Universidade Federal do ABC

CENTRO DE ENGENHARIA, MODELAGEM E CIÊNCIAS SOCIAIS APLICADAS

Análise Comparativa de Algoritmos de Balanceamento de Carga em Redes SDN

Herbert Hurtado Antunes Junior

Marcos Aurelio da Silva Costa

Trabalho de Conclusão de Curso

2026

Herbert Hurtado Antunes Junior

Marcos Aurelio da Silva Costa

TRABALHO DE GRADUAÇÃO EM ENGENHARIA DE INFORMAÇÃO

Análise Comparativa de Algoritmos de Balanceamento de Carga em Redes SDN

Trabalho de Graduação apresentado na disciplina Trabalho de Graduação III do Curso de Engenharia de Informação na Universidade Federal do ABC - UFABC.

Orientador: Claudio José Bordim Jr

Santo André

2026

Resumo

O crescimento exponencial do tráfego de dados em redes modernas impõe desafios crescentes à gestão eficiente de recursos computacionais. As *Software-Defined Networking* (SDN) emergem como paradigma arquitetural que centraliza o controle da rede em controladores programáveis, possibilitando maior flexibilidade e automação. Este trabalho apresenta uma análise comparativa de quatro estratégias de balanceamento de carga em ambiente SDN simulado: Controlador Simples (CS), *Round-Robin* (RR), *Least-Loaded* (LL) e um controlador baseado em *Machine Learning* (ML) utilizando o algoritmo *Random Forest*. A infraestrutura experimental foi implementada no emulador Mininet com protocolo *OpenFlow* (OF) 1.3, sobre uma topologia hierárquica de três camadas composta por 10 *switches* de acesso, 3 *switches core*, 100 *hosts* e 5 servidores, os experimentos foram conduzidos em quatro cenários de tráfego distintos: Carga Normal, *Hotspot*, Tráfego Misto e Carga Sustentada.

Os resultados demonstraram que qualquer estratégia de balanceamento ativo supera significativamente o encaminhamento sem critério. O controlador LL obteve o melhor desempenho geral, embora com vantagem modesta sobre o RR. O controlador baseado em ML, posicionou-se entre o RR e o LL, evidenciando limitações relacionadas ao viés do conjunto de dados de treinamento. Os resultados indicam que algoritmos simples como o RR já capturam grande parte do benefício do balanceamento ativo em redes SDN com tráfego heterogêneo.

Palavras-chave: *Software-Defined Networking*; Balanceamento de Carga; *Machine Learning*; *Random Forest*; *OpenFlow*; *Mininet*; *RYU*.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ARP — *Address Resolution Protocol*

CS — Controlador *Simples*

CLI — *Command Line Interface*

CPU — *Central Processing Unit*

IP — *Internet Protocol*

KB/s — *Kilobytes* por segundo

LL — *Least-Loaded*

MAC — *Media Access Control*

Mbps — *Megabits* por segundo

ML — *Machine Learning*

OF — *OpenFlow*

ONOS — *Open Network Operating System*

OSPF — *Open Shortest Path First*

OVS — *Open vSwitch*

PNAD — Pesquisa Nacional por Amostra de Domicílios

POX — *Python OpenFlow* Controlador

QoS — *Quality of Service*

RR — *Round-Robin*

SDN — *Software-Defined Networking*

STP — *Spanning Tree Protocol*

SVM — *Support Vector Machine*

TCP — *Transmission Control Protocol*

veth — *Virtual Ethernet*

1. Introdução e Motivação

O crescimento exponencial do tráfego de dados em redes modernas impõe desafios cada vez maiores à gestão e à distribuição tráfego em redes. De acordo com a Pesquisa Nacional por Amostra de Domicílios (PNAD Contínua), o número de indivíduos conectados à internet cresceu mais de 6 milhões, chegando a 89,1% da população. [1]

Software-Defined Networking (SDN) surgem como um conceito central para enfrentar esse problema, ao separar o plano de controle do plano de dados e centralizar a inteligência de roteamento em controladores programáveis. Isso permite que a inteligência da rede seja centralizada em controladores lógicos e que a infraestrutura de encaminhamento se torne programável. Assim a rede não depende mais de configurações manuais em dispositivos, um processo frequentemente lento, propenso a erros e que dificulta a inovação devido à complexidade de gerenciamento e à baixa flexibilidade. A centralização do controle da rede possibilita maior agilidade operacional e maior eficiência na implementação de serviços por meio de *software*. [2]

Portanto, algoritmos de balanceamento de carga desempenham papel fundamental na maximização do aproveitamento dos recursos disponíveis e na redução de gargalos. Este trabalho propõe a avaliação comparativa de quatro estratégias de balanceamento, Controlador Simples (CS), *Round-Robin* (RR), *Least-Loaded* (LL) e *Machine Learning* (ML) sobre uma topologia hierárquica simulada no *Mininet* com o protocolo *OpenFlow* (OF) 1.3. O ambiente experimental foi projetado para reproduzir condições realistas de operação: a topologia é composta por três camadas, contendo 10 switches de acesso, 3 switches core e 5 servidores reais, com 100 hosts gerando tráfego segundo a distribuição de Pareto 80/20, na qual 20% dos hosts são responsáveis por 80% do volume total de dados. Um gargalo deliberado de 50 Mbps nos enlaces entre os switches core e os servidores garante que desequilíbrios de carga sejam visíveis e mensuráveis, tornando possível diferenciar

o desempenho dos algoritmos avaliados. Os quatro controladores foram implementados sobre o *framework* Ryu, executado em container *Docker* com comunicação via *OpenFlow 1.3*.

Os experimentos foram conduzidos em quatro cenários de tráfego distintos, cada um projetado para expor características específicas do problema de balanceamento: Carga Normal, que simula uma curva de uso gradual ao longo do dia; *Hotspot*, com pico de demanda concentrado por disparos simultâneos de *flows bulk*; Tráfego Misto, com todos os perfis ativos ao mesmo tempo; e Carga Sustentada, com chegada contínua e espaçada de novos *flows*. Cada cenário foi executado três vezes por controlador com semente de teste fixa, garantindo reprodutibilidade e comparabilidade dos resultados.

O objetivo central deste estudo é avaliar a eficácia de um controlador de rede baseado em ML em comparação a algoritmos clássicos de balanceamento, como RR e LL. O modelo proposto utiliza o histórico de carga dos servidores para a tomada de decisão, empregando uma metodologia específica de direcionamento de tráfego com o intuito de otimizar a distribuição de carga e o desempenho da infraestrutura. Os resultados obtidos buscam contribuir para a compreensão de quando a complexidade adicional de uma abordagem baseada em ML se justifica frente a controladores mais simples.

2. Fundamentação Teórica

2.1 *Software-Defined Networking*

2.1.1 Conceito

As SDN representam um paradigma arquitetural que propõe uma mudança fundamental na forma como as redes de computadores são projetadas, gerenciadas e operadas. Em arquiteturas de rede tradicionais, cada dispositivo possui seu plano de controle (responsável pelas decisões de encaminhamento) e seu plano de dados (responsável pelo encaminhamento efetivo dos pacotes) integrados e acoplados em

um mesmo hardware proprietário. Essa abordagem, embora funcional, impõe limitações à flexibilidade e à capacidade de gerenciamento em larga escala, pois qualquer alteração no comportamento da rede exige intervenção manual, dispositivo por dispositivo, frequentemente por meio de interfaces proprietárias e heterogêneas. [8]

O SDN rompe com esse modelo ao promover a separação explícita entre o plano de controle e o plano de dados. Nessa arquitetura, a inteligência de controle é centralizada logicamente em uma entidade de *software* denominada controlador SDN, que mantém uma visão global e unificada do estado da rede. Os dispositivos de encaminhamento tornam-se elementos simplificados, responsáveis apenas por executar as regras de encaminhamento instaladas pelo controlador, sem necessidade de lógica de controle local. Essa separação confere à rede um grau de programabilidade inédito: operadores e desenvolvedores passam a interagir com a infraestrutura de rede por meio de interfaces de programação de alto nível (APIs), podendo definir e modificar o comportamento da rede de forma centralizada, dinâmica e automatizada. [8]

A imagem a seguir ilustra a arquitetura de redes SDN, evidenciando a separação entre os planos de controle e de dados. Na camada superior, encontram-se as aplicações de rede, responsáveis por implementar políticas e serviços. No nível intermediário, o controlador centraliza a lógica da rede, tomando decisões sobre o encaminhamento de tráfego. Já na camada inferior, estão os dispositivos de rede (como switches), que compõem o plano de dados e executam as instruções recebidas do controlador por meio de interfaces, como o protocolo *OpenFlow.* Essa abordagem promove maior flexibilidade, programabilidade e gerenciamento centralizado da rede, facilitando a implementação de políticas dinâmicas e o controle eficiente dos recursos. [8]

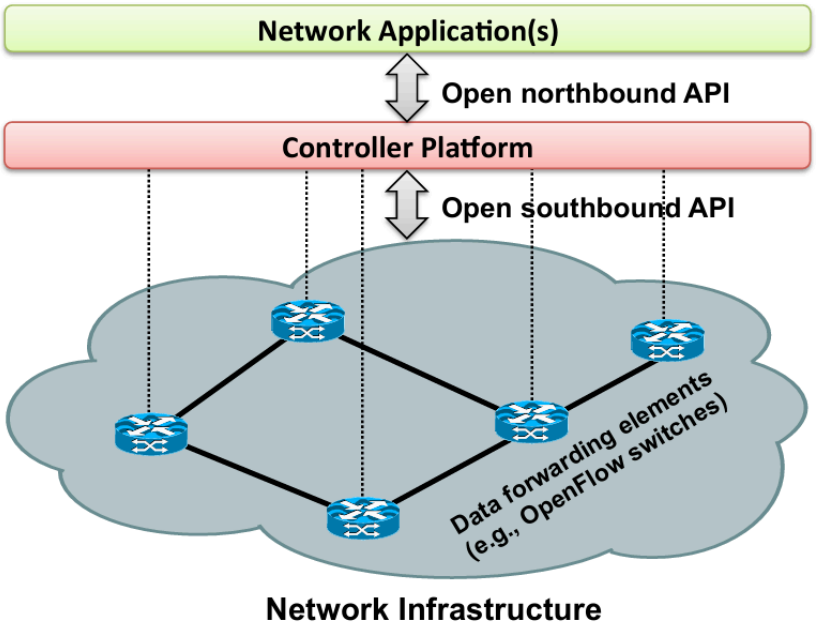

Figura 1 — Infraestrutura de Redes SDN

Fonte: [8]

2.1.2 *OpenFlow*

O OF é o protocolo de comunicação mais amplamente adotado para implementar a separação entre plano de controle e plano de dados em ambientes SDN. O protocolo define uma interface padronizada por meio da qual o controlador instala, modifica e remove regras de encaminhamento nas tabelas de *flows* dos *switches*. Cada regra, denominada *flow entry*, é composta por três elementos principais: um conjunto de campos de correspondência, que definem as características dos pacotes aos quais a regra se aplica; contadores, que registram estatísticas como número de pacotes e *bytes* processados; e um conjunto de ações, que especificam o que deve ser feito com os pacotes correspondentes, encaminhar para uma porta específica, descartar, modificar cabeçalhos, ou enviar ao controlador para processamento. [9]

2.1.3 Controlador

Em redes tradicionais, cada *switch* ou roteador toma suas próprias decisões de encaminhamento de forma independente, com base em tabelas construídas localmente por protocolos distribuídos como (*Open Shortest Path First)* (OSPF) ou

Spanning Tree Protocol (STP). Essa arquitetura distribuída é robusta, mas oferece pouca flexibilidade programática, alterar o comportamento da rede exige reconfigurar individualmente cada dispositivo. O paradigma SDN rompe com esse modelo ao separar explicitamente o plano de dados, responsável pelo encaminhamento dos pacotes, do plano de controle, responsável pelas decisões de como esse encaminhamento deve ocorrer. Nessa separação, o controlador SDN ocupa o plano de controle: é uma aplicação de *software* centralizada que mantém uma visão global da topologia e programa os dispositivos de rede de forma dinâmica.

A comunicação entre o controlador e os *switches* é padronizada pelo protocolo OF, que define uma interface bem específica entre os dois planos. Nesse modelo, os *switches* deixam de ser dispositivos autônomos e passam a funcionar como elementos de encaminhamento programáveis, eles aplicam regras chamadas de *flows*, que o controlador instala em suas tabelas de *flow*. Cada *flow* é composto por um conjunto de campos de correspondência, que definem quais pacotes ele afeta (como endereço *Media Access Control* (MAC) de origem, endereço IP de destino ou porta TCP), e um conjunto de ações a executar sobre esses pacotes, como encaminhar por uma porta específica, descartar, ou modificar campos do cabeçalho.

Quando um pacote chega a um *switch* e não corresponde a nenhum flow instalado, o *switch* não descarta o pacote nem toma uma decisão autônoma, ele encapsula o pacote em uma mensagem do tipo *PacketIn* e a envia ao controlador, que ao receber esse *PacketIn*, analisa o pacote, consulta seu estado interno sobre a topologia e as políticas em vigor, e toma uma decisão. Essa abordagem é eficiente e reduz a carga no plano de controle, pois o *switch* passa a processar o *flow* autonomamente após a primeira decisão. O controlador também pode emitir uma instrução direta ao *switch* descrevendo o que fazer com aquele pacote específico, sem instalar nenhum *flow* persistente. Nesse caso, cada pacote subsequente do mesmo *flow* continuará sendo enviado ao controlador via *PacketIn*, mantendo o controlador no caminho de decisão para todo o tráfego. [10]

2.2 *Mininet*

O *Mininet* é um emulador de rede, ou mais precisamente, “um sistema de orquestração de emulação de rede que permite a criação de topologias completas em um único *kernel Linux*.” [3] Diferente de um simulador puramente matemático, o *Mininet* usa virtualização para executar instâncias reais de *software* de rede, incluindo o *stack Transmission Control Protocol/Internet Protocol* (TCP/IP) do *kernel* e códigos de usuário, fazendo com que um *host* virtual se comporte como uma máquina real.

O funcionamento do *Mininet* baseia-se na combinação de recursos nativos do *kernel Linux* para criar isolamento e conectividade. Isso permite, por exemplo, que dois servidores *web* distintos rodem na mesma porta 80 dentro do mesmo sistema hospedeiro sem conflitos. A conectividade entre nós é estabelecida através de pares de *Ethernet virtual* (veth), que funcionam como cabos virtuais conectando interfaces ou portas de *switches*. Pacotes enviados por uma interface são entregues à outra, sendo percebidos pelo sistema como portas *Ethernet* totalmente funcionais. O controle de Tráfego garante fidelidade às características de hardware. O *Mininet* utiliza o tc do *Linux* para impor taxas de dados (largura de banda), atraso e perda de pacotes nos enlaces. O tráfego entre interfaces é geralmente gerenciado pelo *Open vSwitch* (OVS) ou pela ponte padrão do *Linux*. O OVS é fundamental em contextos de SDN, pois permite a programação do plano de dados via protocolo OF. [3]

2.2.1 Integração com *Software-Defined Networking*

O *Mininet* é uma ferramenta essencial para o desenvolvimento de SDN em projetos por permitir a separação clara entre o plano de dados e o plano de controle. Ele suporta o protocolo OF (como a versão 1.3), que define a interface de comunicação entre os *switches* e os controladores.

A ferramenta permite o uso de diversos controladores, como o *Ryu*, *Python OF* Controlador (POX) ou *Open Network Operating System* (ONOS), que podem ser executados dentro do ambiente emulado ou como controladores externos. Essa

flexibilidade possibilita que projetos de rede testados no *Mininet* sejam transferidos para hardware real com alterações mínimas, uma vez que o controlador "enxerga" a rede emulada de forma idêntica a uma rede física.

A manipulação do *Mininet* ocorre primordialmente através de três níveis de API em *Python*. Baixo nível: Manipulação direta de classes de nós e enlaces *(Host, Switch, Link*); Médio nível: Utiliza o objeto *Mininet* como um container para gerenciar a rede, permitindo métodos como *addHost(), addSwitch() e addLink()*; Alto nível: Abstração via classe Topo, que permite criar topologias parametrizadas e reutilizáveis através de *scripts Python*.

Além da automação por *scripts*, o *Mininet* oferece uma *Command Line Interface* (CLI) para interação manual, onde é possível testar conectividade (comando *pingall*), medir o *throughput* com o *iperf* ou executar qualquer programa *Linux* disponível no sistema hospedeiro dentro dos hosts virtuais. [3]

2.3 *Machine Learning*

O ML é um campo na ciência da computação voltado ao desenvolvimento de modelos matemáticos e algoritmos capazes de identificar padrões complexos em grandes volumes de dados. Diferente da programação convencional, o ML permite que sistemas realizem previsões ou tomadas de decisão de forma autônoma, evoluindo seu desempenho à medida que são expostos a novos exemplos.

2.3.1 Tipos de Aprendizado

Os algoritmos de ML são classificados em três tipos principais, de acordo a forma de aprendizado disponível:

Aprendizado Supervisionado: o algoritmo aprende a partir de um conjunto de pares entrada-saída rotulados, buscando inferir uma função que mapeie entradas e saídas. O objetivo é minimizar o erro entre a predição do modelo e o rótulo verdadeiro. Exemplos clássicos incluem a regressão linear, a regressão logística, as

árvores de decisão, as *Support Vector Machines* (SVM) e o *Random Forest*. As tarefas típicas são classificação e regressão.

Aprendizado Não Supervisionado: não há rótulos disponíveis. O algoritmo busca descobrir estruturas, padrões ou agrupamentos intrínsecos aos dados. Os principais exemplos são os algoritmos de *clustering* (como *K-Means*).

Aprendizado por Reforço: um agente aprende a tomar decisões sequenciais interagindo com um ambiente, recebendo recompensas ou penalidades conforme suas ações. O objetivo é maximizar a recompensa acumulada ao longo do tempo. [4]

2.3.2 *Random Forest*

Para compreender o *Random Forest*, é necessário, primeiramente, entender o conceito de árvore de decisão, que constitui sua unidade fundamental. Uma árvore de decisão é uma estrutura hierárquica de nós e ramificações que representa um conjunto de regras de decisão inferidas a partir dos dados de treinamento. Cada nó interno da árvore corresponde a um teste sobre um atributo, cada ramo representa o resultado desse teste e cada nó folha contém uma predição, seja uma classe (no caso de classificação) ou um valor numérico (no caso de regressão). [5]

A imagem a seguir demonstra o funcionamento de uma árvore de decisão.

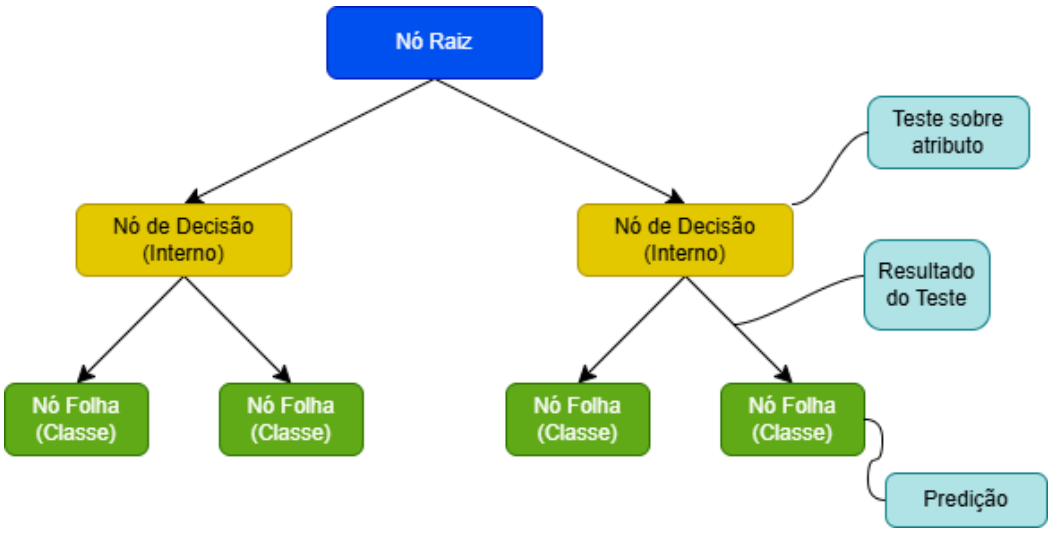

Figura 2 — Estrutura Árvore de Decisão

Fonte: Autoria Própria.

Apesar de sua interpretabilidade e simplicidade, as árvores de decisão isoladas tendem a sofrer com alta variância, tornando-se sensíveis a pequenas variações nos dados de treinamento, um fenômeno conhecido como *overfitting*. O *Random Forest* surge como solução a este problema ao aplicar dois conceitos centrais: o *bagging* e a aleatorização na seleção de atributos.

Bagging e Bootstrap

O *bootstrap* é uma técnica que consiste em gerar múltiplos subconjuntos de treinamento a partir do conjunto de dados original por meio de reamostragem com reposição. Cada subconjunto amostrado possui o mesmo tamanho do conjunto original, porém algumas instâncias podem aparecer mais de uma vez, enquanto outras podem ser excluídas. [6]

A partir de cada subconjunto *bootstrap*, uma árvore de decisão independente é treinada. Ao final do processo, as predições de todas as árvores são combinadas por meio de votação majoritária (para classificação) ou pela média aritmética (para regressão), resultando em uma predição final mais confiável e com menor variância do que qualquer árvore individualmente considerada. Está última etapa é denominada *bagging,* abreviação para *bootstrap aggregating*. [7]

Vantagens e Limitações

O *Random Forest* apresenta diversas vantagens que justificam sua ampla adoção em aplicações práticas. Do ponto de vista preditivo, o algoritmo é capaz de atingir alta acurácia em uma grande variedade de problemas sem requerer extensa engenharia de atributos. O modelo é naturalmente robusto a *outliers* e ruído nos dados, tolerante a atributos irrelevantes e capaz de capturar interações complexas entre variáveis sem que sejam especificadas explicitamente. Além disso, o algoritmo não pressupõe normalidade na distribuição dos dados nem relações lineares entre variáveis, tornando-o uma escolha adequada para dados reais frequentemente heterogêneos.

Do ponto de vista operacional, o treinamento das árvores pode ser paralelizado de forma trivial, o que proporciona ganhos significativos de eficiência computacional em ambientes com múltiplos processadores. A estimativa utilizando um subconjunto do conjunto de dados de treinamento elimina a necessidade de dados de validação separados, maximizando o uso dos dados disponíveis, e a importância dos atributos é computada como subproduto do treinamento sem custo adicional significativo.

2.4 Balanceamento de Carga

O balanceamento de carga desempenha um papel fundamental em redes modernas, especialmente em ambientes de SDN, uma vez que permite a distribuição eficiente das requisições entre diferentes recursos computacionais. Sua principal finalidade é evitar a sobrecarga de determinados componentes, reduzindo a ocorrência de gargalos, falhas e degradação da Qualidade de Serviço (QoS). Ele contribui diretamente para a melhoria do desempenho da rede, ao otimizar a utilização de recursos, minimizar atrasos, reduzir perda de pacotes e aumentar a confiabilidade do sistema . Além disso, técnicas adequadas permitem prever e mitigar congestionamentos antes que impactem o funcionamento da rede, tornando-se um elemento essencial em cenários com alta demanda e grande volume de dados. [11]

Para avaliar a eficiência dessa distribuição de carga, utiliza-se o índice de equidade, sendo o mais conhecido o índice de Jain. Esse índice mede o quão uniformemente os recursos estão sendo distribuídos entre os elementos do sistema, assumindo

valores no intervalo de 0 a 1, onde valores próximos de 1 indicam uma distribuição equitativa, e valores próximos de 0 indicam forte desigualdade. Em termos práticos, o índice de equidade permite quantificar o grau de balanceamento da rede, sendo amplamente utilizado como métrica de desempenho em estudos de balanceamento de carga . Dessa forma, ele fornece uma base quantitativa sólida, auxiliando na identificação de soluções mais eficientes e justas na alocação de recursos da rede. [12]

3. Metodologia e Desenvolvimento

3.1 Plataforma de Simulação

O ambiente experimental foi desenvolvido integralmente em ambiente virtualizado utilizando o *Mininet* como simulador de redes. O *Mininet* permite criar topologias completas como *switches*, *hosts* e enlaces dentro de um único sistema *Linux*, executando instâncias reais de *software* de rede.

Os *switches* são implementados via *Open vSwitch* (OVS), os controladores via *framework Ryu* em container *Docker* com *Python 2.7*, e a comunicação entre planos de dados e controle segue o protocolo OF 1.3 . Para geração de tráfego foi utilizado o *iperf* em modo cliente-servidor, com múltiplas conexões simultâneas, e os *flows* são gerados por *threads Python* independentes com atrasos programados para simular uma chegada distribuída.

Com o objetivo de garantir reprodutibilidade será disponibilizado ao final do relatório um link com o ambiente virtual que foi realizada a simulação, tal qual os arquivos para gerar os controladores, topologia e tráfego. (ver Apêndice A)

3.2 Topologia Hierárquica

A topologia implementada é hierárquica em três camadas, conforme descrito a seguir:

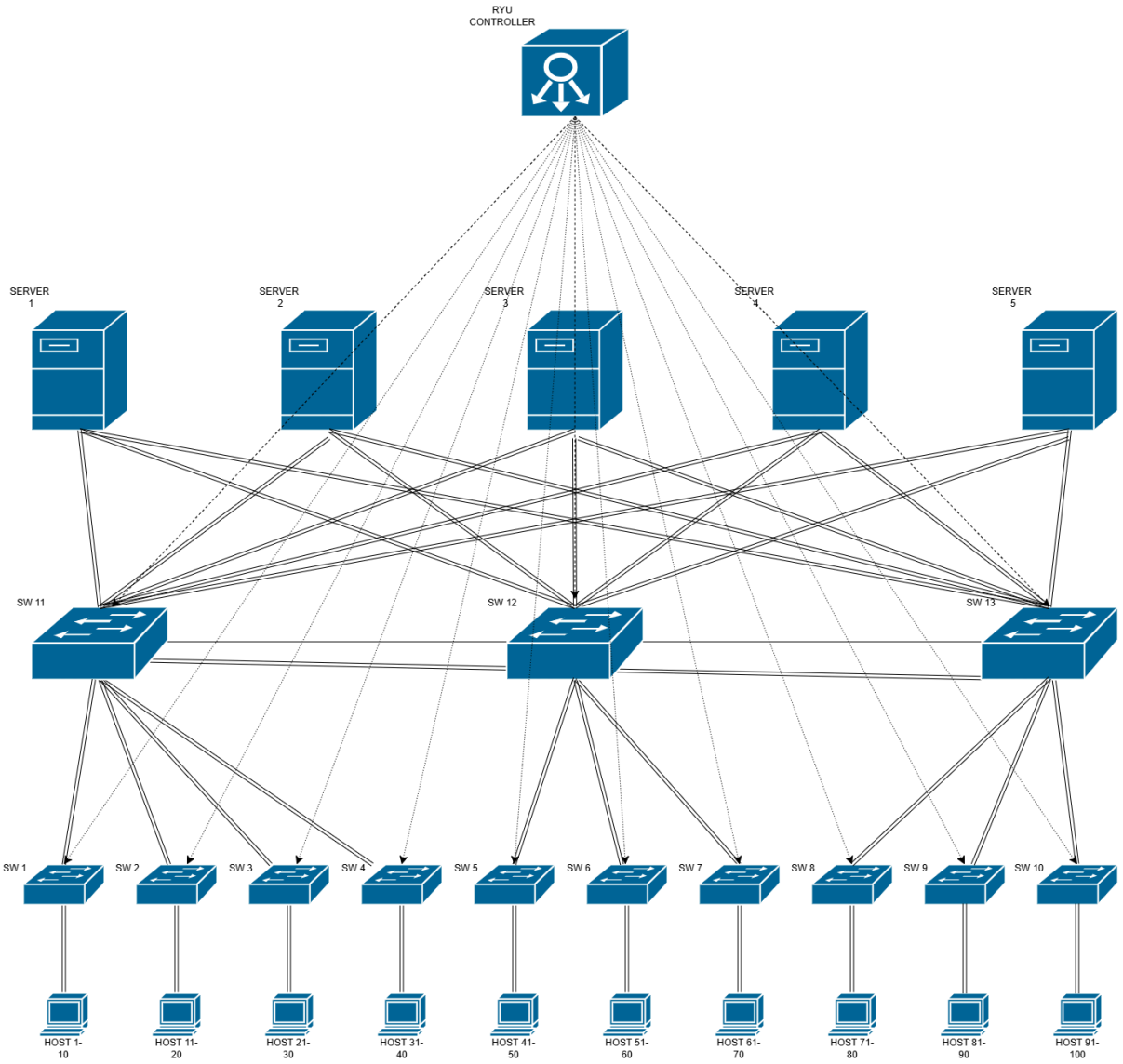

Figura 3 — Topologia hierárquica da rede simulada no Mininet

Fonte: Autoria Própria.

Camada de Acesso

- 10 *switches* de acesso (s1 a s10)
- 100 *hosts* distribuídos, 10 *hosts* por *switch*
- Endereçamento: 10.0.X.Y/16 (X = índice do *switch* 0–9, Y = número do *host* 1–10)
- Largura de banda *host* ↔ *switch*: 100 Mbps

Camada *Core*

- 3 *switches core* (s11, s12, s13) conectados entre si
- s1–s4 conectados a s11; s5–s7 conectados a s12; s8–s10 conectados a s13
- Largura de banda acesso↔*core*: 1 Gbps; *core*↔*core*: 10 Gbps

Camada de Servidores

- 5 servidores reais (*server1 a server5*), IPs: 10.0.100.1 a 10.0.100.5
- IP virtual de balanceamento: 10.0.200.1 — destino único de todos os *flows* de teste
- Cada servidor conectado simultaneamente aos 3 *switches* core
- Largura de banda *core*↔servidor: 50 Mbps

O gargalo de 50 *Megabits* por segundo (Mbps) no enlace core→servidor é deliberado: garante que *flows bulk* de 80 Mbps saturem o enlace, tornando o desequilíbrio entre servidores visível e mensurável e, consequentemente, diferenciando o desempenho dos algoritmos de balanceamento.

3.3 Mecanismo de Redirecionamento

O IP virtual 10.0.200.1 é o destino de todos os *flows* de teste. Os controladores interceptam requisições *Address Resolution Protocol* (ARP) para esse IP e decidem para qual servidor real redirecionar cada novo cliente. O redirecionamento é realizado trocando o 10.0.200.1 pelo IP real do servidor escolhido e trocando o MAC do IP virtual pelo MAC real do servidor, sem instalação de *flows* para que o controlador processe cada pacote individualmente, permitindo decisões dinâmicas por *flows*.

3.4. Controladores Implementados

3.4.1 Controladores SDN

Neste trabalho, essa distinção tem implicações diretas na arquitetura de balanceamento implementada. Para os *switches* de acesso, responsáveis pela conectividade dos *hosts*, o controlador utiliza a abordagem de *FlowMod* (aprendendo a localização do host via *PacketIn*), instalando *flows* que permitem ao *switch* encaminhar pacotes subsequentes daquele *flow* sem intervenção adicional. Já para os *switches core* (ponto de decisão entre os clientes e os servidores), o controlador deliberadamente opta pelo *PacketOut* sem instalação de *flows*. Essa escolha garante que cada pacote destinado ao IP virtual de balanceamento (10.0.200.1) passe obrigatoriamente pelo controlador, que altera os campos de destino do pacote antes de instruir o *switch* a encaminhar o pacote pela porta correta. Essa granularidade por pacote é o que permite ao controlador aplicar dinamicamente a lógica de balanceamento a cada novo *flow*, sem que os *switches core* possuam qualquer lógica de decisão própria.

Além do plano de encaminhamento, o controlador também atua como coletor de telemetria da rede. Por meio de mensagens, ele requisita periodicamente aos *switches core* as estatísticas de cada porta, em particular o volume de *bytes* transmitidos e recebidos em cada interface conectada a um servidor. Essas estatísticas são a matéria-prima para os algoritmos de balanceamento baseados em carga, sem elas, o controlador não teria como distinguir um servidor ocioso de um servidor saturado. O intervalo de coleta, a forma como essas métricas são processadas e o algoritmo que as transforma em uma decisão de roteamento são justamente os elementos que diferenciam os quatro controladores implementados e avaliados neste trabalho.

É importante destacar que os *hosts* da topologia não interagem diretamente com o controlador, eles são *endpoints* comuns que desconhecem a existência do plano de controle. O controlador aprende sua localização passivamente, à medida que seus pacotes geram eventos *PacketIn* nos *switches* de acesso, mas nunca os programa

ou configura diretamente. Da mesma forma, o mapeamento entre as portas dos *switches core* e os servidores físicos é definido estaticamente no código do controlador, refletindo a topologia fixa da rede experimental.

3.4.2 Controlador *RYU*

A implementação do controlador SDN neste trabalho adota uma arquitetura que separa o ambiente de simulação de rede do ambiente de execução do controlador. O *Mininet* é executado diretamente na máquina virtual, enquanto o *framework Ryu* (responsável por hospedar a lógica do controlador) roda dentro de um container *Docker*. A comunicação entre os dois se dá via protocolo OF sobre TCP, de forma transparente e sem qualquer mecanismo especial de rede entre eles.

Após o *switch* ser criado, cada *switch* é configurado para se conectar a um controlador externo no endereço 127.0.0.1:6633 via TCP. O *switch* fica em estado de tentativa de conexão contínua até que um controlador apareça nesse endereço e aceite a sessão OF. A partir desse momento, o *switch* passa a operar sob controle remoto: envia eventos ao controlador quando não sabe como encaminhar um pacote, e aplica as instruções que recebe de volta.

O *framework Ryu* é executado dentro de um container *Docker* baseado na imagem pública *osrg/ryu*, que já contém o *Python* 2.7, o *Ryu* e todas as suas dependências instaladas. Nenhuma modificação foi feita na imagem em si, ela é utilizada como ambiente pronto. O código do controlador desenvolvido neste trabalho é um aplicativo *Ryu*, um arquivo *Python* que utiliza as APIs do *framework* para definir o comportamento do controlador: quais eventos processar, como reagir a um *PacketIn*, como calcular a carga dos servidores e como emitir *Packet Outs* e *FlowMods*. Esse arquivo é injetado dentro do container em tempo de execução via montagem de volume, isso foi necessário pois o *Mininet* e o *Ryu* rodam originalmente em *Python* 2.7 (versão recomendada) porém a máquina virtual utilizada no experimento rodava em *Python* 3.10.14 e o *downgrade* gerava problemas no sistema operacional. Para contornar essa incompatibilidade, optou-se por executar o *Ryu* em *Docker*. O

Docker, nesse cenário, é essencialmente um encapsulamento para o ambiente *Python* 2.7 com o *Ryu* instalado; a comunicação em si é TCP local comum, sem nenhuma camada adicional de virtualização de rede entre os componentes.

Essa arquitetura oferece uma vantagem prática relevante para o contexto de pesquisa: trocar o controlador em execução é uma operação simples de substituição do arquivo montado via volume e reinicialização do container, sem qualquer alteração na topologia do *Mininet* ou nos *switches*. Os quatro experimentos deste trabalho foram conduzidos exatamente dessa forma, garantindo que o ambiente de rede permanecesse idêntico entre as execuções e que a única variável alterada fosse a lógica de controle.

O desenvolvimento dos controladores apresentados neste trabalho contou com o auxílio de ferramentas de inteligência artificial, em particular o modelo *Claude*, utilizando como assistente na geração e refinamento do código *Python*. As implementações foram supervisionadas, validadas e ajustadas pelos autores, garantindo que o comportamento de cada controlador estivesse em conformidade com os requisitos funcionais e com a arquitetura definida para o experimento.

3.4.3 Controlador Simples

Implementa um *learning switch* global sem qualquer lógica de balanceamento. Aprende a localização dos MACs via *PacketIn* e instala *flows* diretos entre origem e destino. Para o IP virtual, não realiza nenhuma substituição de endereço, os pacotes destinados a 10.0.200.1 são tratados como qualquer outro destino, resultando em *flood* quando o MAC não é conhecido. Na prática, o primeiro servidor que responder ao ARP fica com todos os clientes, gerando concentração extrema de carga pois todo tráfego gerado durante o experimento irá apenas para esse servidor.

Serve como referência inferior: qualquer algoritmo de balanceamento deve superar o CS em todos os cenários.

3.4.4 Controlador *Round-Robin*

O algoritmo RR é uma das estratégias de balanceamento de carga na qual os servidores disponíveis são organizados em uma sequência circular, e cada nova conexão é atribuída ao próximo servidor dessa sequência, independentemente de qualquer outro fato. Quando o último servidor da lista é atingido, o ciclo recomeça a partir do primeiro. Essa abordagem garante que cada servidor receba exatamente a mesma quantidade de clientes, porém ela não considera a heterogeneidade que pode ocorrer de cliente para cliente a distribuição é, por construção, perfeitamente uniforme em termos de número de conexões.

Quando um cliente envia sua primeira requisição ao IP virtual de balanceamento 10.0.200.1, o controlador intercepta o evento *PacketIn* gerado pelo *switch core*, consulta o índice atual para determinar qual servidor deve atender aquela conexão, e em seguida incrementa o índice para a próxima decisão de servidor. O redirecionamento é então executado, alterando o endereço MAC e o endereço IP de destino do pacote para os valores reais do servidor escolhido, antes de instruir o *switch* a encaminhar o pacote pela porta correspondente.

Uma vez que um cliente é mapeado a um servidor, esse mapeamento é persistido. Enquanto o *flow* TCP daquele cliente estiver ativo, todos os seus pacotes subsequentes são direcionados ao mesmo servidor, preservando a afinidade de sessão necessária para o correto funcionamento das conexões TCP. Essa afinidade é fundamental: redirecionar pacotes de uma mesma conexão TCP para servidores distintos resultaria na quebra da sessão, pois o estado da conexão está associado a um único *endpoint* servidor.

Para evitar que o cache cresça indefinidamente e acumule entradas de clientes que já encerraram suas conexões, o controlador executa periodicamente uma limpeza a cada 5 segundos. Clientes cujos *flows* não apresentam atividade recente são removidos do cache, liberando-os para receberem um novo mapeamento na próxima conexão.

O RR carrega uma limitação estrutural relevante para o cenário avaliado neste trabalho: ele é completamente insensível ao estado de carga real de cada servidor no momento da decisão. O algoritmo não distingue entre um servidor ocioso e um servidor que está processando múltiplos *flows bulk* de 80 Mbps cada. A décima primeira conexão será atribuída ao mesmo servidor que receberia a primeira, independentemente do estado em que se encontra.

Essa característica tem implicações diretas em cenários de tráfego heterogêneo. Quando a distribuição de tipos de *flow* não é uniforme ao longo do tempo, o RR pode concentrar múltiplos *flows* pesados no mesmo servidor simplesmente porque eles chegaram em sequência favorável ao índice. Um servidor que recebeu três conexões *bulk* consecutivas estará saturado enquanto outros permanecem ociosos, e o algoritmo não possui nenhum mecanismo para perceber ou corrigir esse desequilíbrio enquanto as sessões estiverem ativas.

É por essa razão que o RR serve, neste trabalho, como referência intermediária: superior ao CS, que não realiza balanceamento algum, mas inferior a qualquer algoritmo que incorpore informação sobre o estado real da rede no momento da decisão.

3.4.5 Controlador *Least-Loaded*

O controlador LL representa uma evolução direta sobre o RR ao incorporar informação sobre o estado real da rede no momento de cada decisão de roteamento. Em vez de distribuir conexões de forma cíclica e cega, ele observa continuamente a carga em cada servidor e direciona novos clientes àquele que apresenta menor utilização no instante da decisão. Observa-se que caso múltiplas requisições intensivas chegassem simultaneamente de tempo o controlador tenderia a direcionar requisições ao mesmo servidor pois não houve tempo para recalcular a nova carga, alternativas mais agressivas, como a redução do intervalo de remapeamento dos servidores gerava muitas *logs* para o controlador calcular impactando no funcionamento da mesma, portanto como opção mais viável adicionalmente colocamos o algoritmo de decisão RR para tomar decisões em que as cargas em todos os servidores estejam muito similares.

A base de funcionamento do LL é a coleta periódica de estatísticas de porta dos *switches core*. A cada 2 segundos, o controlador envia mensagens aos três *switches core* (s11, s12 e s13) solicitando as contagens acumuladas de *bytes* transmitidos e recebidos em cada porta. Como cada porta dos *switches core* está mapeada a um servidor específico, essas contagens representam diretamente o volume de tráfego que cada servidor está processando. O mapeamento entre portas e servidores é fixo e definido estaticamente no código do controlador, refletindo a topologia física da rede experimental.

A partir das contagens acumuladas de dois instantes consecutivos, o controlador calcula a taxa de tráfego atual de cada servidor combinando transmissão e recepção:

rate = (tx_*bytes*_atual − tx_*bytes*_anterior + rx_*bytes*_atual − rx_*bytes*_anterior) / Δt

Essa taxa representa o *throughput* bilateral do servidor no intervalo de medição mais recente, capturando tanto o tráfego de dados enviado ao cliente quanto as confirmações e requisições recebidas.

Um problema prático em qualquer sistema de monitoramento de rede é a variabilidade natural das medições, a taxa de tráfego de um servidor pode oscilar significativamente entre dois intervalos de coleta consecutivos, mesmo sem mudança real na carga. Reagir diretamente a cada medição bruta tornaria o algoritmo instável, redirecionando clientes com base em flutuações momentâneas que não refletem a carga sustentada do servidor.

Para mitigar esse problema, o controlador aplica uma Média Móvel Exponencial sobre as taxas calculadas:

$$server\_load = 0{,}4 \times carga\_anterior + 0{,}6 \times taxa\_atual$$

O coeficiente de suavização *alpha* igual a 0,6 estabelece um equilíbrio deliberado entre reatividade e estabilidade: 60% do peso da carga estimada vem da medição mais recente, enquanto 40% vem do histórico acumulado. Esse valor foi escolhido para permitir que o controlador detecte mudanças reais de carga com rapidez

suficiente para influenciar as decisões de roteamento, sem que ruídos pontuais de medição provoquem oscilações desnecessárias na alocação de clientes.

A cada nova conexão, o controlador consulta o vetor de cargas suavizadas e identifica o servidor com menor valor. No entanto, antes de tomar a decisão, ele verifica se a diferença entre a carga máxima e a carga mínima observadas supera um limiar fixado em 5 *Kilobytes* por segundo (KB/s), caso não seja menor será usado a lógica RR como desempate para tomar a decisão de encaminhamento caso o limiar seja maior, o encaminhamento se dará para o servidor com menor valor.

Assim como o RR, o LL opera com afinidade de sessão TCP: uma vez que um cliente é mapeado a um servidor, todos os pacotes daquela sessão seguem para o mesmo destino enquanto o *flow* estiver ativo. O controlador também mantém o mesmo mecanismo de cache com limpeza periódica a cada 5 segundos para clientes inativos.

Isso implica uma limitação fundamental: o LL só pode agir no momento da decisão inicial de cada sessão. *Flows* longos permanecem no servidor originalmente alocado durante toda a sua duração, mesmo que esse servidor se torne o mais carregado da rede logo após o início da sessão. O controlador não possui mecanismo de migração de sessões ativas. Consequentemente, seu ganho sobre o RR é mais pronunciado em cenários onde novos *flows* chegam continuamente, oferecendo janelas de decisão frequentes, e menos pronunciado em cenários onde um pico de *flows* pesados chega simultaneamente saturando todos os servidores antes que o algoritmo de menor carga possa agir de forma diferenciada.

3.4.6 Controlador ML

O ML representa uma abordagem mais sofisticada. Em vez de seguir uma heurística determinística, como distribuir conexões em ciclo ou sempre escolher o servidor menos carregado no instante da decisão, ele delega a decisão de roteamento a um modelo de ML previamente treinado, capaz de inferir qual servidor deve receber uma nova conexão a partir de um conjunto estruturado de características do estado atual da rede.

Ao iniciar, o controlador carrega um modelo Random Forest serializado e treinado *offline*, utilizando a biblioteca *scikit-learn*. Esse carregamento ocorre uma única vez, na inicialização do processo *Ryu*, e o modelo permanece em memória durante toda a execução do controlador. A partir desse momento, cada nova decisão de roteamento é resolvida por uma chamada baseada no modelo, sem nenhuma atualização ou retreinamento em tempo real. O ML portanto aplica o conhecimento adquirido durante o treinamento *offline*, mas não aprende com as decisões que toma durante a execução.

Assim como o LL, o ML coleta estatísticas de porta dos switches core a cada 2 segundos, e mantém o mesmo mecanismo de suavização por Média Móvel Exponencial com *alpha* igual a 0,6 para estimar a carga atual de cada servidor. Essa base de monitoramento é compartilhada entre os dois controladores porque o modelo foi treinado com dados gerados pelo próprio LL durante as sessões de coleta. Portanto, as características que o modelo aprendeu a interpretar são exatamente as mesmas métricas que o LL calcula.

A cada nova decisão de roteamento, o controlador extrai 26 características do estado atual da rede, organizadas em cinco grupos:

O primeiro grupo contém a carga atual de cada um dos cinco servidores, expressa em *bytes* por segundo após a suavização exponencial. O segundo grupo contém a carga anterior de cada servidor o *snapshot* do intervalo imediatamente anterior à medição atual fornecendo ao modelo uma referência histórica imediata. O terceiro grupo é composto pela diferença entre carga atual e anterior para cada servidor, representando explicitamente a tendência de crescimento ou redução de carga em cada um deles. O quarto grupo contém a proporção de carga relativa de cada servidor (proporção da carga daquele servidor em relação à carga total da rede) , capturando a distribuição proporcional entre os servidores independentemente do volume absoluto de tráfego. O quinto e último grupo reúne seis métricas globais da rede: o desvio padrão das cargas, o intervalo entre a maior e a menor carga, os valores máximo e mínimo absolutos, a entropia de Shannon da distribuição de carga entre os servidores, e o total de *flows* ativos no momento da decisão.

O modelo *Random Forest* recebe o vetor com todos os 26 valores e retorna o índice do servidor recomendado. Esse índice é imediatamente utilizado pelo controlador para executar o redirecionamento, exatamente da mesma forma que os demais controladores: alterando o endereço MAC e o endereço IP de destino do pacote para os valores reais do servidor escolhido e instruindo o *switch* core a encaminhar pela porta correspondente.

Por ter sido treinado exclusivamente em sessões controladas pelo LL, espera-se que o comportamento do ML apresente similaridades com o do LL. Embora o modelo tenha liberdade para escolher qualquer servidor com base nas 26 características projetadas, a distribuição de carga observada durante todo o treinamento foi moldada pelas decisões do LL. O modelo aprendeu a partir de uma rede que sempre esteve sob controle do LL, o que significa que estados de desequilíbrio severo que surgiriam sob outras políticas de balanceamento não estão muito bem representadas. Essa dependência do ambiente de treinamento é um fator relevante para interpretar os resultados obtidos e é discutida em maior profundidade no capítulo seguinte, que detalha o treinamento do modelo, a composição das características, o processo de geração dos rótulos e a avaliação do desempenho preditivo.

3.5. Modelo de Machine Learning

3.5.1 Coleta de Dados e rótulo de Treinamento

As sessões de treinamento foram conduzidas utilizando o mesmo gerador de tráfego dos experimentos de avaliação, porém com sementes distintas com sementes de 1 a 5 combinadas com os cinco cenários de tráfego definidos (Os cinco cenários foram utilizados nas sessões de coleta de dados para o treinamento do modelo. Contudo, para fins de avaliação comparativa dos controladores, optou-se por apresentar apenas quatro cenários, dado que o quinto não evidenciou diferenciação estatisticamente relevante entre os algoritmos avaliados), totalizando 25 execuções independentes. O uso de sementes diferentes das utilizadas nos testes é uma decisão deliberada: garante que o modelo seja treinado com padrões de tráfego distintos dos que serão usados na avaliação, evitando que ele simplesmente memorize sequências específicas de *flows*.

Durante cada execução, o coletor opera em dois momentos distintos. O primeiro é a coleta periódica: a cada 2 segundos, enquanto houver tráfego ativo na rede, o coletor extrai o vetor de 26 características do estado atual e calcula o rótulo ideal para aquele instante. O segundo é a coleta por evento: a cada nova decisão de roteamento tomada pelo controlador LL, o coletor registra o estado da rede naquele momento exato e calcula o rótulo correspondente. Essa dupla estratégia de coleta garante que o conjunto de dados capture tanto os estados estacionários da rede entre decisões quanto os estados transitórios no exato momento em que uma nova conexão precisa ser atribuída a um servidor. Ao final das 25 execuções, o conjunto de dados consolidado continha 2.493 amostras.

O rótulo de cada amostra não é a decisão que o controlador LL tomou naquele momento, mas sim o servidor que maximizaria uma pontuação composta calculada independentemente para cada um dos cinco servidores disponíveis. Para cada servidor candidato, o coletor simula a adição de um *flow* médio estimado de 500 KB/s à sua carga atual (representando a chegada de uma nova conexão hipotética) e calcula a pontuação resultante dessa simulação:

pontuação = 0,35 × Equidade Jain + 0,25 × Entropia + 0,25 × Estabilidade + 0,15 × *Headroom*

O servidor que maximiza essa pontuação entre os cinco candidatos é definido como o rótulo da amostra. O modelo aprende, portanto, a identificar a decisão ótima segundo o critério de pontuação composta, e não a imitar o comportamento do LL.

Cada componente da pontuação captura uma dimensão distinta de qualidade:

A Equidade de Jain mede a equidade da distribuição entre servidores

Jain = ((soma carga atual dos servidores)² / (Número de servidores × (soma do quadrado das cargas atuais nos servidores));

[11]

A Entropia de Shannon penaliza a concentração de carga medindo a diversidade da distribuição. Primeiro normaliza as cargas para que somem 1 depois aplica a fórmula da entropia:

Entropia normalizada = -Σ ((carga atual normalizada × log(carga atual normalizada)) / log(Número de servidores)

A Estabilidade penaliza servidores com carga crescendo rapidamente, para isso o coletor usa as cargas anteriores:

diff_servidor(normalizado) = (carga_atual[servidorX] - carga_prev[servidorX]) / capacidade máxima do enlace (50 Mbps)

se diff_servidor for maior que 0 o mesmo assume o valor do cálculo, se for menor ou igual a 0 o valor assumido fica como 0

Estabilidade = 1 - diff_servidor(normalizado)

Headroom representa a folga disponível até a capacidade máxima estimada de 50 Mbps.

***headroom* = (capacidade_maxima - carga_servidor) / capacidade_maxima**

O servidor que conseguir a maior pontuação será o servidor identificado como o correto.

3.5.2 Treinamento e Avaliação

Parâmetro	Valor
Algoritmo	*Random Forest* (100 árvores)
características de entrada	26
Amostras coletadas	2.493
Split treino/validação	80% / 20% (1.994 / 499)
Acurácia (validação)	88,0%
Cross-validation (5-fold)	60,9% ± 5,4%
Sementes de treinamento	1, 2, 3, 4, 5
Semente de teste	42

Tabela 1 — Parâmetros do modelo Random Forest

Fonte: Autoria Própria.

O conjunto de dados foi dividido em dois subconjuntos mutuamente exclusivos seguindo a proporção 80/20: 1.994 amostras destinadas ao treinamento e 499 amostras destinadas à validação. A divisão foi realizada com Amostragem proporcional (garantindo que a proporção de amostras de cada classe de servidor fosse preservada em ambos os subconjuntos)

O primeiro método foi a avaliação no conjunto de validação *hold-out*, as 499 amostras separadas antes do treinamento e nunca utilizadas durante ele. O modelo atingiu acurácia de 88,0% nesse conjunto, indicando que em 88 de cada 100 decisões o servidor escolhido pelo modelo coincidiu com o servidor ótimo segundo a pontuação composta.

O segundo método foi a validação cruzada de 5 partições sobre o conjunto de dados completo. Nessa técnica, o conjunto de dados é dividido em 5 blocos iguais e o modelo é treinado e avaliado 5 vezes, usando a cada rodada um bloco diferente como validação e os quatro restantes como treinamento. A acurácia média obtida foi de 60,9% com desvio padrão de 5,4%, substancialmente inferior aos 88,0%.

3.5.3 Importância das características

As características mais importantes identificadas pelo *Random Forest* foram as proporções de carga relativa de cada servidor, demonstrando que o modelo aprendeu que a distribuição proporcional entre servidores é mais informativa do que os valores absolutos de carga. O ranking completo das top 10 características é:

- *server3_load_ratio*: 0,0902
- *server5_load_ratio*: 0,0874
- *server4_load_ratio:* 0,0809
- *server1_load_ratio:* 0,0805
- *server2_load_ratio:* 0,0703
- *load_entropy*: 0,0461
- *server2_load:* 0,0406
- *server3_load:* 0,0402
- *server1_load:* 0,0398

- *total_active_flows:* 0,0373

3.6 Gerador de Tráfego e Cenários de Teste

Para comparar os quatro controladores de forma justa, cada um precisa enfrentar exatamente o mesmo tráfego. Se o tráfego fosse puramente aleatório a cada execução, uma diferença de resultado poderia ser simplesmente porque um controlador "teve sorte" de receber *flows* mais leves. Ao mesmo tempo, tráfego completamente determinístico não representa a realidade de uma rede.

A solução foi aleatoriedade controlada por semente: o gerador produz sequências que parecem aleatórias, mas são 100% reproduzíveis quando iniciadas com o mesmo número inicial.

3.6.1 Distribuição de Pareto 80/20

O gerador de tráfego implementa a distribuição de Pareto 80/20: 20% dos hosts são classificados como pesados e geram 80% do tráfego, enquanto os 80% restantes são leves. Os perfis de tráfego utilizados são:

- *web*: 1 Mbps / 5 s — navegação *web* (perfil leve)
- *video*: 5 Mbps / 12 s — streaming de vídeo (perfil intermediario)
- *backup*: 50 Mbps / 12 s — backup de dados (perfil pesado)
- *voip*: 1 Mbps / 10 s — *VoIP* (perfil leve)
- *bulk*: 80 Mbps / 12 s — transferência em massa (perfil pesado)

Perfil intermediário podendo estar presente em cargas leves ou pesadas.

3.6.2 Cenários Avaliados

C1 — Carga Normal

Simula curva de uso ao longo do dia em três fases: fase 1 (t = 0–15 s) com 15 *flows* leves; fase 2 (t = 15–40 s) com 20 *flows* pesados e 15 *flows* leves intercalados; fase 3 (t = 40–60 s) com 10 *flows* leves finais. Total: ~60 *flows*.

C2 — *Hotspot*

Simula pico de demanda concentrado: 10 *bulks* simultâneos disparados em t = 0, seguidos de 25 *flows web* e 18 *flows* médios com chegada gradual. Total: ~53 *flows*.

C3 — Tráfego Misto

Todos os tipos de tráfego ativos simultaneamente, com *backups* concentrados para criar desequilíbrio deliberado entre t = 10 e 20 s. Total: ~63 *flows*.

C4 — Carga Sustentada

Chegada contínua de novos *flows* a cada 3 segundos ao longo de toda a execução, com probabilidade 20% de *flow* pesado e 80% de *flow* leve. Total: ~19 *flows*.

4. Resultados e Análise

4.1 *Throughput* Médio por Cenário (Mbps)

Os resultados a seguir representam médias de 3 execuções independentes por controlador por cenário, com semente de teste fixa (42). A taxa de sucesso de entrega de *flows* foi de 100% em todos os controladores e cenários.

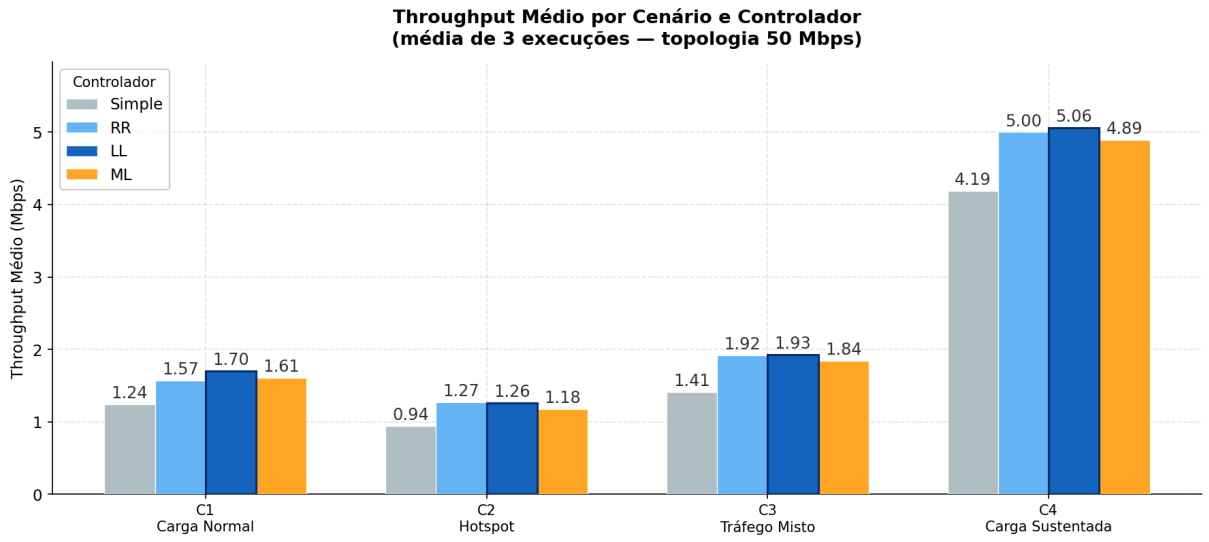

Figura 4 - *Throughput* Médio por Cenário e Controlador

Fonte: Autoria Própria.

4.2 Observações e Discussão

O cenário de carga normal, que simula uma curva de uso gradual ao longo do dia com três fases distintas de intensidade, revelou a maior diferenciação de desempenho entre os algoritmos de balanceamento. O CS registrou *throughput* médio mais baixo de 1,24 Mbps, confirmando o comportamento esperado de concentração de carga em um único servidor após a primeira resposta ARP. O RR atingiu 1,57 Mbps, representando uma melhora substancial pelo simples fato de distribuir as conexões entre os cinco servidores de forma uniforme. O ML obteve 1,61 Mbps, ligeiramente superior ao RR mas abaixo do LL, comportamento consistente com a hipótese de que o modelo tende a reproduzir parcialmente as decisões do LL por ter sido treinado em um ambiente controlado por ele. O LL alcançou o melhor resultado do cenário com 1,70 Mbps. Esse é o cenário onde o LL apresenta sua maior vantagem sobre o RR o que pode ser explicado pela estrutura temporal do C1: a chegada gradual de *flows* em fases bem definidas oferece ao LL janelas de decisão frequentes onde a detecção de carga diferencial entre os servidores é efetiva. Quando os *flows* pesados da fase 2 começam a chegar, o LL já possui medições de carga suficientemente precisas para desviar novos clientes dos servidores que receberam os primeiros *flows bulk*, enquanto o RR continua distribuindo ciclicamente sem considerar esse desequilíbrio.

O cenário de *hotspot*, caracterizado pelo disparo simultâneo de 10 *flows bulk* em t=0, foi o mais desafiador para todos os algoritmos e produziu os menores valores absolutos de *throughput* médio em todos os controladores. O CS registrou 0,94 Mbps, o único cenário onde ficou abaixo de 1 Mbps, evidenciando o impacto severo da concentração extrema de carga. O RR obteve 1,27 Mbps e o LL 1,26 Mbps, ficando praticamente empatado com diferença de apenas 0,01 Mbps. O ML registrou 1,18 Mbps, abaixo de ambos. A explicação para o empate entre LL e RR reside na dinâmica do próprio cenário: os 10 *flows bulk* são disparados simultaneamente antes

que o LL tenha tido qualquer oportunidade de coletar estatísticas de carga diferencial. Com o intervalo de coleta de 2 segundos, o primeiro ciclo de medição do LL captura uma rede já saturada de forma relativamente uniforme entre os servidores, pois foi feita o R.R para a primeira coleta caso os servidores estivessem a mesma quantidade de carga portanto, como foi o RR que atribuiu os 10 *flows* iniciais do LL ciclicamente, já garantiu uma distribuição razoável entre eles. A partir desse ponto, a vantagem informacional do LL sobre o RR é mínima porque todos os servidores estão igualmente sobrecarregados, eliminando a diferença de carga que o algoritmo precisa para tomar decisões superiores às do ciclo simples.

O cenário de tráfego misto, com todos os perfis ativos simultaneamente e *flows* de *backup* concentrados entre t=10s e t=20s para criar desequilíbrio pontual, produziu os maiores valores absolutos de *throughput* médio entre os cenários de carga complexa. O CS registrou 1,41 Mbps, o RR atingiu 1,92 Mbps e o LL alcançou 1,93 Mbps, diferença mínima entre os dois de apenas 0,01, o ML obteve 1,84 Mbps, ficando 0,09 Mbps abaixo do LL. A proximidade entre RR e LL no C3 sugere que a heterogeneidade simultânea dos perfis de tráfego com *flows web, voip, video* e *backup* chegando ao mesmo tempo produziu uma distribuição naturalmente equilibrada quando o RR atribui conexões ciclicamente, reduzindo o espaço de melhoria disponível para o algoritmo baseado em carga. O desequilíbrio pontual criado pelos *backups* concentrados é suficientemente breve para que o LL não consiga capitalizá-lo de forma expressiva dentro do seu ciclo de coleta de 2 segundos.

O cenário de carga sustentada, com chegada contínua de novos *flows* a cada 3 segundos ao longo de toda a execução, produziu os maiores valores absolutos de *throughput* médio de todos os cenários para todos os controladores, reflexo do perfil de tráfego predominantemente leve com chegada espaçada. O CS registrou 4,19 Mbps, o RR 5,00 Mbps, o ML 4,89 Mbps e o LL 5,06 Mbps. A chegada regular e espaçada de novos *flows* é o ambiente mais favorável para o LL: a cada 3 segundos um novo *flow* chega, e com ciclo de coleta de 2 segundos o controlador possui medições recentes e confiáveis da carga de cada servidor no momento de cada decisão. Apesar disso, a vantagem do LL sobre o RR é modesta 0,06 Mbps

indicando que a chegada espaçada e predominantemente leve do C4 não gera desequilíbrios severos o suficiente para que a informação de carga produza decisões expressivamente superiores à distribuição cíclica. O ML, com 4,89 Mbps, ficou 0,11 Mbps abaixo do LL, apresentando sua maior defasagem relativa em termos absolutos neste cenário, embora em termos percentuais a diferença seja pequena dado o nível elevado de *throughput* médio de todos os controladores.

O CS confirmou em todos os cenários sua posição como referência inferior do experimento. A ausência de qualquer mecanismo de balanceamento ativo resultou consistentemente nos menores valores de *throughput* médio, pois em redes SDN com múltiplos servidores e tráfego heterogêneo, a distribuição de carga não emerge naturalmente da arquitetura da rede e exige intervenção ativa do plano de controle. Qualquer estratégia de balanceamento, mesmo a mais simples, mostrou-se superior ao encaminhamento sem critério.

O LL obteve os melhores resultados em três dos quatro cenários avaliados, consolidando-se como o controlador de melhor desempenho geral. No entanto, sua vantagem sobre o RR foi modesta em todos os cenários. O RR, apesar de ser a estratégia de balanceamento mais simples possível, mostrou-se uma alternativa competitiva e consistente, ficando muito próximo do LL na maioria dos cenários. Isso indica que, para o perfil de tráfego e a topologia avaliados, grande parte do ganho do balanceamento já é capturado pela simples distribuição uniforme de conexões mesmo em tráfegos não tão uniformes.

O ML acompanhou as demais opções com balanceamento em todos os cenários, mas não conseguiu superar o LL em nenhum deles, ficando em geral próximo ao RR. Esse resultado sugere que o modelo, em sua configuração atual, não aproveitou plenamente o potencial do ML para o problema de balanceamento. Duas limitações se destacam como as mais prováveis para esse comportamento: a primeira é que o conjunto de dados de treinamento foi gerado exclusivamente em sessões controladas pelo LL, o que significa que o modelo aprendeu a partir de uma rede consistentemente bem balanceada, sem exposição a estados de desequilíbrio mais severos que poderiam revelar decisões superiores às do LL. A segunda é que os

pesos da pontuação composta utilizado para geração dos rótulos (definidos empíricamente como 0,35 para o *Jain Index*, 0,25 para a Entropia, 0,25 para a Estabilidade e 0,15 para o *Headroom*) podem não refletir de forma ótima a função objetivo real do balanceamento nos cenários avaliados, e um ajuste fino desses coeficientes poderia produzir rótulos de melhor qualidade e, consequentemente, um modelo mais eficaz.

5. Conclusão

Este trabalho teve como objetivo avaliar comparativamente quatro estratégias de balanceamento de carga em SDN, o CS, o RR, o LL e um controlador baseado em ML sobre uma topologia simulada no *Mininet*. Os experimentos foram conduzidos em quatro cenários de tráfego distintos, cada um projetado para expor características específicas do problema de balanceamento.

Os resultados obtidos confirmaram a premissa central do trabalho: em redes SDN com múltiplos servidores e tráfego heterogêneo, o balanceamento de carga ativo é indispensável. O CS, que opera sem nenhum mecanismo de distribuição, registrou consistentemente os piores resultados em todos os cenários avaliados, demonstrando que a concentração de carga em um único servidor compromete de forma significativa o aproveitamento dos recursos disponíveis. Qualquer estratégia de balanceamento, independentemente de sua complexidade, mostrou-se superior ao encaminhamento sem critério.

Entre os controladores com balanceamento, o LL consolidou-se como o de melhor desempenho geral, obtendo os melhores resultados em três dos quatro cenários avaliados com uma diferença modesta, o que revelou um resultado igualmente relevante: o RR, apesar de ser a estratégia de balanceamento mais simples possível, mostrou-se uma alternativa competitiva e consistente em todos os cenários. Isso indica que, para o perfil de tráfego e a topologia avaliados, grande parte do ganho do balanceamento já é capturado pela simples distribuição uniforme de conexões. O cenário de *hotspot* ilustrou com clareza o limite do LL: quando dez *flows bulk* são disparados simultaneamente antes que qualquer medição diferencial de carga seja possível, o algoritmo baseado em carga perde sua vantagem informacional e empata com o RR.

O ML acompanhou as demais opções com balanceamento em todos os cenários, mas não conseguiu superar o LL em nenhum deles, posicionando-se em geral próximo ao RR. Esse resultado aponta limitações concretas na configuração adotada. O conjunto de dados de treinamento, gerado exclusivamente em sessões

controladas pelo LL, expôs o modelo apenas a estados de rede consistentemente bem balanceados, restringindo sua capacidade de aprender decisões superiores às do LL em situações de desequilíbrio mais severo. Adicionalmente, os pesos da pontuação composta utilizada para geração dos rótulos foram definidos empiricamente, sem otimização formal, o que pode ter limitado a qualidade do sinal de supervisão fornecido ao modelo durante o treinamento.

Em síntese, os resultados deste trabalho demonstram que redes SDN se beneficiam significativamente do balanceamento de carga ativo, que algoritmos simples como o RR já capturam grande parte desse benefício.

6. Trabalhos Futuros

Como continuidade desse projeto, é proposto o aprimoramento da base de dados utilizada no treinamento do modelo de ML, visando mitigar possíveis vieses. Uma modificação relevante consiste na coleta de dados a partir de múltiplos controladores de referência além do LL, como o RR e abordagens puramente aleatórias. Esta estratégia permitirá que o modelo seja exposto a cenários de desequilíbrio severo, conferindo-lhe maior capacidade de resposta diante de anomalias ou condições críticas de carga na rede.

Podem-se também explorar técnicas de perturbação durante a fase de coleta. Ao introduzir decisões aleatórias de forma controlada, é possível forçar a ocorrência de estados de rede aleatórios, o que pode aprimorar o treinamento supervisionado. Todos os experimentos foram conduzidos em uma topologia fixa de 3 *switches core* (conexão entre os demais *switches* e servidores), 10 *switches* de acesso, 100 hosts e 5 servidores. Avaliar o comportamento dos controladores em topologias com maior número de servidores, diferentes relações de gargalo permitiria verificar se os ganhos observados são generalizados ou específicos da configuração utilizada.

Por fim, a validação dos resultados em hardware real ou em plataformas de emulação de maior fidelidade permitiria confirmar se os ganhos observados no *Mininet* se reproduzem sob condições de latência, *jitter* e variação de *throughput* mais realistas. O *Mininet*, embora amplamente utilizado em pesquisa, compartilha recursos de CPU entre os nós simulados, o que pode influenciar as medições de *throughput* em cenários de alta carga simultânea.

7. Referências

[1]. Brasil conecta 6,1 milhões de novos usuários à internet em apenas dois anos, aponta IBGE. Disponível em: <https://www.gov.br/mcom/pt-br/noticias/2025/Julho/brasil-conecta-6-1-milhoes-de-novos-usuarios-a-internet-em-apenas-dois-anos-aponta-ibge>.

[2]. XIA, W. et al. A Survey on Software-Defined Networking. IEEE Communications Surveys & Tutorials, v. 17, n. 1, p. 27–51, 2015.

[3]. MININET PROJECT. *Mininet: An Instant Virtual Network on your Laptop (or other PC)*. GitHub, 2026. Disponível em: <https://github.com/mininet/mininet>. Acesso em: 5 mar. 2026.

[4]. SILVER, David et al. Mastering the game of Go with deep neural networks and tree search. nature, v. 529, n. 7587, p. 484-489, 2016.

[5]. QUINLAN, J.. Ross . Induction of decision trees. Machine learning, v. 1, n. 1, p. 81-106, 1986.

[6]. BREIMAN, Leo. Bagging predictors. Machine learning, v. 24, n. 2, p. 123-140, 1996.

[7]. BREIMAN, Leo. Random forests. Machine learning, v. 45, n. 1, p. 5-32, 2001

[8]. KREUTZ, Diego et al. Software-defined networking: A comprehensive survey. Proceedings of the IEEE, v. 103, n. 1, p. 14-76, 2014.

[9]. HU, Fei; HAO, Qi; BAO, Ke. A survey on software-defined network and openflow: From concept to implementation. IEEE Communications Surveys & Tutorials, v. 16, n. 4, p. 2181-2206, 2014.

[10]. ZHU, Liehuang et al. SDN controllers: A comprehensive analysis and performance evaluation study. ACM Computing Surveys (CSUR), v. 53, n. 6, p. 1-40, 2020.

[11].BELGAUM, Mohammad Riyaz et al. A systematic review of load balancing techniques in software-defined networking. Ieee Access, v. 8, p. 98612-98636, 2020.

[12].JAIN, Rajendra K. et al. A quantitative measure of fairness and discrimination. Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA, v. 21, n. 1, p. 2022-2023, 1984.

APÊNDICE A – Ambiente de execução e arquivos do projeto

Este apêndice tem como objetivo apresentar os códigos desenvolvidos neste projeto, bem como os recursos necessários para a reprodução dos experimentos realizados. A disponibilização desses materiais visa garantir a reprodutibilidade dos resultados e a transparência metodológica.

O acesso aos arquivos pode ser realizado por meio dos seguintes links:

- **VM do projeto:** <https://drive.google.com/drive/folders/1AfgTVdQWhTVMXCYU-wQl4rLDH--Fl6pQ.>
- **Repositório dos códigos:** <https://github.com/BBberto/TG---INFO/>