a hipótese definida na Seção 2. Um ambiente baseado em virtualização foi construído, sendo executado em um servidor físico de marca Dell modelo EMC PowerEdge R720 com 2 processadores Intel Xeon® E5-2630 v2 2.60GHz, 6 cores por socket (24 vCPUs), 48GB de memória RAM, 2TB de HDD e sistema operacional Ubuntu 18.04.5 LTS. O hipervisor utilizado foi o Virtualbox (6.1.8), em conjunto com o Vagrant (2.2.13) e Ansible (2.9.15), para provisionamento da infraestrutura. Esta PoC foi desenvolvida sob a perspectiva de IaC, disponível para fins de auditoria e replicação em um repositório público[3].

5.1. Descrição dos componentes

A topologia, descrita na Tabela 1 e apresentada na Figura 3, é composta por 9 máquinas virtuais tendo todas as suas conexões providas por redes do tipo *internal network* do Virtualbox com taxa de transmissão de 1Gbps.

Nome	OS	vCPUs	Memória	Finalidade
dashServer	Ubuntu 16.04.7 LTS	1	4GB	Servidor de streaming de vídeo
clientVlc	Ubuntu 20.04.1 LTS	16	8GB	Cliente de vídeo
sinkServer	Ubuntu 20.04.2 LTS	2	1GB	Coletor de métricas de telemetria
loadGen1	Ubuntu 20.04.1 LTS	8	8GB	Gerador de Carga
loadGen2	Ubuntu 20.04.1 LTS	8	8GB	Gerador de Carga
loadGen3	Ubuntu 20.04.1 LTS	8	8GB	Gerador de Carga
bmv2_1	Ubuntu 20.04.1 LTS	4	1GB	Switch com suporte a P4
bmv2_2	Ubuntu 20.04.1 LTS	4	1GB	Switch com suporte a P4
bmv2_3	Ubuntu 20.04.1 LTS	4	1GB	Switch com suporte a P4

Tabela 1. Detalhamento das máquinas virtuais.

O dashServer é o componente responsável por disponibilizar streaming de vídeo no padrão MPEG-DASH (*Dynamic Adaptative Streaming over HTTP*) [ISO 2014] para o cliente e os três geradores de carga. O MPEG-DASH é o atual padrão tecnológico utilizado por empresas de vídeo, tais como Netflix® e Google® [Lederer 2015]. No

[3]https://github.com/leandrocalmeida/PoCSBRC2021.

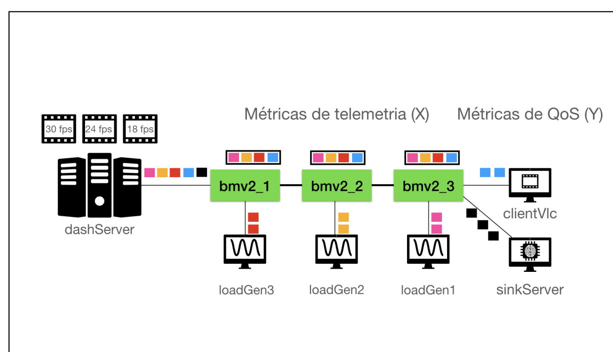


Figura 3. Cenário do experimento.

experimento, foram disponibilizados dois streamings de vídeo: uma transmissão de uma partida de futebol para acesso do cliente; e uma lista de reprodução contendo os dez vídeos mais acessados no Youtube® para acesso dos geradores de carga. Foram instaladas as aplicações Apache versão 2 como servidor web; FFmpeg (2.8.17) para codificação dos vídeos; e MP4box (0.5.2) para criação dos arquivos de manifesto do MPEG-DASH.

O clientVlc é o componente responsável por consumir o streaming de vídeo da partida de futebol, no qual executou o player de vídeo VLC (3.0.8), com modificações [Fernandes 2018] para coleta de métricas de serviço.

Os componentes bmv2_{1,2,3}, são switches BMv2, foram programados para anexar os metadados descritos na Seção 3.1 em todos os pacotes com instruções INT, conforme o trecho de Código 1 utilizado no *egress pipeline*.

```
1   action add_swtrace(switchID_v swid) {
2           hdr.nodeCount.count = hdr.nodeCount.count + 1; //incrementa o contador de nós
3           hdr.INT.push_front(1); //ajusta a posição do headerStack
4           hdr.INT[0].setValid(); //torna o cabeçalho válido
5
6           //extração dos metadados
7           hdr.INT[0].swid = swid;
8           hdr.INT[0].ingress_port = (ingress_port_v)standard_metadata.ingress_port;
9           hdr.INT[0].ingress_global_timestamp =
            ↪  (ingress_global_timestamp_v)standard_metadata.ingress_global_timestamp;
10          hdr.INT[0].egress_port = (egress_port_v)standard_metadata.egress_port;
11          hdr.INT[0].egress_spec = (egressSpec_v)standard_metadata.egress_spec;
12          hdr.INT[0].egress_global_timestamp =
            ↪  (egress_global_timestamp_v)standard_metadata.egress_global_timestamp;
13          hdr.INT[0].enq_timestamp = (enq_timestamp_v)standard_metadata.enq_timestamp;
14          hdr.INT[0].enq_qdepth = (enq_qdepth_v)standard_metadata.enq_qdepth;
15          hdr.INT[0].deq_timedelta = (deq_timedelta_v)standard_metadata.deq_timedelta;
16          hdr.INT[0].deq_qdepth = (deq_qdepth_v)standard_metadata.deq_qdepth;
17
18          //incrementa o tamanho do pacote IP
19          hdr.ipv4.totalLen = hdr.ipv4.totalLen + 32;
20      }
```

Código 1: Ação que insere os metadados INT nos pacotes.

O sinkServer é o componente responsável por coletar o tráfego INT e armazenar no formato suportado pelos métodos inteligentes. Códigos escritos em linguagem Python foram utilizados para executar a funcionalidade de coleta e armazenamento.

Os geradores de carga (loadGen{1/2/3}) são componentes responsáveis pela

simulação de clientes consumindo o streaming dos dez vídeos mais acessados no Youtube®. Ambos executam obedecendo uma função sinusoidal, descrita na Equação 4, onde: A representa a amplitude; F a frequência; e λ é a fase em radianos.

$$f(y) = A \sin(F + \lambda) \tag{4}$$

5.2. Descrição do experimento

O experimento teve duração aproximada de 2 horas (7200 segundos), no qual o dashServer disponibilizou todos os recursos (vídeo/áudio) com características distintas, conforme a Tabela 2, para que o cliente pudesse fazer a transição entre todos os tipos disponibilizados de acordo com a carga na rede.

Tipo	Resolução	FPS	GOP[4]	Kbps	Buffer	Codec
vídeo	426x240	18	72	280	140	h264
vídeo	854x480	24	96	980	490	h264
vídeo	1280x720	30	120	2080	1040	h264
áudio	-	-	-	128	-	AAC
áudio	-	-	-	64	-	AAC

Tabela 2. Parâmetros utilizados nos recursos disponibilizados pelo dashServer.

A cada segundo foram coletadas métricas de qualidade de serviço de vídeo no componente clientVlc. Em paralelo, a cada segundo pacotes com instruções INT foram enviados do componente dashServer para o componente sinkServer. Neste caso, nos componentes bmv2_{1,2,3} foram anexados os metadados descritos na Seção 3.1 em cada pacote, conforme a Figura 4. A cada salto, o pacote INT aumenta em **32 bytes** (metadados) em relação ao seu tamanho original. Considerando que a solução aqui apresentada se aplica inicialmente em redes de *datacenters*, onde o número máximo de saltos é geralmente não maior que cinco [Li et al. 2019], não haverá fragmentação de pacotes.

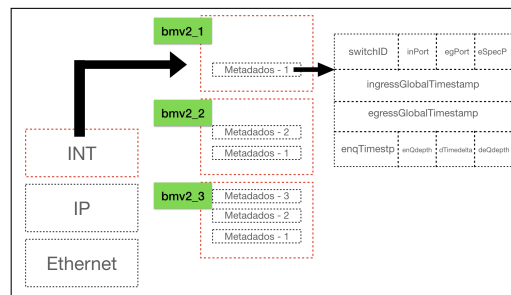


Figura 4. Metadados INT utilizados no experimento.

Ao chegar no componente sinkServer, os metadados foram extraídos e armazenados no formato adequado para os métodos de ML. Além disso, os geradores de carga executaram a lista de reprodução com os parâmetros definidos na Tabela 3.

Após a realização do experimento, os dados (métricas de QoS e INT) foram integrados em uma matriz M_{mxn}, onde: m representa o número de amostras (série temporal a cada segundo); e n representa a quantidade de atributos utilizados para a regressão.

[4] *Group of Pictures.*

Componente	A	F	lambda
loadGen1	4	15	5
loadGen2	4	15	5
loadGen3	4	15	5

Tabela 3. Parâmetros utilizados nos geradores de carga.

Antes de submeter os dados para o método de ML, foi realizado uma etapa de pré-processamento, na qual objetivou-se melhorar a qualidade e a representação dos dados. O pré-processamento foi realizado seguindo os passos descritos abaixo:

Tratamento de dados incompletos/faltantes: dados incompletos/faltantes (*NaN - Not a Number*) podem acarretar em problemas na execução dos métodos [Kotsiantis et al. 2006]. Por este motivo, inicialmente, realizou-se um processo de busca e tratamento desses dados, através da função `removeDadosFaltantes` . Neste caso, as amostras com valores do tipo NaN foram removidas.

Remoção de atributos com o mesmo valor: atributos com valor único não contêm informações que ajudem a distinguir os objetos, portanto são considerados irrelevantes [Faceli et al. 2011]. Por este motivo, atributos com valor único foram removidos com a função `removeAtributoscomMesmoValor` .

Normalização de atributos: atributos que possuem escalas muito diferentes podem acarretar em problemas nos métodos de aprendizado de máquina [Faceli et al. 2011]. Por este motivo, realizou-se o processo de normalização *z-score* [Kotsiantis et al. 2006] dos atributos com a função `StandardScaler` do pacote Scikit-learn. Neste caso, a média de cada atributo ficou igual a zero e desvio padrão igual a um.

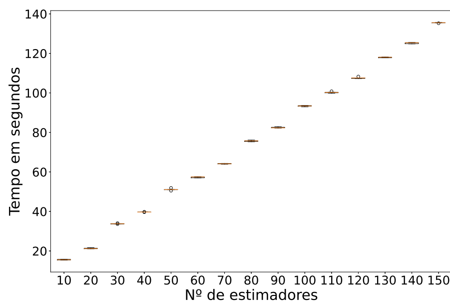
Após a etapa de pré-processamento, os dados foram divididos em partições (80% para treino e 20% para teste) utilizando a função `RepeatedKFold` do pacote Scikit-learn do python. Cada partição foi submetida ao método de floresta aleatória através da função `RandomForestRegressor` do pacote Scikit-learn, variando (10 - 160) o parâmetro número de estimadores. Além disso, realizou-se uma busca em grade (com hyperparâmetros), com objetivo de encontrar os melhores modelos. Ao total, foram realizadas 1875 (15 x 125) análises de configurações para cada métrica (aúdio e vídeo).

6. Resultados

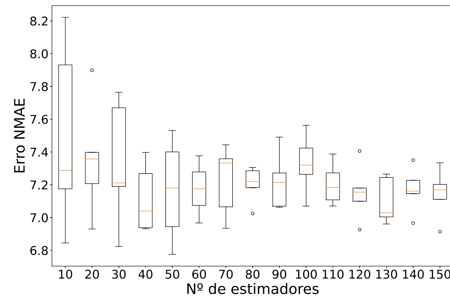
Os resultados foram obtidos a partir das estimativas de duas métricas de serviço: **quadros de vídeo tocados por segundo** e **buffers de áudio tocados por segundo**. Para cada valor de estimadores (número de árvores utilizadas), têm-se um diagrama de caixa (*boxplot*), contendo os quartis q_1, q_2 e q_3; os limites inferior (l_i) e superior (l_s); e em alguns casos os valores discrepantes (*outliers*).

A Figura 5 apresenta os resultados para a métrica quadros de vídeo tocados por segundo. Neste caso, é possível observar na Figura 5(a), o tempo de treinamento (com média de 74s), em segundos, do método floresta aleatória no eixo das ordenadas e o número de estimadores no eixo das abscissas, no qual o aumento no número de estimadores implicou no aumento do tempo de treinamento. Conjectura-se que este era o comportamento esperado, visto que o método utilizou mais árvores de regressão (estimadores) para realizar as estimativas.

Na Figura 5(b), é possível observar o erro NMAE (com média de 7.22%), no eixo



(a) Tempo de treinamento

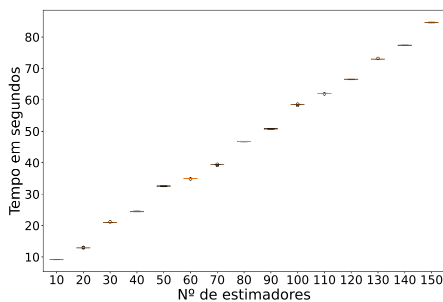


(b) Erro NMAE

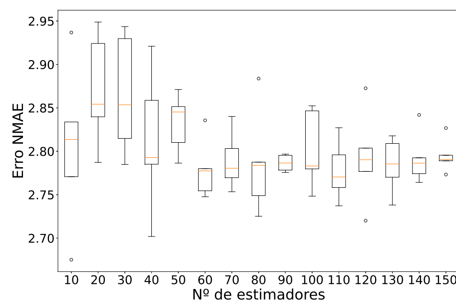
Figura 5. Método floresta aleatória - vídeo quadros/segundo.

das ordenadas e o número de estimadores no eixo das abscissas. O aumento no número de estimadores não apresentou diferenças com significância estatística entre os erros, ou seja, pode-se utilizar menos estimadores reduzindo o tempo de treino. Conjectura-se que isso aconteceu porque no cálculo, conforme descrito na Equação 2, existe a normalização do erro absoluto médio.

A Figura 6 apresenta os resultados para a métrica buffers de áudio tocados por segundo. De forma similar a métrica de vídeo, o tempo de treinamento (com média de 46.26s) teve crescimento proporcional ao aumento no número de estimadores. Além disso, o erro NMAE (com média de 2.80%) também não apresentou diferenças, com significância estatística, em relação ao aumento no número de estimadores. Da mesma forma como a métrica de vídeo, pode-se utilizar um número menor de estimadores, reduzindo o tempo médio de treinamento.



(a) Tempo de treinamento

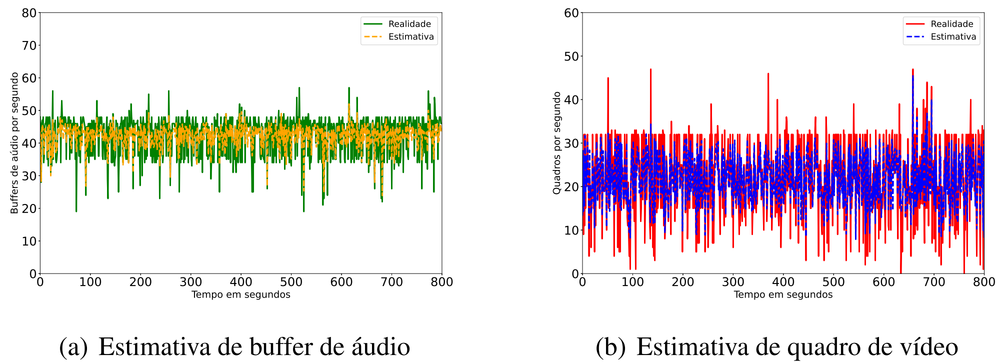


(b) Erro NMAE

Figura 6. Método floresta aleatória - áudio buffers/segundo

Nos gráficos de erro NMAE para ambas as métricas, 5(b) e 6(b), observa-se que o erro foi menor para a métrica de áudio. Conjectura-se que isso aconteceu, devido ao áudio ser menos sensível as oscilações geradas pela carga na rede.

A Figura 7 apresenta as estimativas computadas pelo método floresta aleatória em comparação com os dados reais. Neste caso, a Figura 7(a) representa uma disposição visual das estimativas em relação a realidade para a métrica de áudio. Já a Figura 7(b) representa a mesma disposição visual, mas para a métrica de vídeo.



(a) Estimativa de buffer de áudio

(b) Estimativa de quadro de vídeo

Figura 7. Estimativas vs Realidade.

Visualmente, na Figura 7, é observado que o método inteligente é mais acurado para a métrica de áudio em relação a métrica de vídeo. Este indício ganha força quando observa-se o valor do erro NMAE, o qual obteve 1.97% para áudio e 6.70% vídeo. Em relação aos trabalhos relacionados, um resumo dos resultados estão desctitos na Tabela 4.

Estudo	Tempo de treinamento (áudio)	NMAE (áudio)	Tempo de treinamento (vídeo)	NMAE (vídeo)
Este artigo	46.26s	2.80%	74s	7.22%
[Stadler et al. 2017]	334s	20.6%	281s	9.17%
[Calasans 2020]	0.047s	13%	0.109s	16.5%

Tabela 4. Análise dos resultados em relação aos trabalhos relacionados.

O trabalho relacionado [Stadler et al. 2017] utilizou uma combinação de métricas de um cluster computacional (uso de CPU, memória e disco) em conjunto com métricas de uma rede openflow (número de bytes/pacotes transmitidos e recebidos). Foi obtido um erro NMAE de 9.17% (vídeo), ou seja, utilizou-se mais métricas e obteve-se um erro maior em relação a este trabalho. Além disso, foi utilizado um serviço de vídeo estático, diferente do MPEG-DASH que é dinâmico sob o ponto de vista da carga na rede. Este trabalho conseguiu lidar com essa dinamicidade e obteve um erro menor.

Já no trabalho [Calasans 2020] foram utilizadas métricas de rede openflow (número de bytes/pacotes enviados e recebidos) para estimar métricas de um serviço MPEG-DASH. Os parâmetros de vídeo e áudio foram idênticos ao deste estudo. Apesar de utilizar apenas métricas de rede, obtiveram um erro NMAE de 16.5% (vídeo).

Comparando os valores de erro NMAE em relação aos trabalhos relacionados, é possível observar indícios estatísticos de que utilizar métricas finas de telemetria para estimar métricas de um serviço de vídeo pode apresentar melhores resultados, ou seja, um erro menor e consequentemente uma predição mais acurada. Neste contexto, este trabalho apresentou melhores resultados em relação ao estado da arte atual.

Já acerca dos tempos para treinamento do método, a comparação não pode ser realizada visto que as bases utilizadas nos três estudos possuem tamanhos distintos. Além disso, não se tem informação das características do computador utilizado para realizar o treinamento das bases. Sendo assim, este dado é meramente informativo neste trabalho.

7. Conclusões

Este trabalho apresentou indícios de que é possível utilizar métricas finas de telemetria de rede em plano de dados programável, em conjunto com métodos de aprendizado de máquina, para estimar métricas de QoS de vídeo com taxa de transmissão adaptativa.

Resultados preliminares de uma PoC minimalista indicaram que o erro NMAE, para as métricas de áudio e vídeo, são menores em relação ao estado da arte atualmente. Todo o conjunto de dados utilizado para treinar o modelo floresta aleatória, bem como a infraestrutura da PoC foi disponibilizada sob a perspectiva de infraestrutura como código para fins de replicação e comparação.

Apesar dos avanços apresentados neste trabalho, estudos mais aprofundados ainda precisam ser realizados. Questões importantes que ainda estão em aberto são: 1) uma extração dos metadados em apenas uma parcela dos nós da rede traria resultados satisfatórios em relação ao erro NMAE ou acurária do modelo? 2) Se sim, quais seriam os melhores nós para realizar a extração dos metadados? Mais próximo do serviço, ou mais próximo do cliente? 3) Será que a taxa de extração em um nível de granularidade diferente, traria resultados melhores?

Além disso, experimentos com outros tipos de vídeo (ex.: filmes, desenhos e lives), com uma maior duração de tempo e com outras trilhas de áudio devem ser realizados em trabalhos futuros. Outros métodos de aprendizado de máquina que utilizem características diferentes do floresta aleatória também devem ser estudados.

Referências

Arslan, S. and McKeown, N. (2019). Switches know the exact amount of congestion. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA. Association for Computing Machinery.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Boutaba, R., Salahuddin, M., Limam, N., Ayoubi, S., Shahriar, N., Estrada-Solano, F., and Caicedo Rendon, O. (2018). A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9.

Calasans, Marta e Lacerda, C. (2020). *DASH sobre OpenFlow: estimando métricas de QoS a partir da rede*. PhD thesis, Universidade Federal de Uberlândia, Uberlândia - MG - Brazil.

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. (2016). Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53.

Faceli, K., Lorena, A. C., Gama, J., and Carvalho, A. C. P. d. L. F. d. (2011). *Inteligência artificial: uma abordagem de aprendizado de máquina*. LTC.

Fernandes, J. R. M. (2018). *Construção de um serviço conteinerizado de vídeo sob demanda baseado em DASH para experimentação e coleta de métricas de desempenho*. PhD thesis, UNIVERSIDADE FEDERAL DE UBERLÂNDIA.

Garcia, L. F. U., Villaça, R. S., Ribeiro, M. R. N., Martins, R. F. T., Verdi, F. L., and Marcondes, C. (2018). Introdução à linguagem p4 - teoria e prática. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) - Minicursos*.

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., and Menth, M. (2021). A survey on data plane programming with p4: Fundamentals, advances, and applied research.

ISO (2014). Dynamic adaptive streaming over http (dash)-part 1: Media presentation description and segment formats. *ISO/IEC*, pages 23009–1.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning*. Springer Texts in Statistics. Springer, New York, NY.

Kim, Y., Park, J., Kwon, D., and Lim, H. (2018). Buffer management of virtualized network slices for quality-of-service satisfaction. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, number 18725013 in 1, pages 1–4, Verona, Italy. IEEE.

Kotsiantis, S., Kanellopoulos, D., and Pintelas, P. (2006). Data preprocessing for supervised learning. *International Journal of Computer Science*, 1:111–117.

Lederer, S. (2015). Why youtube & netflix use mpeg-dash in html5. `https://bitmovin.com/status-mpeg-dash-today-youtube-netflix-use-html5-beyond/`. Accessed: 2020-03-24.

Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., and Yu, M. (2019). Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM ’19, page 44–58, New York, NY, USA. Association for Computing Machinery.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

P4 (2020). In-band network telemetry (int) dataplane specification. Technical report, P4 Consortium.

Stadler, R., Pasquini, R., and Fodor, V. (2017). Learning from network device statistics. *J. Netw. Syst. Manag.*, 25(4):672–698.

Xiong, Z. and Zilberman, N. (2019). Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets ’19, page 25–33, New York, NY, USA. Association for Computing Machinery.