DISSERTAÇÃO DE MESTRADO

Detecção de tráfego anômalo de rede
utilizando clusterização em Big Data

Mateus Almeida Rocha

Brasília, Dezembro de 2020

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

Detecção de tráfego anômalo de rede utilizando clusterização em Big Data

Mateus Almeida Rocha

Dissertação de Mestrado submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Daniel Guerreiro e Silva, UnB _____
Orientador

Prof. Georges Daniel Amvame Nze, UnB _____
Examinador Interno

Prof. Anderson Clayton Alves Nascimento, UW _____
Examinador Externo

Publicação: PPGENE.DM-758/20
Brasília, 14 de Dezembro de 2020

FICHA CATALOGRÁFICA

ROCHA, MATEUS ALMEIDA

Detecção de tráfego anômalo de rede utilizando clusterização em Big Data [Distrito Federal] 2020.

xvi, 93 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2020).

Dissertação de Mestrado - Universidade de Brasília, Faculdade de Tecnologia.

Departamento de Engenharia Elétrica

1. Big Data	2. k-prototypes
3. clusterização	4. detecção de intrusão
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ROCHA M. A. (2020). *Detecção de tráfego anômalo de rede utilizando clusterização em Big Data*. Dissertação de Mestrado, Publicação PPGENE.DM-758/20, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 93 p.

CESSÃO DE DIREITOS

AUTOR: Mateus Almeida Rocha

TÍTULO: Detecção de tráfego anômalo de rede utilizando clusterização em Big Data.

GRAU: Mestre em Engenharia Elétrica ANO: 2020

_____

Mateus Almeida Rocha

Depto. de Engenharia Elétrica (ENE) - FT

Universidade de Brasília (UnB)

Campus Darcy Ribeiro

CEP 70919-970 - Brasília - DF - Brasil

Agradecimentos

Agradeço ao Laboratório Latitude pela gentileza ao ceder a infraestrutura necessária para a correta implementação deste trabalho. Ademais, sou grato aos meus amigos pelo apoio e também pelas inúmeras vezes em que me ajudaram a encontrar soluções para problemas de implementação, mesmo quando pareciam impossíveis.

Por fim, direciono minha gratidão à minha família e amigos do peito, não somente pelos conselhos e ensinamentos passados ao longo de nossa jornada juntos, mas também por acreditarem no meu potencial de maneira tão otimista e me incentivarem a ser cada dia melhor tanto como pessoa quanto profissional.

Mateus Almeida Rocha

RESUMO

Na sociedade atual, o volume de informações trafegadas pela Internet atinge patamares cada vez maiores. Todo o tipo de mensagens são trocadas na grande rede de computadores, incluindo dados confidenciais e sigilosos. Esse cenário atrai atividades de criminosos virtuais, que utilizam os mais diversos tipos de vulnerabilidades para roubar e manipular informações para interesse próprio. Diversas respostas podem ser utilizadas para tentar combater essa ameaça virtual, sendo uma delas utilizar técnicas de aprendizado de máquina para detectar tráfego potencialmente malicioso e tomar as devidas precauções. Devido à grande quantidade de informação disponível para se trabalhar, é necessário utilizar ferramentas de Big Data, para tratá-las de maneira eficiente.

Propõe-se então uma arquitetura em ferramentas de Big Data para detectar anomalias no tráfego de rede utilizando clusterização. Também é proposto que ela seja capaz de realizar o processamento do tráfego em tempo quase real, e que seja capaz de manter os modelos gerados atualizados de acordo com a variação da rede, através de retreinos periódicos.

Na metodologia, é explicado os componentes da arquitetura proposta e como ocorre a comunicação entre eles. Nos experimentos, são testados dois cenários distintos: primeiramente, é feita uma análise *offline* com um *dataset* disponível na plataforma Kaggle, sendo seguido por uma análise *online* no servidor de *proxy*-reverso do laboratório Latitude. Por fim, são analisados os resultados coletados e métricas como acurácia, precisão, especificidade e tempo de predição são comparadas com outros trabalhos que possuem proposta similar.

ABSTRACT

In modern society, the sheer volume of information sent through the Internet grows by the day. Many types of messages are exchanged on the Internet, including confidential and secret data. This scenario attracts the activity of cybercriminals, who can exploit vulnerabilities in order to manipulate and steal information for personal gain. Many answers arise from the growing virtual menace, including the use of machine learning techniques to detect potentially malicious traffic, which would give means to take the proper precautions. With the massive amount of data available nowadays, the use of Big Data frameworks becomes a necessity, to process it efficiently.

We then propose an architecture using Big Data frameworks capable of detecting anomalies in network traffic using clustering techniques. It is also proposed that it should be able to process network traffic in near real time, and that the generated models should be kept updated according to the variation of network, through periodic retrains.

In out methodology, we explain the components of the proposed architecture and how they communicate with each other. In the experiments, two different scenarios are tested: first, an offline analysis is performed with a dataset available on Kaggle, followed by an online analysis on the reverse proxy server from Latitude, a laboratory from the Electrical Engineering Department. Finally, the collected results are analyzed and metrics such as accuracy, precision, specificity and prediction times are compared with other papers with similar proposals.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Com o advento da Internet e o avanço das redes de comunicações na sociedade moderna, cada vez mais indivíduos encontram-se conectados e usufruindo de serviços presentes na rede de computadores. Porém, uma parcela considerável desses usuários é leiga no assunto, não conhecendo sobre ataques virtuais, como é arquitetada a Internet e nem sobre segurança de redes. Além da grande quantidade de pessoas conectadas à rede de computadores, a crescente modernização dos ataques virtuais causou a evolução das ferramentas computacionais de ataque, tornando-as mais acessíveis, sofisticando suas sintaxes e automatizando a descoberta de vulnerabilidades. Essas características formam um cenário deveras favorável para a ascensão de ataques virtuais, como mostrado no relatório de segurança de 2020 da *National Technology Security Coalition* [12].

Neste relatório, a NTSC traça uma linha do tempo com os principais eventos que aconteceram no mundo de segurança cibernética durante 2019. Vítimas de todos os status sociais foram afetadas por ataques virtuais, incluindo pessoas físicas, empresas, *websites* e até mesmo governos foram alvos de chantagens, vazamentos de informações, indisponibilidade de serviço e fraudes.

Porém, há um grande destaque aos eventos que já aconteceram no ano de 2020 também. Segundo este relatório, o principal alvo dos ataques cibernéticos atualmente são redes privadas de companhias, onde os atacantes tentam conseguir acesso ilícito para lucrar, seja vendendo essas informações no mercado informal, explorando vulnerabilidades encontradas durante o processo de invasão ou até mesmo através de chantagens.

Percebe-se que a invasão de ambientes virtuais é um problema sério no cenário atual. Dessa forma, faz-se necessário a utilização de Sistemas de Detecção de Intrusão, principalmente dentro de redes privadas pertencentes a grandes companhias.

Para se construir Sistemas de Detecção de Intrusão robustos, capazes de se identificar ameaças virtuais conhecidas ou até mesmo novas, é necessário que grandes volumes de tráfego de rede sejam analisados para conseguir gerar modelos de normalidade de uma rede e, com isso, ser capaz de detectar quaisquer discrepâncias apresentadas no tráfego de rede. Ademais é comum que a massa de dados disponível para treinar o modelo seja não-rotulada, pois o processo de curadoria de dados de rede é custoso e demandante.

Para processar o tráfego de rede, faz-se necessário utilizar ferramentas de *Big Data*, visto que elas são capazes de processar grandes volumes de informação de maneira eficiente e distribuída, permitindo escalar horizontalmente os recursos necessários para processar grandes volumes de dados, ao invés de utilizar computadores robustos.

Porém, mesmo com o uso das ferramentas de *Big Data*, ainda se tem como desafio processar grandes volumes de tráfego de rede em tempo hábil, para que o sistema seja capaz de notificar um administrador de rede no momento em que um ataque esteja acontecendo. Caso contrário, é possível que o ataque somente seja percebido depois que sistemas importantes foram comprometidos.

1.1 OBJETIVOS

O objetivo primário deste trabalho é construir uma arquitetura em *Big Data* que seja capaz de analisar tráfego de rede em regime online para detectar quaisquer padrão de anomalia em um determinado servidor. Para tal, propõe-se utilizar um algoritmo não-supervisionado de aprendizado de máquina que seja capaz de processar o tráfego não categorizado que é recebido nas interfaces de rede.

Com o intuito de impedir que o modelo de detecção de anomalias fique obsoleto devido às contantes mudanças na rede, propõe-se também que a arquitetura proposta seja capaz de gerar novos modelos periodicamente e sempre utilize o último gerado para realizar suas predições.

Para validar a arquitetura montada, foram utilizadas duas fontes distintas de dados. O primeiro *dataset*, disponível para o público através da plataforma *Kaggle* possui informações sobre ataques cibernéticos feitos em uma rede simulada das forças aéreas dos Estados Unidos; O segundo foi coletado em uma *Honeynet* montada no laboratório LATITUDE, pertencente ao Departamento de Engenharia Elétrica da Universidade de Brasília.

1.1.1 Objetivos específicos

- Construir uma arquitetura em *Big Data* capaz de detectar anomalias de rede.
- Processar dados não-rotulados.

- Processar tanto atributos numéricos quanto categóricos.
- Processar o tráfego em *near real time*.
- Manter o sistema atualizado através de retreinos periódicos do modelo utilizado para detecção de anomalias.

1.2 ORGANIZAÇÃO DO TRABALHO

O restante deste trabalho está assim organizado: o capítulo 2 provê conceitos necessários sobre redes de computadores, *Honeynets*, ferramentas de Big Data, detecção de intrusão e o algoritmo de clusterização K-prototypes; O capítulo 3 explica detalhadamente as ferramentas utilizadas para a implantação da arquitetura proposta e os componentes que foram desenvolvidos; O capítulo 4 ilustra os experimentos que foram conduzidos e os resultados coletados, além de comparar as métricas calculadas com outros trabalhos similares; Por fim, o capítulo 5 tece as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo fará uma breve explicação sobre os conceitos necessários para o entendimento da metodologia implementada e dos experimentos realizados. Primeiramente, são abordados conceitos fundamentais de redes de computadores, englobando a divisão lógica da rede, os protocolos analisados e os cabeçalhos de tais protocolos. A seguir, explica-se o funcionamento de *Honeynets*, das ferramentas de *Big Data* e as técnicas para detecção de intrusão em redes de computadores. Por fim, aborda-se o algoritmo de clusterização K-prototypes.

2.1 DIVISÃO EM CAMADAS

A crescente complexidade da Internet tornou a sua divisão em camadas lógicas praticamente inevitável [13]. A tarefa de possibilitar a comunicação entre aplicações localizadas em computadores diferentes (na maioria dos casos separados geograficamente) não é simples, e tentar resolver essa tarefa com um único protocolo o tornaria extremamente complexo e muito suscetível à falhas. O processo de comunicação entre duas máquinas envolve múltiplos elementos trabalhando em harmonia: sistemas operacionais, roteadores, *switches*, *Firewalls*, infraestruturas de telecomunicações e diferentes meios físicos. Tentar analisar, entender e, principalmente, protocolar todos esses elementos ao mesmo tempo seria difícil para a mente humana.

Dessa forma, dividir a Internet em diferentes camadas, cada uma delas focando em funções específicas, provê um conjunto de ferramentas que facilita o desenvolvimento, a análise e o aprendizado dos conceitos de maneira mais eficiente. Com essa abordagem, cada camada individual deve se preocupar apenas em prover serviços para a camada imediatamente acima e solicitar outros serviços da camada imediatamente abaixo.

Outro principal motivo para a separação da Internet em camadas diz respeito aos grandes fabricantes de equipamentos de redes. Não seria prático tentar montar uma rede utilizando exclusivamente equipamentos de um mesmo fabricante. A consistência provida pela padronização e por protocolos bem definidos permite que equipamentos construídos para trabalhar em camadas específicas possam ser substituídos por outros de diferentes marcas, porém com a mesma finalidade, sem alterar o funcionamento das outras camadas.

2.1.1 O modelo de padronização OSI

O modelo OSI (acrônimo para *Open Systems Interconnection*) foi desenvolvido pela Organização Internacional de Normalização (*International Organization for Standards*) ao reconhecer a necessidade de padronizar a arquitetura da Internet em camadas. Esse modelo é ilustrado na figura 2.1.

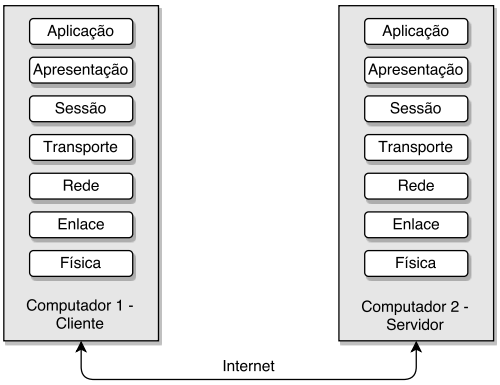

Figura 2.1: Representação simplificada do modelo OSI [Fonte: Autor].

Conforme mostrado na figura 2.1, o modelo OSI engloba todos os elementos da comunicação, desde o nível mais elevado (a camada de aplicação) até o nível mais baixo (a camada física). As camadas do modelo OSI, de baixo para cima, são: física, enlace, rede, transporte, sessão, apresentação e aplicação. A figura 2.1 mostra duas pilhas em paralelo, para representar dois sistemas computacionais diferentes ao estabelecer uma conexão.

Um exemplo mais realista de uma rede local, onde diversas unidades computacionais podem se comunicar, é mostrado na figura 2.2:

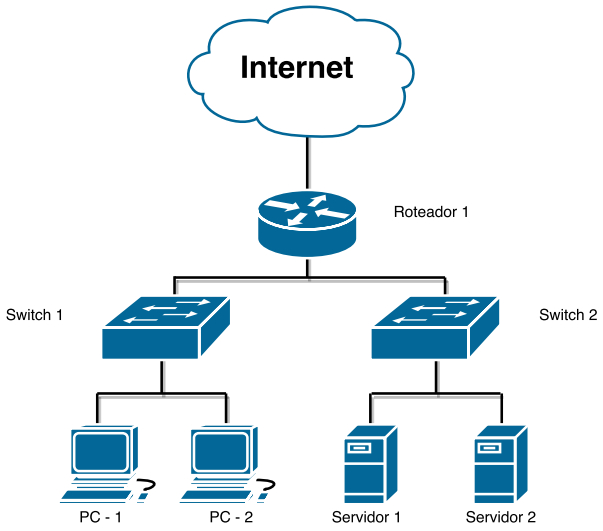

Figura 2.2: Arquitetura simplificada de uma rede [Fonte: Autor].

Na topologia ilustrada na figura 2.2, todo o tráfego de rede trocado com a Internet é roteado pelo Roteador 1. Há ainda, duas sub-redes, cada uma separada por um *switch* diferente. Esses aparelhos são responsáveis por segmentar os domínios de rede e somente se comunicam graças ao roteador. Dessa forma, os computadores 1 e 2 são capazes de comunicar com os servidores 1 e 2 somente se houver rotas no roteador que permitam essa comunicação.

Apenas as camadas do modelo OSI que estão envolvidas com a proposta deste trabalho são explicadas nesta sessão.

2.1.2 A camada de rede

A camada de rede possui a responsabilidade de mover informações de rede em rede, até que o tráfego de origem chegue ao seu destino [13]. Portanto, é possível afirmar que ela provê as funcionalidades necessárias para tornar a comunicação fim-a-fim das camadas superiores possível.

A unidade de dados desta camada é chamada de pacote. Para permitir que computadores conectados a redes diferentes sejam capazes de trocar pacotes entre si, a rede possui um mecanismo de endereçamento único e, com base nessa informação, consegue determinar a melhor rota dentre todas as disponíveis (com base em diferentes tipos de métricas). O protocolo de rede mais utilizado na atualidade é o *Internet Protocol*, ou IP [14]. Ele é um padrão criado para rotear pacotes entre diferentes redes, e é alocado, de maneira lógica, para cada interface de um aparelho capaz de conectar à Internet.

Por fim, o aspecto mais importante de um pacote IP é seu endereçamento: ele sempre possui um endereço atribuído para a origem, que identifica quem gerou o pacote, e outro para o destino, que pertence à interface onde aquele pacote deve ser entregue [15].

2.1.3 A camada de transporte

A principal função da camada de transporte é possibilitar que processos rodando em computadores distintos se comuniquem de maneira transparente. Para tal, essa camada utiliza todos os serviços oferecidos pela camada de rede, mostrados na sessão 2.1.2. A camada de transporte somente é capaz de prover uma comunicação fim-a-fim pois é a primeira no modelo OSI a verdadeiramente estabelecer conexões entre os sistemas finais [16]. Essa afirmação é ilustrada na figura 2.3:

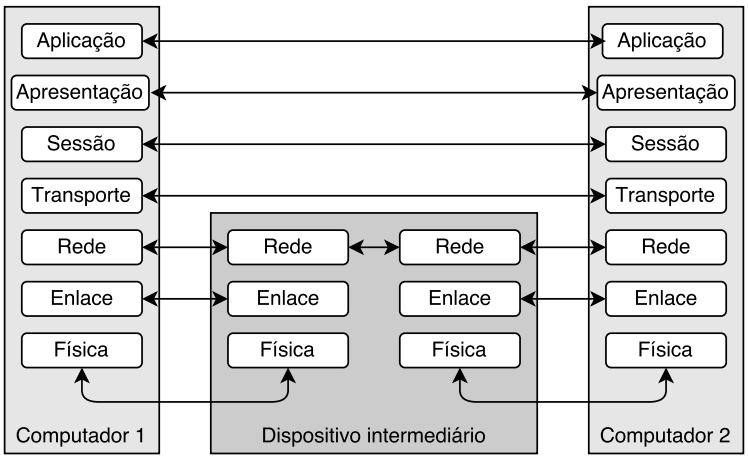

Figura 2.3: Representação da comunicação fim a fim na camada de transporte [Fonte: Autor].

A partir da análise da figura, percebe-se que a camada de transporte é a primeira a criar uma conexão lógica com o destino final, ao passo que as camadas inferiores preocupam-se apenas com o próximo salto na comunicação. Essa característica faz com que a camada de transporte seja capaz de prover comunicação lógica entre os processos que rodam em ambas as máquinas [17]. Apesar dos processos não estarem conectados fisicamente, do ponto de vista das aplicações é como se estivessem.

Nessa camada, a troca de informações não é mais visualizada como uma troca de pacotes individuais, mas sim como um fluxo de informação que segue através da rede. Utiliza-se para esse fim dois principais protocolos: o *Transmission Control Protocol* - TCP e o *User Datagram Protocol* - UDP. As aplicações podem requisitar qualquer um dos dois, a depender do tipo e qualidade do serviço que desejarem [18]. Caso necessite de uma comunicação orientada à conexão, ou quando a integridade dos dados transmitidos tenha elevada necessidade de ser mantida, o protocolo TCP é utilizado. Esse protocolo possui mecanismos para verificar se a informação recebida é precisamente a mesma que foi transmitida. Porém, se a integridade da informação não for prioridade, ou se manter uma taxa constante de dados seja mais vantajoso, escolher o protocolo UDP pode ser mais benéfico para a aplicação. A conexão estabelecida por esse protocolo não possui nenhum tipo de mecanismo de verificação da transmissão.

2.1.4 A camada de aplicação

No topo da arquitetura OSI, está a camada de aplicação. Ela não é o mais abstrato que se pode subir na pilha de camadas, pois alguns programas adicionam camadas adicionais acima das sete aqui citadas (a rede Tor, por exemplo [19]), porém é o mais distante que o modelo OSI engloba. Essa camada é utilizada por aplicações de rede e atua como intermediadora para o usuário, sendo uma interface de dados capaz de traduzir as entradas do usuário em informações transportáveis pela Internet [20].

Devido à sua natureza, a camada de aplicação é aquela que interage diretamente com o usuário final mais frequentemente. Por consequência, protocolos como o HTTP, FTP, SMTP e TELNET [21] transportam diretamente informações importantes e possivelmente sensíveis. Portanto, é comum que ataques foquem nesta camada, pois a interação humana é significativamente mais suscetível à falhas e brechas, o que pode permitir a um atacante se aproveitar para acessar patrimônios lógicos que deveriam ter acesso restrito.

2.2 OS PROTOCOLOS ANALISADOS

Nesta sessão são analisados os principais protocolos utilizados a partir da camada 3, a de rede, introduzida na seção 2.1.2. Ela foi escolhida como camada inicial principalmente pelo fato de análises profundas de pacotes tipicamente começarem por ela, pois as camadas anteriores não possuem informações significativas quanto à carga útil transportada nos pacotes [22].

2.2.1 Internet Protocol

Na atualidade, o protocolo mais utilizado na Internet é o IP. Trata-se de um protocolo versátil, pois oferece suporte a diversos protocolos de camadas superiores, tais como o TCP e o UDP. Grande parte das aplicações presentes na Internet dependem do IP, tais como navegadores, clientes FTP e agentes de email, o que aumenta ainda mais a sua importância.

O protocolo IP é descrito na RFC-791 [1], criada em 1981 pela organização de padronização IETF. Ele foi concebido para ser utilizado em redes de comutação de pacotes e transmite blocos de informação, os datagramas, entre transmissores e receptores. Cada entidade em uma rede IP deve ser unicamente identificada por um número de endereçamento de tamanho fixo, o endereço IP.

Há suporte para fragmentação e remontagem de pacotes, para que o fluxo de bytes seja capaz de trafegar continuamente em meios físicos cujo tamanho máximo dos pacotes seja diferente, sem que as entidades que estão se comunicando percebam a diferença. Não exitem, porém, mecanismos de confiabilidade, controle de fluxo ou sequenciamento. Cada datagrama é tratado como uma entidade independente e não relacionada aos outros datagramas, o que garante a não formação de nenhuma espécie de conexão ou circuito lógico. A relação do IP com outros protocolos é descrita na figura 2.4.

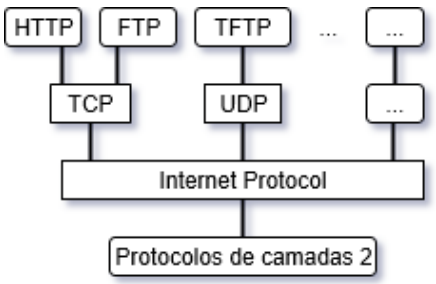

Figura 2.4: Hierarquia dos protocolos a partir do IP. [Fonte: Autor].

Na figura é possível perceber que o IP é versátil, possuindo interfaces de comunicação com protocolos distintos da camada de transporte. Para o escopo deste trabalho, porém, somente somente será estudado o protocolo TCP.

Os campos do cabeçalho presente no protocolo IP são ilustrados na figura 2.5.

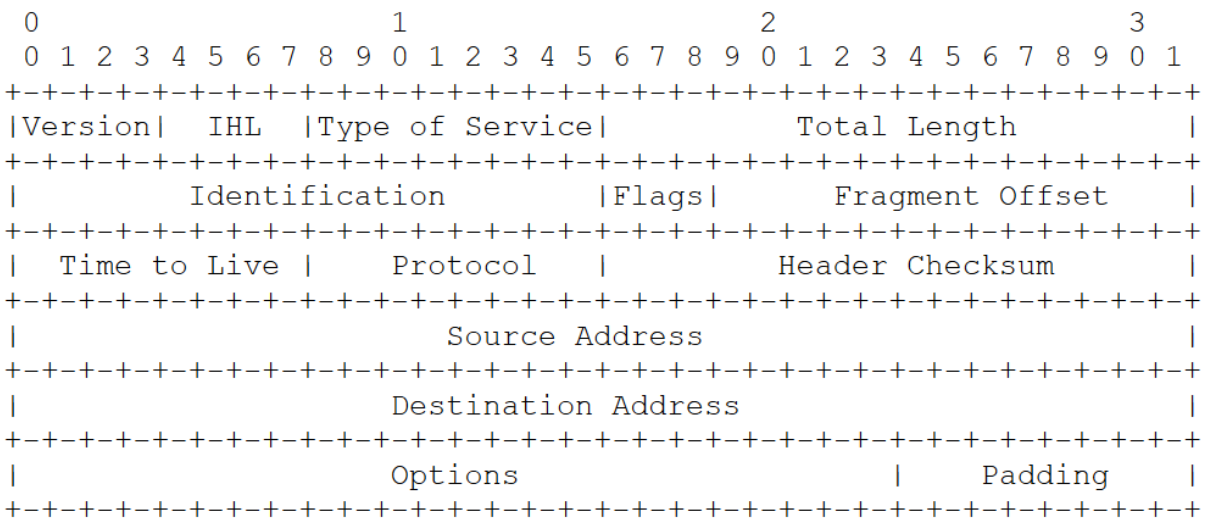

Figura 2.5: Cabeçalho do protocolo IP, de acordo com a RFC 791. [1].

Na figura 2.5, cada sinal de adição representa um bit. Os campos representados são descritos a seguir:

- Destination Address: 32 bits

  Endereço IP do destinatário.

- Time to Live: 8 bits .

  Indica a versão do protocolo IP que o pacote está seguindo.

- IHL: 4 bits.

  Tamanho do cabeçalho de Internet, medido em palavras de 32 bits. Ou seja, aponta para onde as informações úteis do pacote começam.

- Type of Service: 8 bits

  Provê uma indicação da qualidade de serviço desejada, para o caso da rede possuir priorização de tráfego.

  A disposição dos bits é:

  - Bits 0-2: Precedência.
  - Bit 3: 0 = Atraso normal, 1 = Atraso baixo.
  - Bit 4: 0 = *Throughput* normal, 1 = *Throughput* alto.
  - Bit 5: 0 = Confiabilidade normal, 1 = Confiabilidade alta.
  - Bits 6-7: Reservado para usos futuros.

  As precedências podem ser:

  - 111 - Network Control
  - 110 - Internetwork Control
  - 101 - CRITIC/ECP
  - 100 - Flash Override
  - 011 - Flash
  - 010 - Immediate
  - 001 - Priority
  - 000 - Routine

  Vale ressaltar que o uso dos parâmetros de precedência no pacote pode afetar o custo do serviço de transporte de datagramas, pois priorizar parte do tráfego significa despriorizar a parte remanescente.

- Protocol: 8 bits

  É responsável por mostrar qual o protocolo utilizado na camada superior.

- Version: 4 bits

  Valor utilizado pelo transmissor para facilitar a reconstrução do datagrama, caso ele seja fragmentado.

- Flags: 3 bits

  Algumas *flags* de controle:

  - 0: reservado, deve ser zero.
  - 1: (DF) 0 = pode fragmentar, 1 = não fragmente.

  - 2: (MF) 0 = último fragmento, 1 = há mais fragmentos.

- Fragment Offset: 13 bits

  Indica a posição do fragmento no datagrama original.

  Indica o tempo máximo que o pacote deve sobreviver na rede.

- Identification: 16 bits

- Total Length: 16 bits

  Indica o tamanho total do datagrama, medido em octetos. O valor máximo permitido neste campo é de 65.535 octetos, apesar dele ser impraticável para aparelhos menos robustos.

- Header Checksum: 16 bits

  É o *checksum* do cabeçalho. Deve ser recalculado a cada salto da rede, pois alguns dos campos no cabeçalho mudam (Time to Live, por exemplo).

  Endereço IP do remetente.

- Source Address: 32 bits

- Options: variável

  Datagramas podem ou não possuir campos opcionais. Entretanto, esses campos devem estar implementados em todos os equipamentos capazes de processar pacotes IP.

2.2.2 Transmission Control Protocol

O TCP é descrito na RFC 793 [2] e foi desenvolvido para ser utilizado como um protocolo altamente confiável para transmissões fim-a-fim em redes de comutação de pacotes. Para prover tal função, ele é orientado a conexão, e possui suporte a diversos outros protocolos das camadas adjacentes. O TCP assume pouca confiabilidade sendo prestada pelos serviços das camadas inferiores, estando preparado para lidar com o serviço de melhor esforço prestado pelo IP, por exemplo. Devido à sua natureza conservadora, o TCP é capaz de funcionar corretamente tanto em redes de comutação de circuitos quanto em comutação de pacotes.

Para prover serviços confiáveis, o TCP possui os seguintes mecanismos:

- Confiabilidade:

O TCP é capaz de se recuperar de falhas durante a comunicação, incluindo pacotes que são danificados durante o caminho, perdidos, duplicados ou entregues fora da ordem original de transmissão. Para tal, todo segmento recebe um número de sequência e requer confirmação (ACK) quando recebidos. Caso a confirmação não seja recebida pelo transmissor, o segmento é retransmitido. Os números de sequência, por sua vez, são utilizados pelo receptor para ordenar corretamente os pacotes recebidos. Os possíveis erros de transmissão são percebidos devido ao *checksum* adicionado ao pacote recebido da camada de aplicação, e sempre que um erro é detectado, uma retransmissão da parte danificada é feita.

- Segurança:

  O protocolo TCP não se preocupa com questões de segurança, deixando esse aspecto para as camadas superiores.

- Transferência de informações:

  O TCP é capaz de transmitir um fluxo contínuo de bytes em ambos os sentidos da comunicação através de segmentos.

- Multiplexação:

  Um único computador é capaz de processar múltiplas conexões TCP de diferentes aplicações simultaneamente graças ao uso de portas para identificar processos. O conjunto formado da combinação entre endereços IP e o número de porta é chamado de *socket*.

- Conexões:

  Este é o aspecto mais marcante deste protocolo. Ambos o controle de fluxo e a confiabilidade prestadas pelo TCP exigem que informações de estado sejam mantidas para cada conexão ativa. A combinação dessa informação, *sockets* e número de sequência dos pacotes é chamada de conexão. Antes que dois processos distintos efetivamente iniciem uma comunicação, os protocolos TCP em ambas as pontas devem primeiro estabelecer uma conexão. Esse fato torna o TCP um protocolo orientado à conexão.

- Controle de fluxo:

  O receptor de uma conexão TCP é capaz de regular a vazão dos dados enviados pelo transmissor com o uso das mensagens de confirmação. Caso o transmissor receba os ACKs em uma taxa menor daquela que ele está enviando o tráfego original, ele diminui a vazão de dados.

O cabeçalho do TCP é ilustrado na figura 2.6.

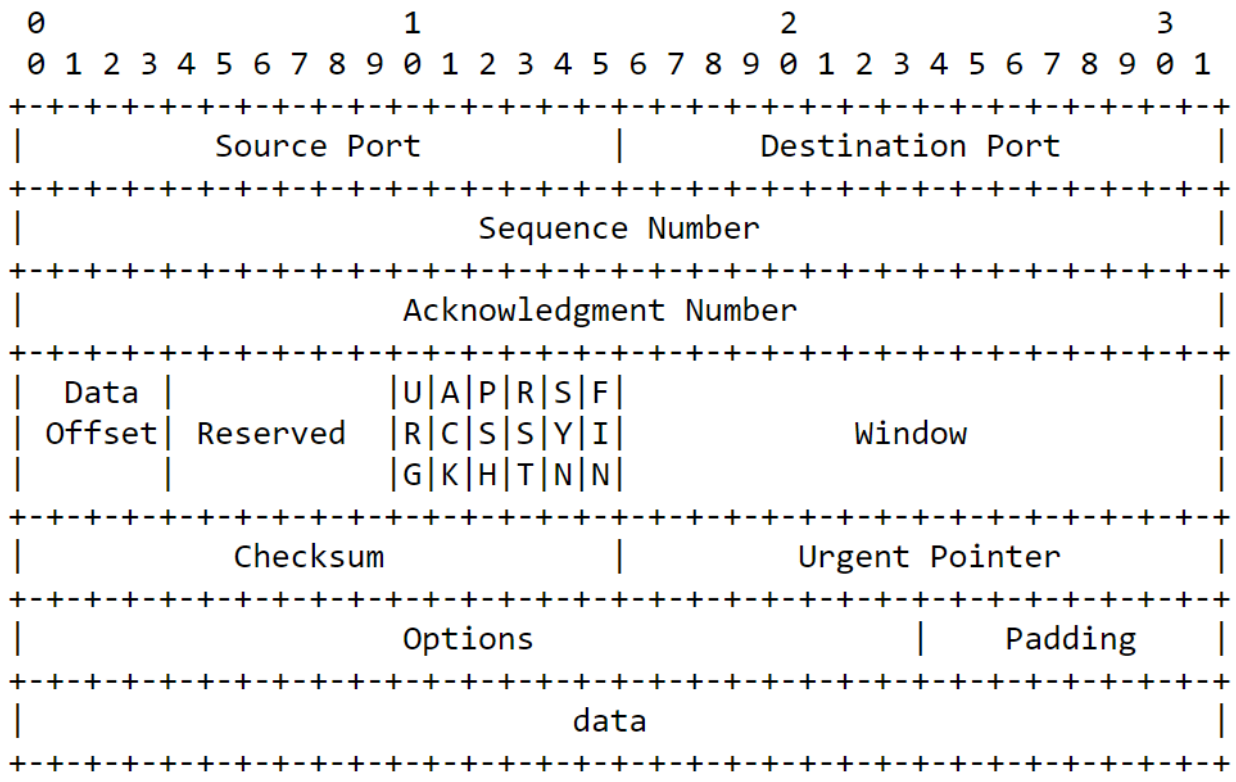

Figura 2.6: Cabeçalho do protocolo TCP, especificado na RFC 793 [2].

Na figura 2.6 é possível ver todos os campos possíveis em um cabeçalho TCP. Eles são:

- *Control Bits*: 6 bits (da esquerda para a direita).

  Também conhecidos como *flags*, são:

  - URG: *Urgent Pointer*
  - ACK: *Ackowledgment*
  - PSH: *Push Function*
  - RST: *Reset the connection*
  - SYN: *Synchronize sequence numbers*
  - FIN: *No more data from sender*

- *Source Port*: 16 bits.

  A porta utilizada para identificar o processo de origem.

- *Sequence Number*: 32 bits.

  O número de sequência consiste do primeiro octeto de informação contido no segmento. Caso a *flag* SYN esteja marcada, então esse número passa a ser o número de sequência inicial (ISN).

- *Destination Port*: 16 bits.

  Número de porta utilizado para identificar o processo receptor do segmento.

- *Acknowledgment Number*

  Caso a *flag* ACK esteja ativada, esse campo informa ao receptor qual o valor do próximo número de sequência que o transmissor espera receber.

- *Window*: 16 bits

  Também conhecido como a janela de transmissão, trata da quantidade de octetos de informação que o transmissor está disposto a aceitar em um dado momento.

- *Data Offset*: 4 bits.

  A quantidade de palavras de 32 bits que formam o cabeçalho. Efetivamente, esse número indica aonde a carga útil do segmento começa.

- *Urgent Pointer*: 16 bits

  Este campo pode conter um ponteiro que indica aonde começa a informação urgente dentro da carga útil do segmento, caso ela exista. Ele somento deve ser interpretado caso a *flag* URG esteja ativada.

- *Checksum*: 16 bits

  Código de verificação adicionado para validar a integridade da informação transportada no segmento.

- *Reserved*: 6 bits.

  Reservado para uso futuro e deve ser zero.

- *Options*: variável

  Esses parâmetros são opcionais e podem assumir tamanho variado, mas sempre múltiplo de 8 bits. As opções definidas são:

– *End of option list*: indica o fim das opções presentes no segmento. Somente deve ser utilizado caso o último octeto das opções não coincida com o final do cabeçalho.

– *No-Operation*: deve ser usado para separar opções.

– *Maximum Segment Size*: Possui 16 bits de comprimento e indica o tamanho máximo do segmento que o transmissor está disposto a aceitar. Caso ausente, o receptor está livre para utilizar qualquer tamanho de segmento.

2.2.3 HyperText Transfer Protocol - HTTP

O HTTP é um dos protocolos da camada de aplicação mais conhecidos. Seu amplo uso se dá graças à sua natureza distribuída e colaborativa, sendo o protocolo base de navegação na Internet desde a década de 1990 [3]. De maneira simplificada, o HTTP baseia-se em ambos os protocolos TCP e IP das camadas inferiores para fornecer um canal de comunicação para a troca de páginas HTML (*HyperText Markup Language*), imagens, resultados de consultas e outros.

A porta padrão utilizada por este protocolo é a 80, porém outras portas tais como 8080 e 443 também são comumente associadas a ele. Por fim, o HTTP é capaz de prover uma maneira padronizada para dispositivos computacionais se comunicarem. Essa interface de comunicação é possível graças às suas especificações em como clientes devem fazer a requisição de informações desejadas e como os servidores devem responder essas mensagens. A figura 2.7 mostra o funcionamento do protocolo HTTP:

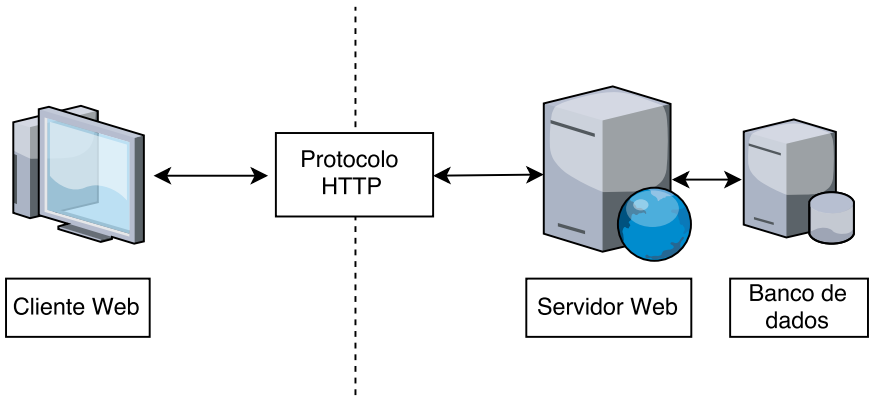

Figura 2.7: Representação do funcionamento do protocolo HTTP [Fonte: Autor].

Conforme pode ser visto, este protocolo é responsável por intermediar as mensagens trocadas por servidores e clientes, criando uma abstração de comunicação que é capaz de conectar sistemas operacionais e unidades computacionais completamente distintas. Na arquitetura do HTTP, cliente é aquele que inicia a conexão através de uma requisição a um endereço na forma de URL (*Uniform Resource Locator*), ao passo que o servidor responde essas mensagens. Comumente essas duas entidades ficam se permutando, de forma que ora uma é cliente, ora ela é servidor.

O cabeçalho da mensagem HTTP pode conter outros subcabeçalhos, que podem ser [3]: *general-header*, *request-header*, *response-header* e *entity-header*. Estes devem seguir o formato padrão de cabeçalho genérico, em que cada campo consiste de um par chave-valor separados por dois pontos (":"). A estrutura do cabeçalho é mostrada na figura 2.8:

```
cabeçalho-da-mensagem = nome-do-campo ":" [ valor-do-campo ]
    nome-do-campo     = token
    valor-do-campo    = *( conteúdo-do-campo )
    conteúdo-do-campo = <Os octetos que compõem a mensagem>
```

Figura 2.8: Estrutura básica do pacote HTTP. Adaptado de [3]

No corpo da requisição HTTP, as mensagens podem conter verbos [23]. Estes são responsáveis por indicar o tipo de ação que o requerente deseja que o servidor provenha. Os possíveis verbos que podem ser trocados são:

- POST: é utilizando quando necessita-se de criar um novo recurso no servidor. Quando esse verbo é usado, a carga útil do pacote HTTP contém as informações do elemento novo a ser criado.
- GET: é o verbo responsável por indicar que o cliente quer fazer uma busca a um elemento. Nesse caso, a URL contém todas as informações necessárias para localizar o recurso solicitado.
- DELETE: é responsável por excluir recursos existentes.
- PUT: similar ao POST, porém nesse caso o recurso é apenas atualizado, ao invés de ser criado.

Há alguns outros verbos utilizados que são menos comuns e, portanto, não farão parte do escopo deste trabalho, sendo eles: HEAD, TRACE e OPTIONS. Quando um servidor recebe um dos verbos mencionados acima, ele pode responder ao cliente diversos códigos de estado na mensagem de volta, de acordo com a completude da ação.

Os principais códigos de estado possíveis são:

- 2xx: Sucesso.

  Indica sucesso no processamento da requisição do cliente. Pode ser:

  - 200: OK. Todas as solicitações foram processadas e concluídas com êxito.
  - 202: Accepted. A requisição foi aceita, porém a resposta pode não incluir o recurso solicitado.
  - 204: No content. A resposta não possui corpo de mensagem.
  - 205: Reset content. A resposta indica ao cliente que ele deve resetar a visualização do recurso.
  - 206: Partial content. Indica que a resposta contém apenas parte do conteúdo.

- 3xx: Redireção.

  Esse código indica que alguma ação ainda é necessária da parte do cliente para a conclusão da requisição. Pode ser:

  - 301: Moved Permanently. O conteúdo solicitado foi movido permanentemente.
  - 303: See Other. O recurso encontra-se temporariamente disponível em outro endereço.
  - 304: Not Modified. O recurso não foi modificado desde a última vez que foi solicitado e, portanto, o cliente deve usar uma cópia de cache.

- 4xx: Erro do cliente.

  Ocorre um erro na hora que o servidor tenta processar a requisição, e ele assume que o cliente é responsável por isso. Pode ser:

  - 400: Bad Request. A requisição feita estava mal formada.
  - 401: Unauthorized. O recurso solicitado requer autenticação.

  - 403: Forbidden. O servidor negou o acesso ao recurso.
  - 404: Not Found. O recurso solicitado não foi encontrado.
  - 405: Method not allowed. Foi utilizado um verbo HTTP inválido.
  - 409: Conflict. Acontece quando um cliente tenta modificar um recurso que é mais recente que o horário do próprio cliente.

- 5xx: Erro do servidor.

  Indica que ocorreu um erro no servidor enquanto ele processava a requisição. Pode ser:

  - 500: Internal Server Error. Código de erro genérico.
  - 501: Not Implemented. O servidor não suporta a funcionalidade requisitada.
  - 503: Service Unavailable. O serviço está indisponível, geralmente esse erro ocorre junto com um *timeout*.

2.3 HONEYNETS

Para se compreender melhor o conceito de *honeynets*, primeiro deve-se estudar o que é um *Honeypot* [24], pois estes são a unidade fundamental que efetivamente forma a estrutura de uma *honeynet*.

2.3.1 Honeypots

Sob o contexto de segurança da informação, os *honeypots* são unidades computacionais intencionalmente deixadas vulneráveis na Internet para atrair a atenção de possíveis *hackers* e outras ameaças [24]. Ao se construir um *honeypot*, deve-se tomar o cuidado de fazê-lo parecer o máximo possível com uma vítima real, de modo que o atacante se sinta à vontade e pense que não está caindo em uma armadilha. Dessa forma, é possível registrar todas as ações tomadas e assim efetivamente analisar o comportamento de ataques virtuais.

2.3.1.1 Vantagens

Os *honeypots* são unidades intencionalmente deixadas com vulnerabilidades, porém não costumam ser divulgadas na Internet através de mecanismos de resolução de nomes, tais como o DNS. Assim, é possível concluir que quaisquer atividades externas registradas por um *honeypot* somente podem ter sido consequência de um atacante que estava ativamente procurando por vítimas na Internet. Portanto, todos os arquivos de registro gerados por essa máquina infectada possuem informações valiosas [24].

Para montar um *honeypot*, qualquer unidade computacional pode ser utilizada, de forma que nenhuma estrutura moderna ou cara é necessária para se montar uma rede dessas máquinas. A título de exemplo, em 2016 houve um ataque massivo de DDoS gerado por uma rede de dispositivos IoT infectados, e esses dispositivos são caracterizados por serem simples e não possuírem elevado poder computacional [25].

2.3.1.2 Desvantagens

Uma das maiores desvantagens de um *honeypot* também é uma de suas maiores vantagens: o fato dele não ser anunciado na Internet. De um lado, quaisquer atividades que cheguem a esta máquina podem imediatamente ser consideradas suspeitas. De outro, essa máquina é totalmente passiva ao que está acontecendo ao seu redor, e somente registrará um evento quando a comunicação estiver direcionada a ela.

2.3.1.3 Tipos

Essencialmente, há dois tipos de *honeypots*:

- Alta-interação: O segundo tipo consiste de efetivamente implantar máquinas com sistemas operacionais e aplicações reais, de modo que nada seja emulado. Dessa forma, quaisquer atacantes que por ventura invadam essa máquina, dificilmente perceberão que se trata de um ambiente artificial. Com esse ambiente ”real”, os atacantes costumam não medir esforços para tomar controle da máquina, e técnicas de invasão novas podem ser aprendidas dessa forma. Alta-interação, porém, são consideravelmente mais difíceis de implantar.

- Baixa-interação: Esses *honeypots* apenas emulam um sistema vulnerável. Apesar de serem significativamente mais simples de implementar, eles não permitem que o atacante ganhe controle da máquina. Dessa forma, é possível que o atacante perceba que está lidando com uma armadilha e aborte todas as operações. Ao fazer isso, a máquina em questão pode não registrar o ataque por completo, impedindo assim que aquele padrão de comportamento seja corretamente estudado posteriormente.

2.3.1.4 O valor agregado por *honeypots*

Honeypots possuem um amplo vetor de utilidades, podendo ser implantados tanto em organizações e empresas quanto em ambientes puramente focados em pesquisas científicas. Eles possuem o potencial de auxiliar na prevenção, detecção e eventualmente em resposta a ataques oriundos da Internet [24].

Quando se trata de prevenção, *honeypots* podem vir a retardar a cadência com que um ataque se alastra pela rede, pois todas as máquinas intencionalmente vulneráveis devem ser infectadas antes dele seguir adiante. Dessa forma, o atacante investe tempo e recursos em máquinas que não lhe trarão benefícios, pois são armadilhas, o que pode dar tempo a equipe técnica de perceber um ataque. Ademais, caso o atacante perceba a presença de *honeypots* na rede, há a possibilidade dele perder o interesse naquele ambiente.

Honeypots também são capazes de auxiliar a detecção de vulnerabilidades nunca antes vistas, antes de uma máquina importante ser comprometida. Dessa forma, com o conhecimento adquirido da armadilha, é possível reagir de maneira mais rápida e eficiente caso um ataque real aconteça.

Finalmente, além das duas vantagens citadas acima, *honeypots* também dão a oportunidade para o administrador das armadilhas de reagir a um ataque, pois as ações que ocorrem dentro da armadilha podem ser completamente isoladas e analisadas, situação que dificilmente seria possível em um ambiente de produção.

2.3.2 Honeynets

Na seção 2.3.1, foi explicado o que são *honeypots*, suas vantagens e desvantagens, seus valores e também suas aplicações. Com esses conceitos, é possível definir uma *honeynet*: trata-se de um conjunto de diferentes *honeypots*, de forma a simular um ambiente de produção interconectado por uma rede que seja realístico e convincente [26].

Por possuir uma maior variedade de programas e arquiteturas diferentes, uma *honeynet* é mais eficiente em capturar maiores diversidades de ataques de rede. Vale ressaltar que, apesar de conseguir coletar mais informações de ataques, uma maior presença de máquinas também implica em maior risco para o administrador, pois mais máquinas podem ser comprometidas a ponto de se perder o controle lógico delas. Um exemplo de *honeynet* pode ser visto na figura 2.9.

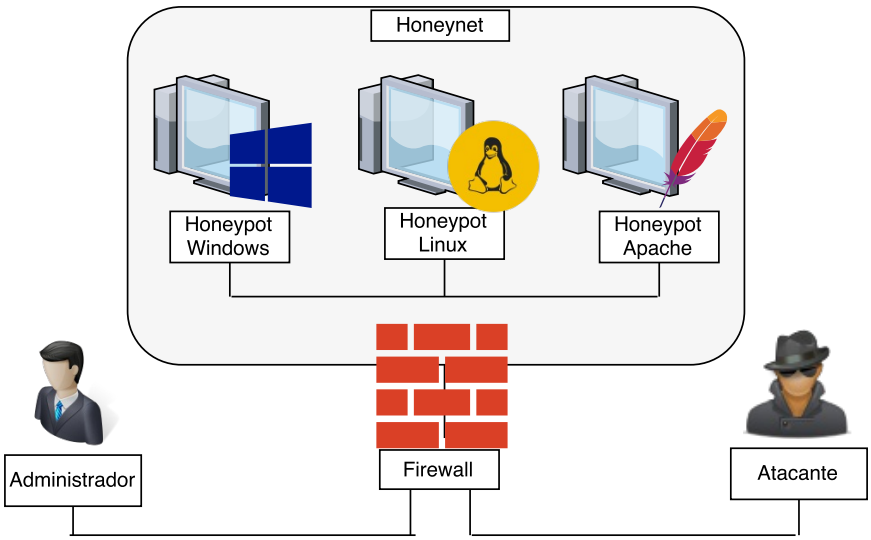

Figura 2.9: Arquitetura simplificada de uma *honeynet* [Fonte: Autor].

A figura contém uma *honeynet* composta por três sistemas distintos: um *desktop* executando *Windows*, outro *desktop* executando uma distribuição *Linux* e, por fim, um servidor *web* Apache. Essa arquitetura mostrada consiste de um ambiente heterogêneo, o que permite a coleta de informações diversificadas.

É importante que a *honeynet* esteja protegida por um *Firewall* para permitir que todo o ambiente possa ser controlado corretamente pelo administrador de rede. Dessa forma, impede-se que infecções e/ou comprometimentos de máquinas que ocorram nessa rede venham a afetar outras redes locais, tais como a rede de gerência e a rede de produção. Preferencialmente, a interface do *Firewall* conectada à *honeynet* deve estar no modo *bridge*, para evitar a adição de mais um salto na rede e assim ajudar a camuflar a presença do *firewall* para o atacante.

2.4 BIG DATA

O estudo de *Big Data* se tornou fundamental contexto atual devido ao grande volume de informações que é gerado e consumido pelos incontáveis dispositivos computacionais que formam a Internet moderna. Diariamente, petabytes de registros são coletados por empresas e instituições de ensino sobre seus usuários conectados. Informações essas que podem descrever comportamentos, interações, conexões e permitir a criação de modelos quando propriamente analisados.

O desafio desse ramo de pesquisa é justamente conseguir extrair informações úteis a partir de informações aparentemente desconexas. O ciclo de vida de um projeto de *Big Data* pode ser definido com os seguintes estágios [27]:

- Definição do problema: consiste em avaliar o problema e estimar o potencial de ganho que este pode trazer caso estudado;
- Pesquisa: procurar pelo estado da arte sobre o assunto e comparar a situação de outras entidades que lidam com o mesmo problema;
- Avaliação dos recursos humanos: consiste de verificar se a equipe é capaz de executar a tarefa com êxito ou se é necessário expandi-la;
- Captura de dados: consiste de uma das etapas mais importantes do processo e envolve pegar dados sem processamento das mais diversas fontes possíveis;
- Tratamento de dados: muitas vezes as informações coletadas não se encontram em um formato favorável à análise e, portanto, devem ser manipulados;
- Armazenamento de dados: após o processamento das informações, elas devem ser armazenadas, preferencialmente em um banco de dados que permita rápidas consultas;

- Análise exploratória de dados: é a etapa em que as informações são efetivamente estudadas e entendidas, normalmente através de estudos estatísticos e gráficos;
- Preparação para modelagem e avaliação: operações tipo normalização e dedução costumam ser feitas nessa etapa. Basicamente, qualquer tipo de processamento prévio à modelagem deve ser concluído nessa fase;
- Modelagem: com todas as informações já estudadas e remodeladas, é possível criar modelos que definam os fenômenos analisados ou que detectem padrões;
- Implementação: fazer a implantação do modelo criado na etapa anterior, monitorar seu desempenho e avaliar sua performance.

Ao analisar o ciclo de vida de uma análise em Big Data, percebe-se que grande parte dos esforços da equipe responsável pelo projeto consiste em remodelar os dados recebidos, para somente então conseguir extrair informações úteis a partir deles.

2.4.1 Sistemas de arquivos distribuídos

Para conseguir armazenar grandes arquivos no contexto de *Big Data*, uma das abordagens mais eficientes é utilizar-se de arquiteturas que combinam o poder de armazenamento de diversas unidades computacionais distintas, criando *clusters* de computadores.

Esta abordagem permite que arquivos chegando na escala dos terabytes sejam armazenados sem que *hardware* específico (com grandes quantidades de armazenamento) seja utilizado. Ao invés disso, combina-se a capacidade de armazenamento de diversos sistemas computacionais ao dividir-se o arquivo em vários blocos menores, e distribuindo esses blocos entre os computadores do grupo [28].

Uma das vantagens adquiridas ao dividir arquivos em blocos de informação e distribui-los entre vários computadores é a possibilidade de replicar esses blocos entre diferentes máquinas para diminuir a probabilidade de falha, caso algum dos nós computacionais do *cluster* venha a falhar. Porém, essa abordagem não é eficiente quando os arquivos armazenados são pequenos ou são atualizados frequentemente, devido ao alto custo de orquestração dos computadores e o *overhead* gerado toda vez que uma nova operação de escrita é recebida pelo *cluster* [29].

Uma arquitetura comum para os nós do *cluster* utilizados para armazenar os arquivos é distribuí-los em *racks* e conectar todos os nós na rede, preferencialmente com enlaces acima de gigabit *Ethernet*, como mostrado na figura 2.10:

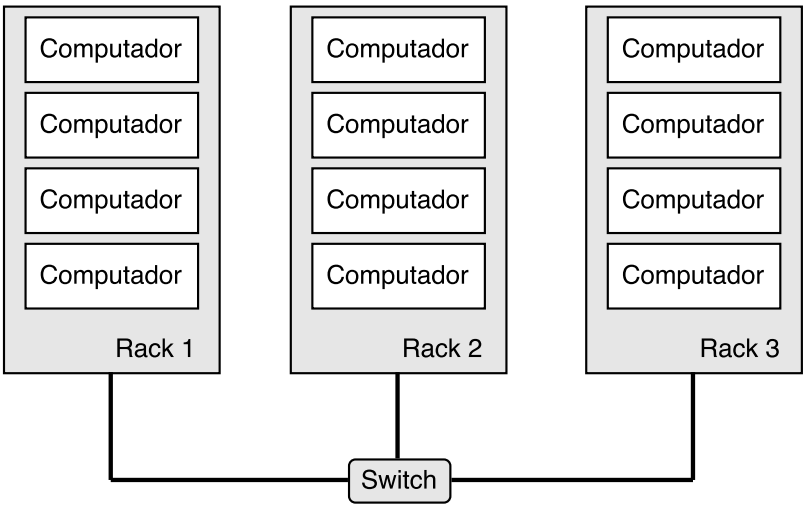

Figura 2.10: Armazenamento distribuído em um *cluster* [Fonte: Autor].

Vale ressaltar que a quantidade de computadores, racks, rede que integra as diferentes máquinas, tamanho dos blocos de armazenamento e quantidade de replicações de cada um desses blocos dentre os computadores disponíveis são todos parâmetros escolhidos pelo administrador na hora de configurar o ambiente.

2.4.2 MapReduce

Operações envolvendo *Big Data* exigem o processamento de imensas quantidades de informação de maneira rápida. Um fator que facilita esse processo é a regularidade entre as informações, o que permite o uso de técnicas de processamento em paralelo para aumentar a eficiência, sendo a mais conhecida delas o *MapReduce*.

O *MapReduce* é um estilo de computação distribuída de dados que é implementado em vários sistemas conhecidos, tais como o GDFS e o Hadoop. Essa abordagem é eficaz para fazer processamentos em larga escala, tendo tolerância à falhas de *hardware* pois utiliza sistemas de arquivos distribuídos para armazenar os dados a serem lidos. O processamento das tarefas também é feito de maneira distribuída: há a divisão da tarefa total em diversas sub-tarefas e estas são distribuídas para os dispositivos computacionais presentes no *cluster* [29].

Resumidamente, esse algoritmo é capaz de dividir tarefas complexas em partes menores, para que unidades computacionais menos robustas possam ser utilizadas para processa-las, como ilustrado na figura 2.11:

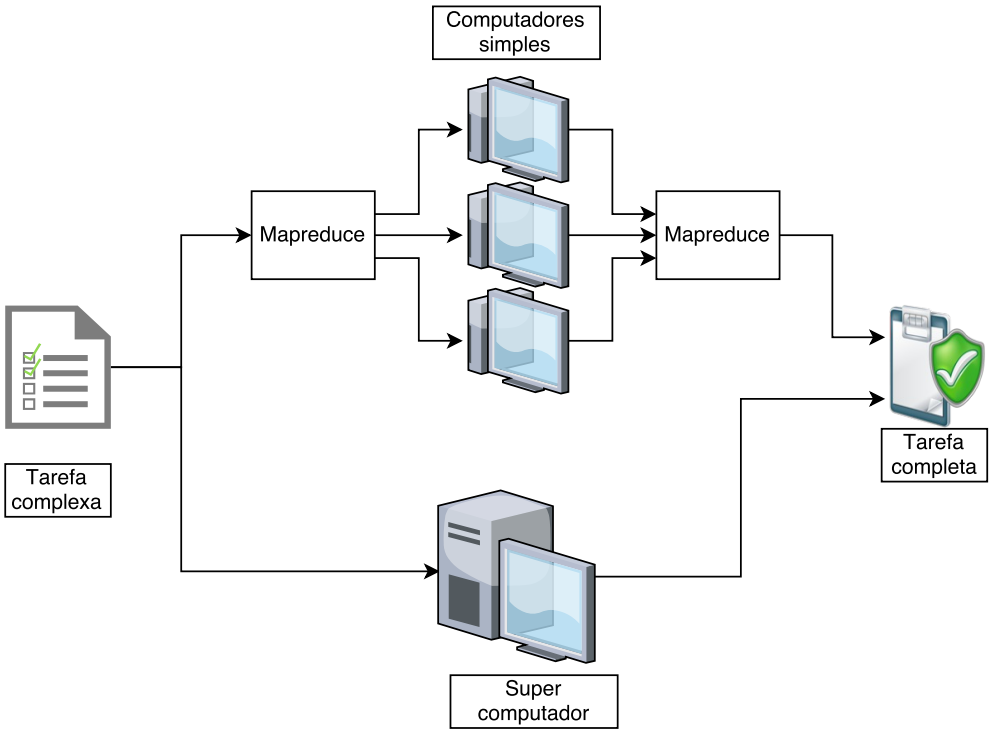

Figura 2.11: Arquitetura simplificada do *MapReduce* [Fonte: Autor].

A figura ilustra uma mesma tarefa complexa sendo executada por um super computador e por um grupo de computadores simples coordenados pelo *Mapreduce*. Para o correto funcionamento do *MapReduce*, o usuário necessita apenas escrever funções de mapeamento (*Map*) e redução (*Reduce*) (quantas forem necessárias), ao passo que o próprio sistema se encarrega de gerenciar execuções em paralelo, coordenação de tarefas e execução das funções escritas pelo usuário e recuperação de falhas.

2.4.2.1 Apache Spark

Por outro lado, o Apache Spark é um *framework* de computação distribuída utilizado majoritariamente para processamento e análise de grandes volumes de informação [30]. Similarmente ao MapReduce, ele também trabalha de maneira distribuída, podendo processar dados em paralelo em um dado conjunto de máquinas. Ao contrário do MapReduce, seu funcionamento é baseado em uma arquitetura do tipo mestre-escravo, onde cada instância é um processo Java distinto, sendo apenas um mestre (também conhecido como *driver*) e N possíveis trabalhadores (*workers*).

Ademais, o Spark permite a construção de aplicações com diversas etapas consecutivas em uma única classe, sem haver a necessidade de concatenar diversas etapas distintas de *mappers* e *reducers*. A maior parte das operações realizadas por esse *framework* são executadas em memória primária (volátil), o que faz com que, de maneira geral, aplicações sejam executadas em várias ordens de grandeza mais rápido que no MapReduce [31].

Para conseguir processar informações de maneira distribuída, o Spark utiliza-se do conceito de *resilient distributed datasets* – RDD (conjunto de dados distribuídos e resilientes), uma coleção de elementos que podem ser operados em paralelo. O grau de paralelismo que um *dataset* pode ser processado depende do número de partições em que ele é separado. Por padrão, o Spark calcula automaticamente o número de divisões de acordo com o *cluster* em questão, porém também é possível especificá-lo manualmente [32].

Um dos diferenciais deste trabalho é a utilização do *framework* Spark para o processamento do tráfego de rede. Ao realizar operações em memória RAM, reduz-se drasticamente o tempo necessário para realizar os cálculos de clusterização, possibilitando assim a análise de tráfego de rede em *near real time* [33].

2.4.3 MapReduce vs. Apache Spark

O *framework* MapReduce é capaz de processar grandes volumes de informação textual de maneira eficiente e distribuída. Porém, apesar de robusto, ele possui limitações que diminuem sua eficiência para processamentos em *near real time*, sendo as principais [34]:

- O *framework* é construído baseado nas operações *map* e *reduce*, fazendo com que todos os processamentos possuam pelo menos uma operação de cada, sempre nessa ordem. Caso operações mais complexas sejam executadas, é necessário concatenar diversos pares de *mappers* e *reducers*, o que pode tornar o processo como um todo significativamente mais lento.

- Requer o desenvolvimento de uma classe Java para cada operação de *map* e *reduce* distinta, fazendo com que o desenvolvimento de operações maiores sejam mais complexas, verbosas e menos intuitivas.

- Esse *framework* depende bastante de leitura e escrita em memória secundária (e.g. discos rígidos), o que limita o desempenho dessas operações de acordo com a velocidade dos discos onde as informações são lidas e escritas.

Dessa forma, é possível perceber que, apesar de possibilitar processamentos de grandes volumes de dados de maneira distribuída, o MapReduce não provê as ferramentas necessárias para realizar operações em *near real time*.

2.4.4 Spark Streaming

Por padrão, o Spark apenas é capaz de processar conjunto de dados estáticos, sejam eles de fontes internas (declarados em memória) ou externas (lidos do HDFS, Hive, HBase, Oracle DB, etc.). Para lidar com fluxos de dados contínuos, o Spark utiliza de uma abstração, denominada Spark Streaming.

O Spark Streaming é um extensão das funcionalidades padrões do Spark, permitindo que tráfego em tempo-real seja capturado e, em sequência, processado. Dessa forma, diversas outras fontes de dados contínuos podem ser processadas, tais como o Kafka e também o Flume. Essa abstração somente é possível graças ao Discretized Stream (também conhecido como Dstream), que representa um fluxo de informações dividido em pequenos lotes, coletados em ciclos pré-configurados.

Dstreams são construídos para serem equivalentes aos RDDs, permitindo que o Spark os processe de maneira transparente. Devido ao fato dos lotes apenas serem processados em períodos de tempo bem divididos, esse processamento é considerado *near real time*. A Figura 2.12 mostra de maneira simplificada o funcionamento do Spark Streaming. Conforme pode ser visto nela, o Spark Streaming é capaz de receber um fluxo contínuo de informações e transformá-los em conjuntos discretos de eventos. Nela, os envelopes representam um objeto da classe *DStream*, que são passados ao Spark de maneira transparente. Os envelopes abertos são representações de RDDs que foram processados pela ferramenta.

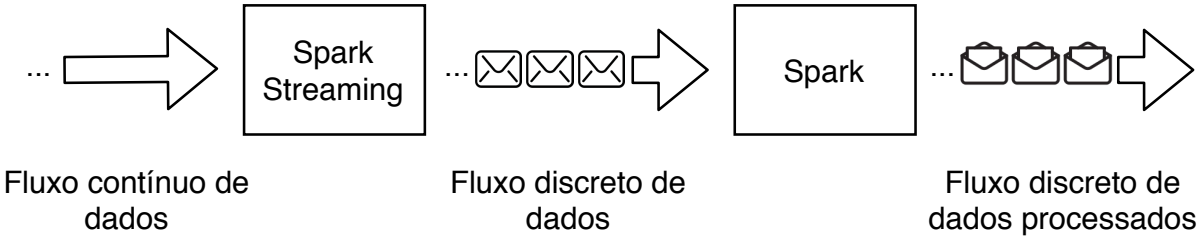

Figura 2.12: Funcionamento do Spark Streaming. Adaptado de [4].

Para tornar o processamento de dados mais rápido, o Spark Streaming apenas permite realizar operações que são transformações de dados. Ou seja, é possível apenas gerar novas variáveis ou sobrescrever as atuais a partir de operações feitas nas variáveis já existentes [4].

O intuito inicial deste trabalho era utilizar o Spark Streaming para realizar o processamento contínuo do tráfego de rede e assim ser capaz de detectar anomalias através da clusterização dos pacotes recebidos. Porém, como esta ferramenta limita-se a apenas realizar transformações de dados, não foi possível treinar novos modelos, pois o modelo gerado pelo algoritmo de clusterização K-prototypes, explicado na seção 2.6, é um objeto completamente novo e não deriva diretamente de uma transformação dos dados de entrada.

Com isso, não foi possível utilizar o Spark Streaming como ferramenta de processamento de dados para a arquitetura proposta. Para se obter o resultado esperado, simulou-se o funcionamento do Spark Streaming no Spark convencional, com o auxílio do Apache Kafka. A implementação resultante é explicada em detalhes no capítulo 3.

2.4.5 Hadoop

O *Hadoop* é um conjunto de ferramentas de código aberto, escrito em Java, mantido pela *Apache*. Dentre suas principais funções, pode-se destacar: processar grandes quantidades de dados com sua própria implementação do *MapReduce* (explicado na seção 2.4.2) e prover um sistema de arquivos distribuído, o HDFS (explicado na sessão 2.4.6) [35].

A gênese deste programa ocorreu após as publicações dos artigos da Google sobre a implementação própria de um sistema de arquivos distribuídos, o *Google File System*, e a sua versão do *MapReduce*, o que eventualmente levou o grupo *Apache* a desenvolver suas próprias versões destes projetos também.

Utilizar o *Hadoop* permite que grandes arquivos sejam processados e analisados de maneira flexível (não há restrição para a extensão do arquivo sendo analisado, seja ele estruturado ou não) e resiliente, devido à sua natureza distribuída [36].

A principal vantagem do *Hadoop* em relação à outras ferramentas de cunho similar é o seu baixo custo de implementação: o projeto em si é de código aberto e pode ser baixado diretamente de seu site e ele é capaz de rodar em equipamentos computacionais comuns, o que elimina a necessidade de utilizar computadores com elevadas especificações. A principal desvantagem, porém, é a grande dependência que as ferramentas têm das velocidades de leitura e escrita dos discos rígidos.

2.4.6 Hadoop Distributed File System - HDFS

Operações em *Big Data* podem ser efetuadas e armazenadas em quaisquer sistemas de arquivos distribuídos, como explicado na sessão 2.4.1. Porém o sistema de arquivos mais utilizado para essa finalidade é o *Hadoop Distributed File System*, que foi inicialmente baseado na implementação feita do *Google File System* - GFS [28].

A arquitetura do HDFS consiste do tipo mestre/escravo. Nela, o mestre é chamado de *NameNode* e ele é responsável por gerenciar todos os metadados e também um ou mais escravos, chamados de *DataNodes*. Um arquivo a ser armazenado no HDFS, por exemplo, é dividido primeiro em diversos blocos e esses elementos são distribuídos entre *datanodes*. O *namenode* é o responsável por determinar quais blocos serão mapeados para quais escravos, quais blocos serão replicados e também quaisquer atualizações de registros necessárias. Porém, são os *datanodes* que se encarregam das operações de leitura e escrita.

De maneira nativa, o HDFS provê uma interface por linhas de comando bastante similar a qualquer distribuição Linux, conforme ilustrado na figura 2.13. Há, porém, implementações de terceiros que consistem de interfaces gráficas para esse sistema de arquivos.

```
[root@vm-cdl-d-01 ~]# hadoop fs -ls /user/mateus
Found 5 items
drwx------   - mateus supergroup          0 2017-05-15 21:00 /user/mateus/.Trash
drwxr-xr-x   - mateus supergroup          0 2017-05-29 10:31 /user/mateus/.sparkStaging
drwx------   - mateus supergroup          0 2017-05-16 14:15 /user/mateus/.staging
drwxr-xr-x   - mateus supergroup          0 2017-05-18 07:54 /user/mateus/CSVs
-rw-r--r--   3 mateus supergroup       4607 2017-05-14 20:00 /user/mateus/teste.csv
[root@vm-cdl-d-01 ~]#
```

Figura 2.13: Interface de linha de comando para o HDFS [Fonte: Autor].

2.4.7 Kafka

O Apache Kafka consiste de uma ferramenta de mensageria escrita nas linguagens Scala e Java [37] que possui alta integração com o ecossitema *Hadoop*. Sua arquitetura é do tipo produtor-inscrito, tolerante à falhas, distribuída e escalável. Um sistema de mensageria é aquele responsável por transferir mensagens de uma entidade (normalmente uma aplicação) para outra. Dessa forma, ao se terceirizar o processo de troca de mensagens, as aplicações podem se focar no processamento de informações, sem se preocupar em como elas são enviadas e roteadas.

O conceito de mensageria distribuída baseia-se no processo de enfileiramento assíncrono de mensagens. Dessa forma, duas arquiteturas são possíveis de serem implementadas: ponto-a-ponto e produtor-inscrito.

Em um serviço de mensageria tradicional, a arquitetura utilizada é a ponto-a-ponto. Nela, as mensagens enviadas pelo remetente são acumuladas em uma fila, e apenas um consumidor pode recebê-las. Assim que uma mensagem é lida da fila, ela é descartada. Essa arquitetura é mostrada na figura 2.14:

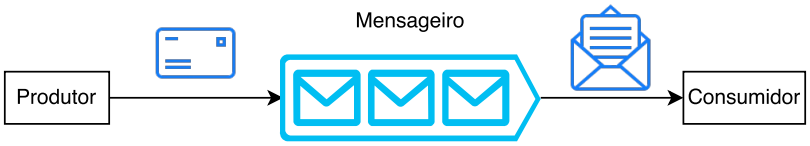

Figura 2.14: Exemplo de arquitetura ponto-a-ponto [Fonte: Autor].

O Kafka, porém, utiliza da segunda e mais robusta arquitetura de mensageria: publicação-inscrição [38]. Nesse sistema, diversos consumidores podem se inscrever em um ou mais tópicos e receber todas as mensagens trocadas naquele tópico. Dessa forma, uma mensagem somente é descartada da fila quando todos os consumidores inscritos naquele tópico receberam-na.

A arquitetura produtor-inscrito é mostrada na figura 2.15:

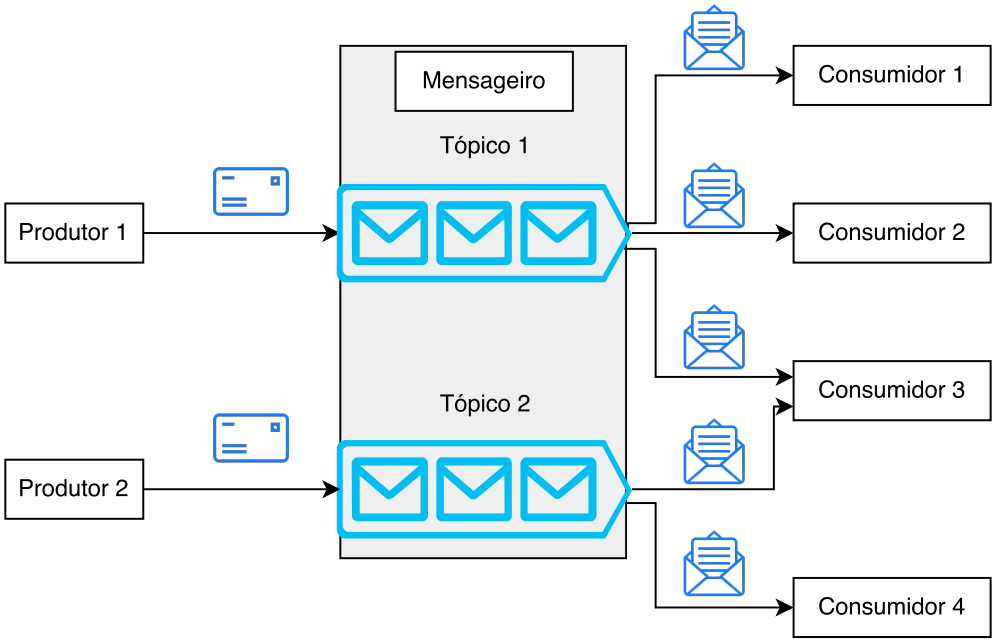

Figura 2.15: Exemplo de arquitetura produtor-inscrito [Fonte: Autor].

Na figura 2.15, percebe-se que o Apache Kakfa pode gerenciar mais de um tópico, e que os consumidores de tópicos distintos não possuem suas mensagens misturadas. O único consumidor que está a receber as mensagens dos dois tópicos é o de número três, pois ele está inscrito em ambos.

Dessa forma, algumas das vantagens de se utilizar o Kafka são:

- Confiabilidade: devido à sua natureza distribuída, ele é capaz de fazer particionamento, replicação e tolerância a falhas.

- Escalabilidade: para crescer horizontalmente o Kafka, basta implantar outro servidor ouvinte e adicioná-lo ao *cluster*.
- Desempenho: o Kafka tem alta vazão de dados tanto para publicação de mensagens quanto para consumo delas.

2.4.8 HBase

Para se armazenar informações e arquivos em *Big Data*, utiliza-se sistemas de arquivos distribuídos, como por exemplo o HDFS (como explicado na seção 2.4.6) ou bancos de dados que sejam preparados para lidar com a quantidade gigantesca de dados.

Uma das soluções de código aberto mais utilizadas na atualidade é o HBase. Esse banco de dados foi derivado do *Big Table*, desenvolvido pela Google [39] e é capaz de prover acesso aleatório rápido para grandes quantidades de informações estruturadas.

Armazenar informações em um sistema de arquivos distribuídos tem diversas vantagens (sessão 2.4.1), porém algumas situações não são favoráveis para essa arquitetura. Por exemplo, o *Hadoop*, que processa arquivos que se encontram dentro de DFS, apenas consegue acessar informações de maneira sequencial, o que significa que até mesmo para uma tarefa muito simples, todo o bloco de dados total deve ser carregado e processado. Pensando nessa limitação, o Hbase foi implementado para prover acesso aleatório para pontos específicos de amostras de dados de maneira atômica. Trata-se de um banco de dados orientado a colunas, onde cada linha é uma tupla de chave e valores, provendo assim as funcionalidades que faltam no ecossistema Hadoop. As operações de leitura e escrita são ilustradas na figura 2.16

Por usar o Hadoop como base de operações, o HBase herda os benefícios de um sistema de arquivos distribuído, incluindo escalabilidade, tolerância à falhas e distributividade [5].

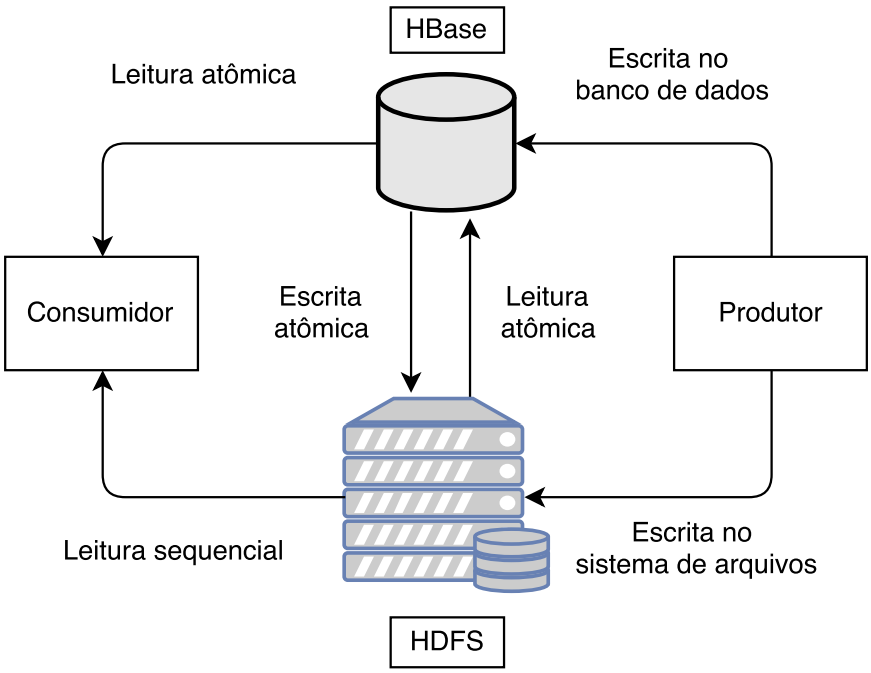

Figura 2.16: Funcionamento simplificado do HBase. Adaptado de [5]

2.5 TÉCNICAS PARA DETECÇÃO DE INTRUSÃO

Em termos gerais, há dois tipos de sistemas de detecção de intrusão: os que monitoram servidores e os que monitoram a rede. O primeiro tipo analisa diversos eventos que acontecem no sistema operacional, tais como identificadores de processo, chamadas de sistema e quantidade de recursos utilizada para tentar identificar quaisquer atividades anômalas. O segundo tipo, por sua vez, leva em consideração métricas de rede para realizar suas análises. De acordo com [40], os sistemas de detecção de intrusão podem ser classificados como :

- Sistema de detecção por assinatura: nesses sistemas, padrões capazes de identificar certos comportamentos são procurados dentro dos dados. Esses padrões também podem ser chamados de assinaturas. Dessa forma, para identificar comportamentos conhecidos previamente, é necessário que o sistema possua um banco de dados contendo informações sobre ataques já analisados, podendo assim distinguir comportamento normal do suspeito.

- Sistema de detecção por anomalia: para esses sistemas, tenta-se estimar um modelo que represente o comportamento normal do sistema em questão. Dessa forma, qualquer atividade que seja distante o suficiente desse modelo é considerada como atividade suspeita. De maneira análoga, pode-se criar um modelo que defina o comportamento anormal e que gere um alarme caso uma atividade seja suficientemente próxima desse modelo.

É possível perceber que ambos os modelos de detecção de intrusão são conceitualmente similares. Suas principais diferenças são os cenários em que cada tipo é mais eficiente: detecção por assinatura é capaz de detectar eficientemente ataques bem conhecidos, mas não consegue identificar novos tipos de intrusão, mesmo que sejam variações de ataques conhecidos. Por sua vez, sistemas de detecção de anomalias conseguem identificar comportamentos suspeitos mesmo que eles sejam desconhecidos, mas, de maneira geral, possuem taxa de erro (falso positivos + falso negativos) mais elevada que os sistemas baseados em assinatura [40].

Devido ao potencial de identificação de novas ameaças e variações de ataques já conhecidos, os sistemas de detecção de intrusão baseados em anomalias estão sendo os principais alvos de pesquisa e desenvolvimento no campo de segurança da informação. De acordo com [40], a detecção por anomalia pode ser dividida em três categorias: estatística, baseada em conhecimento e baseada em aprendizado de máquina. Esta última se baseia em gerar, a partir de um conjunto de dados (*dataset*) ilustrativo do comportamento da rede, um modelo capaz de categorizar as informações de entrada. A tarefa de conseguir o *dataset* para gerar este modelo pode ser árdua e consumir muitos recursos. Por outro lado, entre as vantagens de se utilizar aprendizado de máquina, destaca-se a possibilidade de aprendizado contínuo (*online*), isto é, poder alimentar o modelo com novos dados para melhorar o seu desempenho.

Em seu artigo sobre a utilização de técnicas de aprendizado de máquina em sistemas de detecção de intrusão de rede, [41] indica que diversas técnicas distintas foram implantadas. Com base em levantamento feito com trabalhos até 2010, uma grande variedade de técnicas distintas de aprendizado de máquina foram utilizadas, no entanto cada modelo teve as suas vantagens e desvantagens particulares.

Uma das maiores preocupações de fato ao se gerar modelos de aprendizado de máquina é o potencial de consumir grandes quantidades de recursos computacionais. Técnicas como redes neurais artificiais, especialmente as técnicas de *deep learning*, podem não obter o desempenho necessário para rodar em aplicações do tipo *near real time*. Por outro lado, conforme o estudo feito por [42], utilizar o algoritmo não-supervisionado K-Means é vantajoso para analisar tráfego de rede rapidamente, pois esta técnica possui custo computacional relativamente baixo.

2.6 ALGORITMO K-PROTOTYPES PARA CLUSTERIZAÇÃO DE DADOS

Conforme mencionado na seção 2.5, o algoritmo K-Means é uma alternativa para análise de grandes volumes de dados rapidamente. O K-Means faz parte da família dos algoritmos de clusterização e é do tipo não supervisionado, ou seja, não necessita que as informações processadas sejam previamente classificadas [43].

Algoritmos de clusterização possuem como objetivo principal segmentar a informação que está sendo processada, de modo que os pontos do *dataset* com características similares tendem a ser agrupados juntos [44]. Dessa forma, esses algoritmos são muito utilizados para investigar as relações que determinados atributos podem trazer para o conjunto de dados. A figura 2.17 ilustra o resultado de um processo de clusterização.

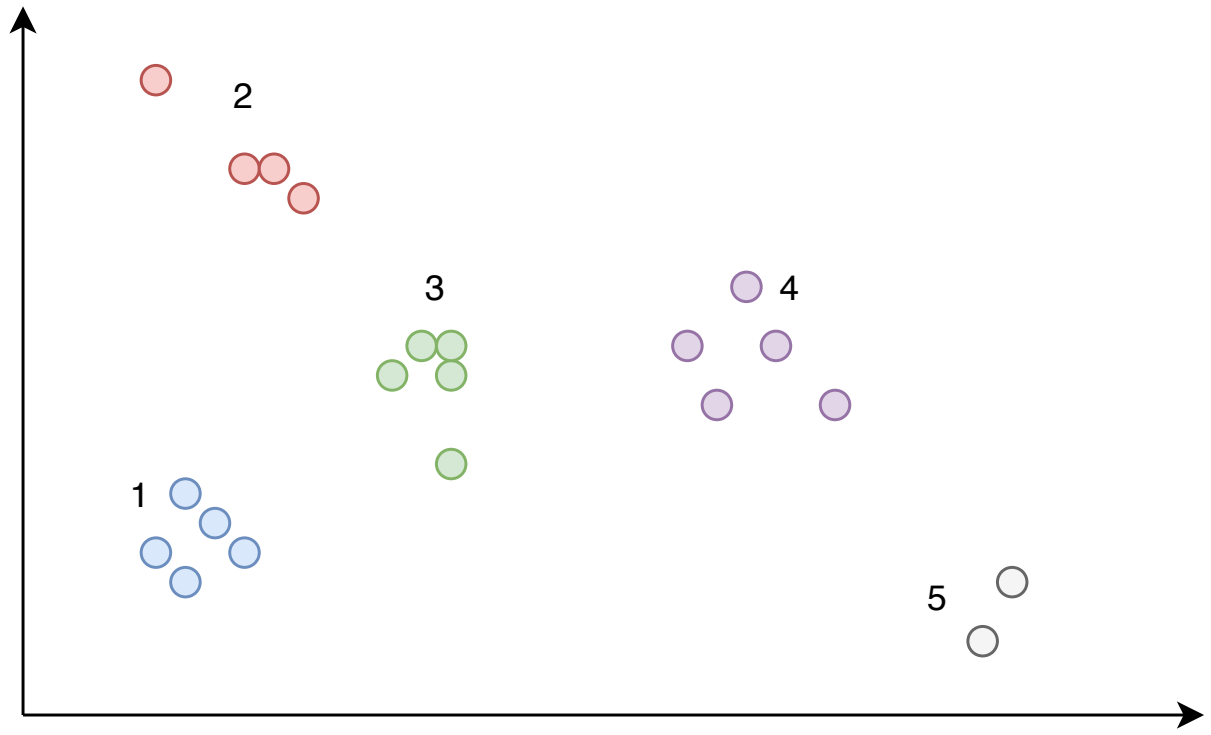

Figura 2.17: Exemplo de clusterização de dados. [Fonte: Autor]

Conforme pode ser visto na figura, os dados foram agrupados em 5 *clusters*, indicados pelas diferentes cores e números. É interessante ressaltar que o processo de clusterizar informações também pode ser utilizado para detectar anomalias no *dataset* [10]. Ao analisar um conjunto de dados e realizar o processo de clusterização, mapeia-se as regiões de densidade desses dados. Dessa forma, é possível assumir que, quanto mais próximo um ponto estiver de uma região de alta densidade, maior é a probabilidade dele pertencer àquela categoria. Dessa forma, pontos suficientemente distantes das regiões de maior densidade podem ser considerados anomalias naquele contexto. O processo de detecção pontos anômalos é mostrado na figura 2.18.

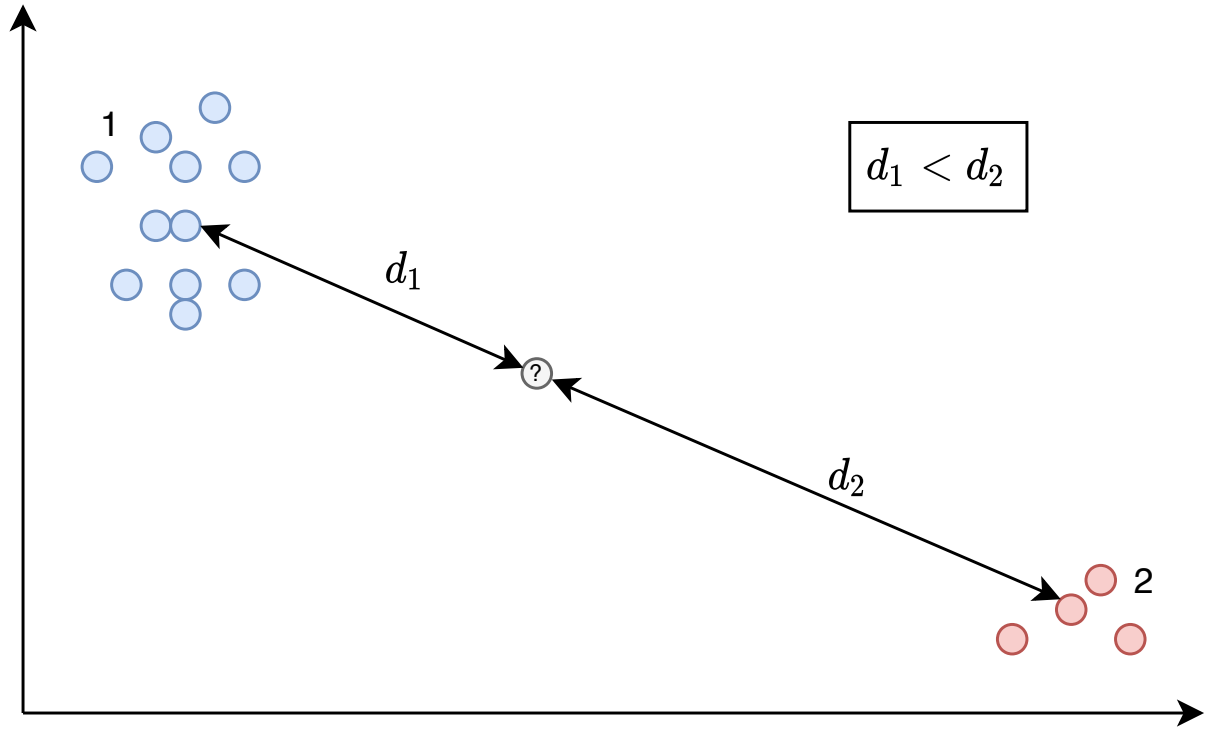

Figura 2.18: Exemplo de detecção de anomalia. [Fonte: Autor]

Conforme pode ser visto na figura, há um ponto ainda sem classificação entre dois clusters. A princípio, este ponto poderia pertencer a qualquer um dos dois ou ser uma anomalia. Assumindo um limiar de decisão λ, se $d_1 \leq \lambda$, então o novo ponto pertence ao *cluster* 1 (ele não pertenceria ao de número 2 pois $d_1 < d_2$). Porém, se $d_1 > \lambda$, o novo ponto se encontraria a uma distância superior ao limiar de decisão estabelecido e, portanto, não pertenceria ao *cluster* 1. Como $d_2 > d_1 > \lambda$, o ponto também não pertenceria ao *cluster* 2, caracterizando-o como uma anomalia.

Para realizar clusterização em tráfego de rede, é necessário extrair e analisar diversos atributos dos pacotes trafegados. Conforme mostrado em [45], alguns dos principais atributos a serem analisados são: endereço IP, número de porta, mensagem e fluxo. Apesar da maior parte desses parâmetros serem numéricos, ainda há aqueles que essencialmente são *strings* - também chamados de categóricos. Essa combinação de *features* numéricas e categóricas impossibilita o uso do algoritmo K-means padrão, pois este trabalha com a métrica de distância euclidiana, logo aceita apenas atributos numéricos.

Em seu paper de 1997, Huang et al. [46] propôs uma variação do algoritmo K-means voltada para o *data mining*, chamado K-Prototypes. Apesar de ser baseado no K-Means, o K-Prototypes não possui a restrição de seu antecessor, podendo processar entradas tanto do tipo numéricos quanto categóricas, mantendo a rapidez e eficiência do K-Means tradicional.

O K-prototypes utiliza o seguinte algoritmo [46] [47]: (i) Primeiramente, são selecionados k pontos do conjunto de dados a ser clusterizado, X. Esses pontos selecionados se tornam os primeiros centróides de seus respectivos clusters; (ii) Todos os outros pontos são alocados para o cluster cujo centróide está mais próximo, sendo a distância calculada de acordo com a equação 2.1; (iii) Calcula-se novos centróides de acordo com os pontos presentes no cluster. Os atributos numéricos do novo centróide são a média dos atributos numéricos dos pontos do cluster. Por sua vez, os atributos categóricos escolhidos são aqueles que mais aparecem dentre os atributos categóricos do cluster; (iv) Calcula-se novamente a distância entre os pontos e seus respectivos centróides. Caso um ponto esteja mais próximo do centróide de um cluster a que ele não pertence, este ponto é movido para o cluster em questão e os centróides de ambos os clusters são recalculados; (v) Os passos (iii) e (iv) são repetidos até que os pontos de X não sejam mais realocados.

A similaridade (distância) entre um ponto X e um centróide Q é medida considerando atributos categóricos e numéricos, de acordo com a fórmula

$$Dist(X, Q) = dist_{num}(X, Q) + \gamma.dist_{cat}(X, Q), \tag{2.1}$$

onde é utilizado o parâmetro γ para determinar o grau de importância das medidas categóricas em relação às numéricas. A variável γ é calculada de acordo com o desvio padrão dos atributos numéricos do *dataset* de treinamento [46]. Segundo o autor, o valor ótimo de γ encontra-se entre $1/3\sigma$ e $2/3\sigma$.

Para este trabalho, foi utilizada a biblioteca para Python kmodes [48], que implementa o algoritmo de clusterização K-prototypes. Nesta implementação, o valor de γ padrão é calculado por $\gamma = 1/2.\sigma$. Por sua vez, a distância numérica entre dois pontos, $dist_{num}(X, Q)$, é obtida através da distância Euclidiana [49]. Por fim, a função de distância entre os pontos categóricos, $dist_{cat}(X, Q)$, é calculada pela *matching dissimilarity function* [50], onde o mesmo peso é atribuído para a presença ou ausência de caracteres.

Caso apenas valores numéricos sejam utilizados, o parâmetro gama é zerado e a fórmula apenas trataria as distâncias numéricas, efetivamente voltando a ser o K-Means tradicional.

3 METODOLOGIA

Neste capítulo é explicado detalhadamente como as ferramentas de Big Data foram utilizadas para implantar a arquitetura de detecção de anomalias de rede proposta. São também abordados os módulos que foram desenvolvidos e a maneira com que eles se comunicam, seja através de mensagens ou de arquivos. Por fim, é mostrado um panorama do funcionamento da arquitetura como um todo.

3.1 TRABALHOS RELACIONADOS

Há uma série de trabalhos que exploram o contexto de detecção de intrusão e tráfego anômalo em associação (ou não) ao contexto de Big Data. O estudo publicado pela MITRE Corporation provê uma introdução sobre mineração de dados para detecção de intrusão em sistemas [51]. Aspectos como coleta de dados, seleção de atributos (*features*), armazenamento de informações e capacidade computacional são discutidos para nortear pesquisadores. A discussão inicial desta publicação foi complementada com o artigo feito por [45], onde os diferentes tipos de atributos (básicos, de conteúdo, temporais e baseados em *host*) são listados e analisados . Outro ponto importante deste trabalho foi o estudo sobre a relevância de determinados atributos na análise do tráfego em questão.

Já o estudo lançado por [52], da Escola de Engenharia e Tecnologia da Informação, na Austrália, analisa diversas técnicas diferentes para detecção de anomalias de rede, explicando suas diferenças, vantagens e desvantagens. Em sua análise, os autores consideram diversos fatores computacionais que influenciam diretamente a escolha e desempenho das técnicas, tais como escalabilidade e complexidade de algoritmos. Similarmente, o *Interscience Institute of Management and Technology*, localizado na Índia, publicou um grande compilado sobre técnicas de detecção de intrusão para redes de computadores utilizando detecção de anomalias, e comparou os resultados dos experimentos na sessão experimental [53].

Outro estudo lançado por uma instituição australiana, a *University of Melbourne*, analisou a possibilidade de realizar detecção de anomalias de redes utilizando métodos não supervisionados, tendo como principal foco os algoritmos de clusterização [54]. Diversos métodos foram analisados durante o trabalho, incluindo os baseados em particionamento, densidade e rede. Também são feitas comparações entre os métodos supervisionados e não-supervisionados. Por fim, algoritmos próprios de clusterização são propostos no trabalho e seus desempenhos são comparados entre si e com algoritmos já existentes.

O Instituto de Tecnologia de Georgia possui um *framework* chamado BotMiner, que analisa tráfego de rede para detectar BotNets [55]. Por usar técnicas de clusterização para auxiliar na tomada de decisão, este trabalho traz informações relevantes sobre o estado da arte desta linha de pesquisa, tais como a montagem de filtros pré-*clustering*, escolha de *features* relevantes e geração de novas com base no tráfego acumulado diário. Também vindo do mesmo instituto localizado na Georgia, o trabalho de [42] analisou a eficácia de utilizar o algoritmo de clusterização hierárquica *single-linkage clustering*, para detectar intrusões de redes através de anomalias encontradas.

A publicação feita em conjunto pelas Universidade de Michigan e Universidade da Florida foca principalmente na utilização de clusterização para detectar intrusão em tráfego de rede [56]. Em sua fundamentação teórica, é feita um recapitulação sobre técnicas de *clustering* e os diferentes tipos de *clusters* (centróide e par-a-par). Ademais, é proposto um algoritmo para detectar anomalias em tráfego de rede. Porém, os algoritmos utilizados não lidam com dados categóricos.

O *Institute for Systems Engineering Research*, da Universidade do Estado de Mississipi publicou um *survey* referente ao uso de *Analytics* em Big Data para detecção de intrusão de rede [57]. No trabalho, diversos pontos importantes são listados, incluindo algoritmos para detecção, possibilidade de usar aprendizado de máquina e *frameworks* no Big Data para processamento de dados.

O departamento de Ciência da Computação da Universidade da Carolina do Norte publicou um estudo sobre os principais problemas e desafios ao se implementar detecção de intrusão de rede utilizando aprendizado de máquina em Big Data [58]. Dentre eles, pode-se destacar: o elevado tempo necessário para se processar grandes volumes de dados e a falta de versatilidade que utilizar apenas aprendizado supervisionado pode trazer aos modelos derivados de tráfego de rede. Com o intuito de endereçar o primeiro problema, o Instituto Nacional de Tecnologia da Índia publicou um trabalho onde construíram um *framework* de Big Data para detecção de anomalias em redes de computadores de maneira rápida e eficiente [59]. Nesse trabalho, o sistema proposto foi desenvolvido no Apache Spark, o que diminuiu significativamente o tempo de processamento. Porém, os métodos de aprendizado de máquina comparados no trabalho foram apenas supervisionados.

Para melhor compreender as características de cada um dos trabalhos previamente citados, a tabela 3.1 apresenta uma taxonomia dos mesmos.

	Portnoy et al. [42]	Bloedorn et al. [51]	Kayacik et al. [45]	Ahmed et al. [52]	Jabez et al. [53]	Leung et al. [54]	Gu et al.[55]	Zhong et al.[56]	Wang et al.[57]	Suthaharan et al.[58]	Gupta et al.[59]
Dados categóricos					x	x					
Dados não categóricos	x						x	x			x
Algoritmos supervisionados					x	x	x		x		x
Algoritmos não supervisionados						x	x	x	x		
Clustering	x					x	x	x	x		
Big Data									x	x	x
Retreinos periódicos											
Detecção de anomalias	x	x	x	x	x	x	x	x	x	x	x
Aspectos computacionais		x	x	x					x	x	
Near Real Time											x

Tabela 3.1: Taxonomia dos trabalhos relacionados.

Conforme pode ser visto na tabela, trabalhos que focam em aspectos computacionais, tais como técnicas de detecção de intrusão, coleta de dados e seleção de atributos não apresentam análises profundas com algoritmos ou clusterização. O contrário também ocorre, onde trabalhos focados em implementação normalmente não abordam assuntos envolvendo aspectos computacionais.

Também é visível que nenhum dos trabalhos relacionados realiza retreinos periódicos para manter seus modelos atualizados. Similarmente, pode-se também perceber que em apenas um deles há a preocupação de construir uma arquitetura capaz de operar rapidamente, beirando o *near real time*. Ademais, percebe-se também que nenhum dos trabalhos relacionados contempla, ao mesmo tempo, todos os objetivos propostos neste trabalho, conforme mostrados na sessão 1.1.

3.2 PROPOSTA DE ARQUITETURA

A proposta deste trabalho é montar uma arquitetura em *Big Data* que seja capaz de detectar anomalias em tráfego de rede utilizando a técnica de clusterização *K-Prototypes*, ao mesmo tempo que busca usufruir ao máximo dos recursos de paralelismo e resiliência providos pelas ferramentas do ecossistema Hadoop. A figura 3.1 mostra uma visão simplificada do que foi implementado:

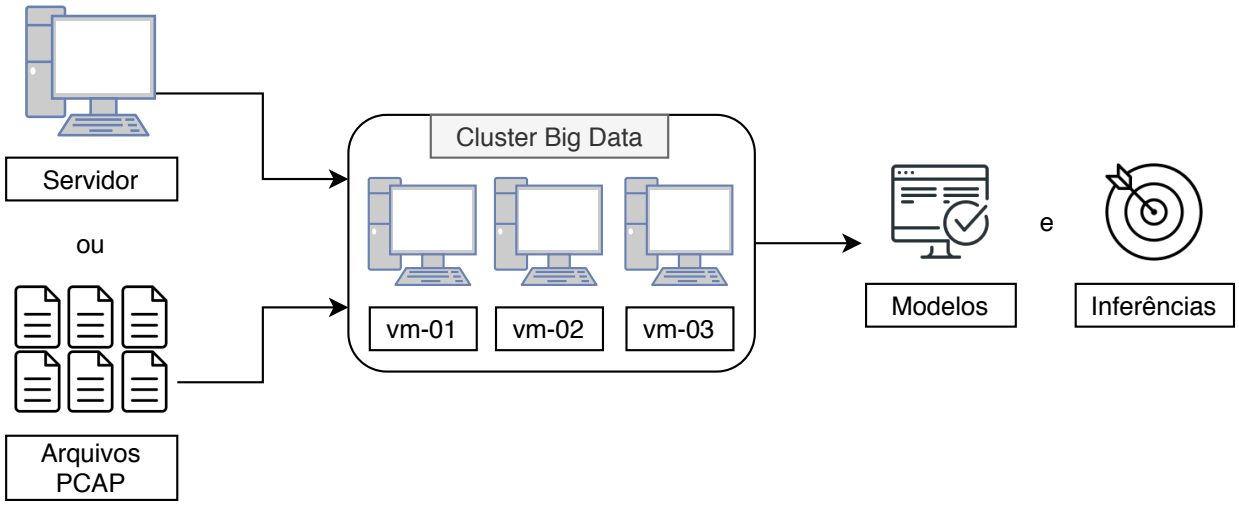

Figura 3.1: Visão geral da arquitetura proposta. [Fonte: Autor]

Conforme pode ser visualizado na figura 3.1, o cluster *Big Data* pode possuir duas fontes de tráfego de rede distintas: ou ele recebe as informações enviadas por um servidor escolhido ou são lidos arquivos de extensão PCAP para extrair a informação armazenada. No primeiro cenário, um pequeno agente escrito em Python é configurado no servidor a ser monitorado responsável por enviar ao cluster *Big Data* o tráfego enviado e recebido em tempo real nas interfaces de rede selecionadas. No segundo, por sua vez, blocos de tráfego de rede são analisados de uma vez, visto que o tráfego contido nos arquivos lidos já foi trocado e as informações lá contidas são apenas os registros do que aconteceu.

Após receber as mensagens, o cluster *Big Data* irá processar as informações com o *framework* distribuído Apache Spark. Como resultado desse processamento, gera-se um modelo que reflete o estado da rede em uma determinada janela de tempo. Para que os modelos gerados não fiquem desatualizados devido ao constante fluxo da rede, novos modelos são retreinados periodicamente pela arquitetura.

O módulo responsável por realizar as inferências com base nos modelos gerados pelo *cluster* encontra-se separado daquele responsável por treinar novos modelos. Dessa forma, é possível executá-los de maneira independente, e também em intervalos de tempo distintos. Como a operação de inferência é computacionalmente mais leve do que o treinamento, é possível executar este módulo com uma frequência mais elevada, gerando assim informações sobre o tráfego atual da rede mais rapidamente.

Nas sessões a seguir, cada componente da arquitetura ilustrada na figura 3.1 é explicado em detalhes.

3.3 CLUSTER BIG DATA

Para implementar uma arquitetura distribuída capaz de realizar clusterização em tráfego de rede em *near real time*, foi construído primeiro um *cluster* de Big Data capaz de rodar as ferramentas necessárias.

A distribuição do Hadoop escolhida para a implementação foi a *Cloudera Distribution of Hadoop* versão 5.13.2. Este pacote contém as principais ferramentas utilizadas nas análises de grandes quantidades de dados, tais como o Apache Spark, Kafka, Hive, HBase e Impala. Outra grande vantagem desta distribuição é o gerenciador gráfico disponibilizado exclusivamente pela Cloudera: o Cloudera Manager. Esta ferramenta gráfica acessada pelo navegador permite gerenciar serviços e a saúde do *cluster* de maneira significativamente mais intuitiva, permitindo assim que os esforços sejam direcionados efetivamente ao desenvolvimento de aplicações.

Foram disponibilizadas três máquinas virtuais para a implantação deste cluster no laboratório Latitude, do departamento de Engenharia Elétrica. A configuração delas foi idêntica, sendo listada a seguir:

- Sistema Operacional: CentOS 7.5

- HD: 300 GB
- RAM: 8 GB
- Rede: 1 Gbps

O ambiente em que essas máquinas são virtualizadas é o *Hyper-V*, da *Microsoft*. Os *hosts* responsáveis por rodá-las encontram-se atrás do firewall principal do laboratório, conforme pode ser visto na figura 3.2.

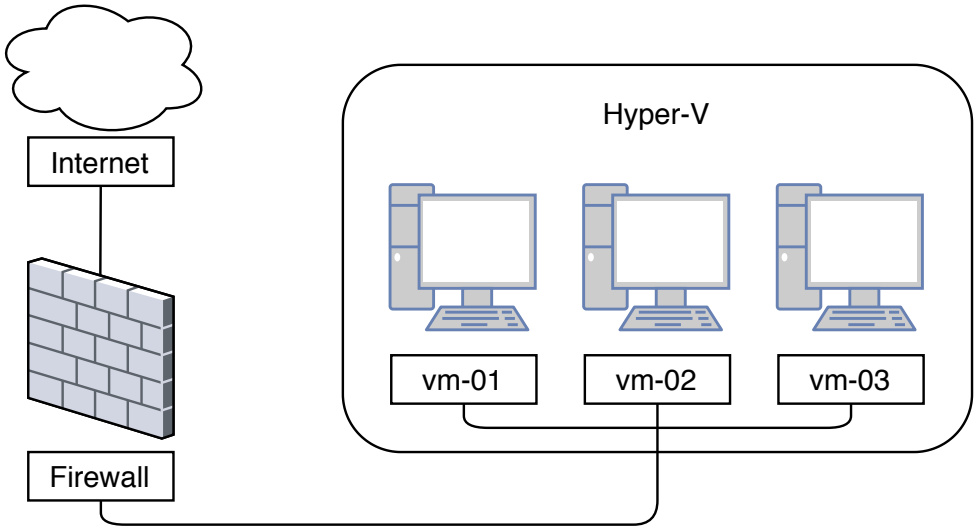

Figura 3.2: Arquitetura de rede simplificada das máquinas virtuais. [Fonte: Autor]

De acordo com a figura 3.2, as máquinas virtuais encontram-se conectadas ao *Firewall* do laboratório, e este por sua vez é o *gateway* para a Internet. É importante destacar que o cluster virtual encontra-se em seu próprio segmento de rede, separado das outras aplicações hospedadas na infraestrutura e, principalmente, da Internet. Foram incluídas diversas regras de roteamento para que conexões de origem externa não sejam capazes de alcançar as máquinas virtuais e que apenas o tráfego necessário de origem interna à essa sub-rede seja permitido.

As regras mencionadas foram implantadas como medida de segurança devido à natureza dos dados que serão analisados no *cluster* Big Data. Caso ocorra alguma falha de segurança durante o processamento do tráfego malicioso pelas máquinas virtuais, as regras criadas visam reduzir as possibilidades dos ataques conseguirem realizar seu conjunto de instruções original.

Foram escolhidas três máquinas virtuais para a construção deste cluster pois este é o número mínimo de computadores em que é possível obter o fator de replicação padrão do Hadoop: 3. Dessa forma, os serviços instalados foram distribuídos entre as máquinas de forma a prover redundância e possibilitar o paralelismo durante os processamentos de dados.

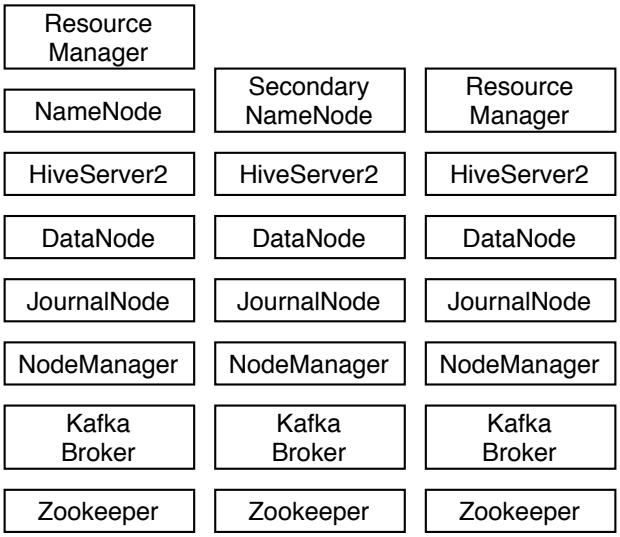

Figura 3.3: Distribuição dos serviços entre as máquinas virtuais. [Fonte: Autor]

Conforme pode ser visto na figura 3.3, cada unidade computacional do cluster possui uma instância dos serviços essenciais do Hadoop: Zookeeper, NodeManager, JournalNode e DataNode. Dessa forma, qualquer uma das três máquinas pode ser retirada do cluster sem comprometer o funcionamento das ferramentas, garantindo que as bases de dados, modelos gerados e outros processamentos em execução não sejam perdidos caso ocorra uma falha em um dos nós. As exceções desta regra são os serviços *Resource Manager* e *NameNode*, das ferramentas YARN e HDFS respectivamente. Elas, por sua vez, possuem apenas duas instâncias, ao invés de três. Porém, elas conseguem trabalhar de forma independente mesmo que a correspondente pare de funcionar, mantendo o correto funcionamento do cluster.

Apesar de não fazer parte do grupo de aplicações padrão do ecossistema Hadoop, é recomendável manter mais de uma instância de Kafka Broker. Como estes serviços trabalham em conjunto, caso algum produtor esteja enviando mensagens para um servidor Kafka e este venha a apresentar falhas, essas mensagens serão encaminhadas para outra instância disponível, de modo a não interromper o fluxo de mensagens e não perder nenhuma informação gerada pelo cliente. Os servidores Kafka trocam metadados entre si frequentemente, de modo que assim que a instância defeituosa voltar a funcionar adequadamente, ela irá receber as informações sobre as mensagens trocadas durante o tempo em que esteve indisponível.

3.4 HONEYNET

Parte do tráfego processado durante os experimentos deste trabalho são oriundos da *Honeynet* montada no laboratório do Departamento de Engenharia Elétrica em 2017. A arquitetura também foi montada em ambiente virtualizado na plataforma *Hyper-V* e, as motivações, configurações e passo-a-passo da implementação podem ser encontrados em [60] e [6].

É importante ressaltar que a *Honeynet* não possui enlaces diretos com nenhuma outra solução hospedada no laboratório, de modo a manter todo o tráfego malicioso contido dentro do segmento de rede e evitar que possíveis ameaças comprometam o funcionamento de outros sistemas. A figura 3.4 mostra a arquitetura lógica da *Honeynet*.

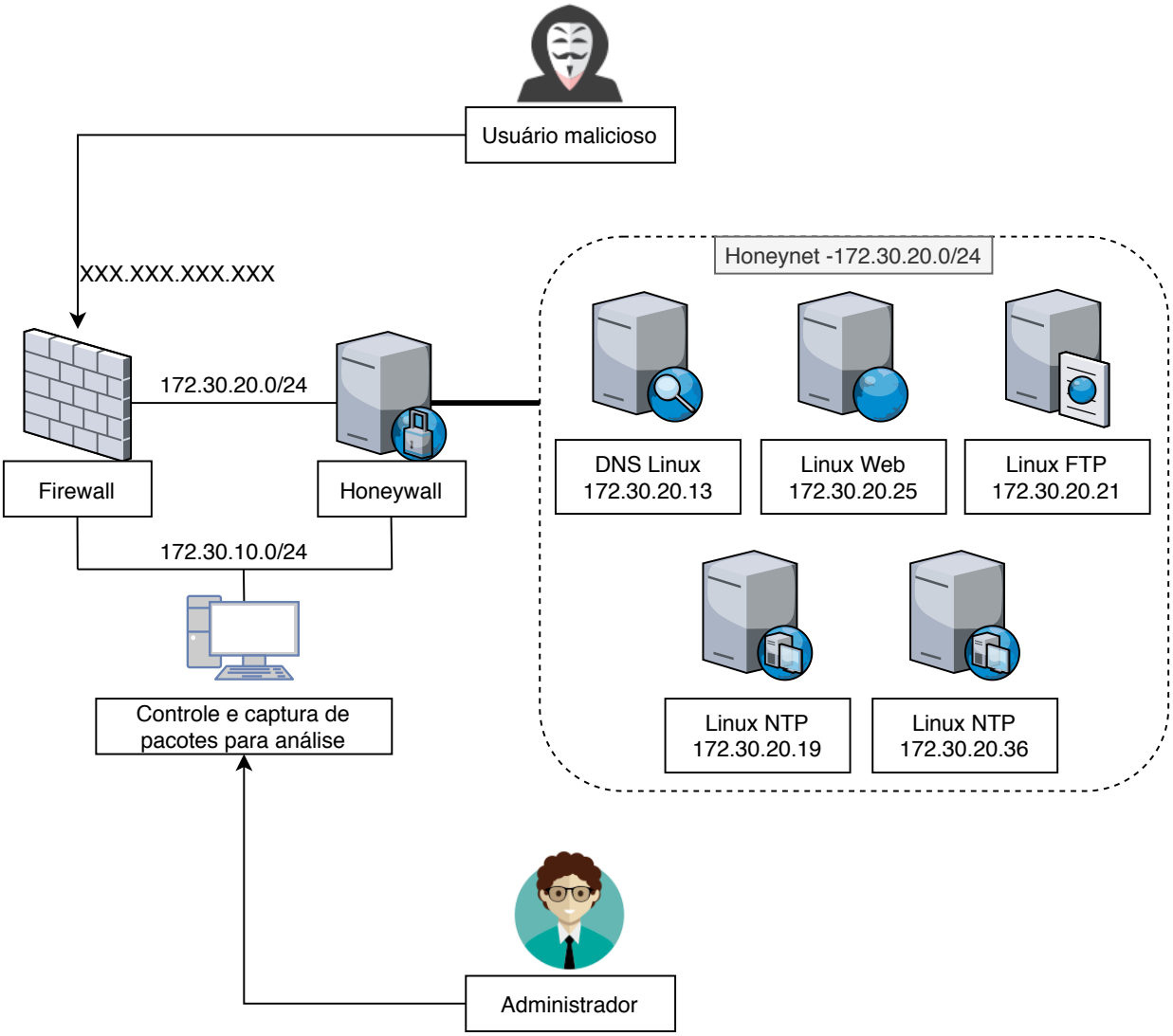

Figura 3.4: Arquitetura lógica da Honeynet montada em [6].

Ao analisar a figura 2.9, é possível perceber que existem três redes distintas nesta arquitetura: gerência, *honeynet* e a Internet. A primeira possui faixa de IP 172.30.10.0/24 e possui acesso às interfaces de gerência de ambos o Firewall e Honeywall, onde as informações de tráfego podem ser gerenciadas e coletadas por um administrador de rede; A segunda, de faixa IP 172.30.20.10/24 é efetivamente o segmento de rede onde ataques acontecem e todos os nós utilizados como isca apenas possuem interface nesta rede, para evitar que atacantes consigam comprometer a máquina e espalhar para outras sub-redes; Por fim, há a Internet, que é de onde todos os ataques são originados.

É importante ressaltar que o endereço de DNS público do *Firewall* externo não é divulgado para nenhum servidor de DNS externo e o endereço IP utilizado na interface também não fora publicado na Internet. Dessa forma, é possível assumir que quaisquer conexões recebidas por esta interface são originadas de atacantes virtuais, que estavam ativamente procurando por endereços IPs desprotegidos na Internet, através de redes de *bots* capazes de automatizar esta busca.

Quando um usuário malicioso descobre o endereço IP público do *Firewall*, ele é capaz de identificar quais os serviços associados a este endereço através do processo de escaneamento de portas. O *Firewall*, por sua vez, foi configurado para fazer o redirecionamento de portas para os serviços correspondentes na *Honeynet*, i.e. os pacotes destinados à porta 23 são encaminhados para o servidor de Telnet, 53 para o DNS, 80 para o servidor *Web* e assim sucessivamente.

Entre o *Firewall* externo e as máquinas virtuais da *Honeynet*, foi montado o *Honeywall*, que é um *firewall* especializado para gerenciar o tráfego de *honeynets*. As conexões deste intermediador de tráfego foram configuradas para ser do tipo ”ponte”, onde não há a diminuição da *flag* TTL dos pacotes trafegados, tornando-o transparente para os atacantes. Esta ferramenta também possui a função de limitar a quantidade de conexões originadas de dentro da *honeynet*, caso ocorra o comprometimento de algum dos nós.

Todas as informações extraídas da *Honeynet* foram obtidas através de uma máquina presente na rede de gerência que coletou esses registros das interfaces de gerenciamento de ambos os *firewalls*. Foram gerados vários arquivos PCAP, a partir do tráfego observado, para serem processados no cluster de *Big Data*.

3.5 PRODUTORES

A função principal deste módulo é capturar as informações de tráfego a ser analisadas e enviá-las para o cluster de *Big Data*, para que possam ser processadas. Para o escopo deste trabalho, percebeu-se a necessidade de criar dois modos distintos de enviar informações para o cluster: de forma *offline* e *online*.

A primeira abordagem, do tipo *offline*, foi criada para validar a arquitetura construída e poder testar o correto funcionamento dos diferentes módulos. Esta abordagem permite que arquivos do tipo PCAP sejam lidos de um sistema de armazenamento (disco rígido local ou o *Hadoop Distributed File System*, por exemplo), possuam suas informações de tráfego extraídas e cada pacote lido é então enviado ao *cluster* para ser processado. A figura 3.5 demonstra o funcionamento deste módulo.

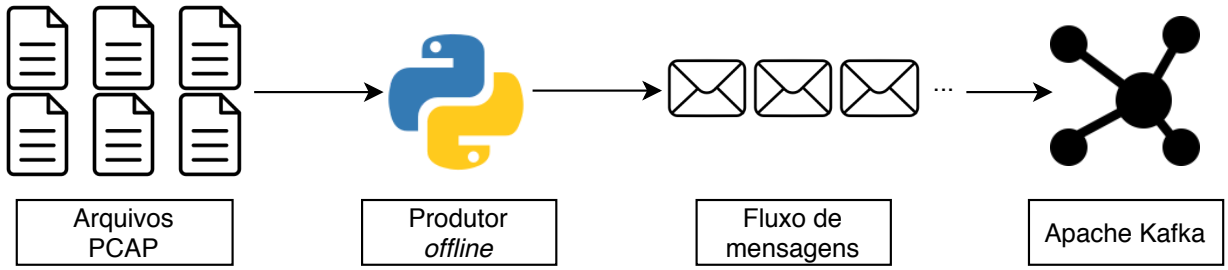

Figura 3.5: Funcionamento do produtor *offline*. [Fonte: Autor]

Conforme pode ser visto na figura, o produtor offline, após ler o tráfego armazenado nos arquivos, reproduz o fluxo original das mensagens respeitando o intervalo entre os pacotes trocados. O destino da cópia das mensagens feita é o serviço de mensageria Apache Kafka, sendo explicado melhor na sessão 3.6.

A segunda abordagem, por sua vez, possui comportamento *online* e tem finalidade de monitorar uma determinada unidade computacional em tempo quase real - *near real time*. O módulo responsável por coletar as informações de tráfego de rede reproduz qualquer tráfego que seja trocado nas interfaces que estão sendo monitoradas. De maneira similar ao módulo *offline*, os pacotes que passam pela interface de rede em questão são copiados e enviados ao Apache Kafka. Uma visão geral deste módulo pode ser visualizada na figura 3.6.

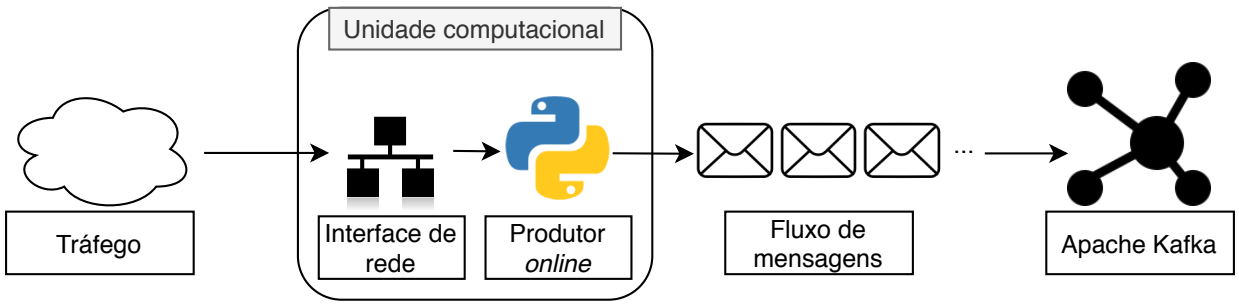

Figura 3.6: Funcionamento do produtor *online*. [Fonte: Autor]

Para extrair as mensagens da interface de rede, foi utilizada a biblioteca *tshark*. Essa biblioteca é especializada em tráfego de rede, e é o mesmo componente utilizado pelo programa *Wireshark* para capturar e analisar tráfego de rede [61]. Foi utilizada uma biblioteca *python* para que os conjuntos de instruções do *tshark* pudessem ser chamados no script que roda na unidade computacional sendo monitorada. Uma desvantagem desta abordagem é a necessidade de privilégios de super usuário ao capturar tráfego da interface, podendo tornar a implantação do módulo mais burocrática, pois não é qualquer usuário do sistema operacional que possui tais permissões.

3.6 APACHE KAFKA

Conforme explicado na seção 3.5, ambos os tipos de produtores desenvolvidos, durante o seu funcionamento, enviam mensagens com informações sobre o tráfego de rede em questão para o serviço de mensageria Apache Kafka. As mensagens, por sua vez, são entregues na mesma ordem de entrada para o treinador, que é outro módulo escrito em Python e executado no *framework* Spark, responsável por clusterizar o tráfego de rede e gerar os modelos que posteriormente serão utilizados para a inferência.

Dessa forma, é possível afirmar que o Apache Kafka atua como intermediador entre produtores e treinador, substituindo assim a necessidade de implementação de uma arquitetura cliente-servidor clássica, ilustrada na figura 3.7.

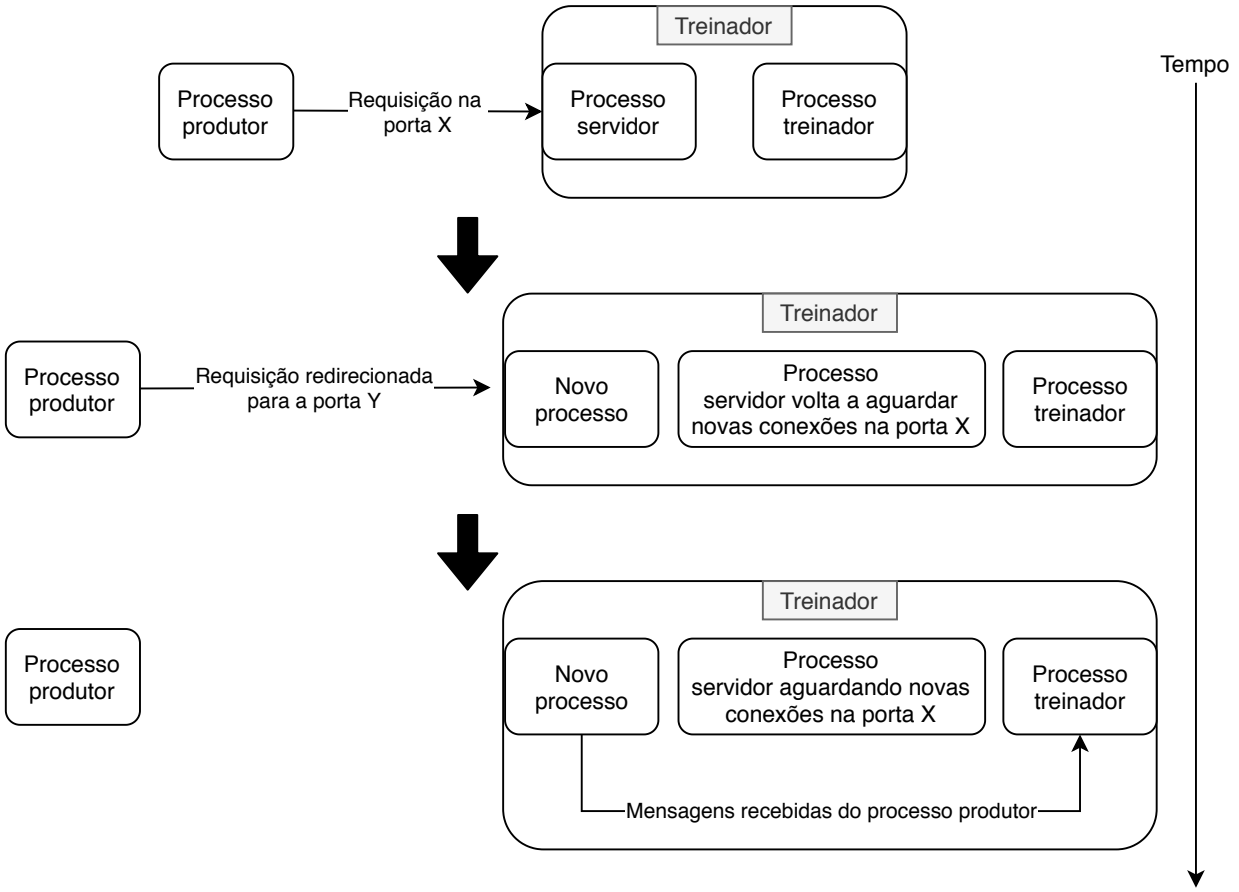

Figura 3.7: Ilustração da complexidade de implantação de uma arquitetura cliente-servidor. [Fonte: Autor]

A figura 3.7 ilustra como seria a troca de mensagens entre produtores e treinador caso uma arquitetura do tipo cliente-servidor fosse implementada. Primeiramente, o módulo do treinador teria um processo exclusivo para ficar esperando novas requisições externas em uma porta do sistema operacional, chamada na ilustração de X. Após receber uma nova requisição, este processo instanciaria um novo processo associado à outra porta do sistema, chamada de Y, e redirecionaria a requisição do produtor para o novo processo. O produtor, após ser redirecionado para o novo processo, envia todas as suas mensagens e termina a conexão. Ao terminar de receber essas mensagens, o novo processo, por sua vez, as repassa para o processo responsável por clusterizar essas mensagens, efetivamente iniciando a fase de treinamento.

Conforme pode ser percebido, implementar uma arquitetura cliente-servidor é uma tarefa complexa, que exige coordenação de múltiplos processos no sistema operacional e, portando, mais sujeita a falhas. Principalmente por esses motivos, preferiu-se utilizar o Kafka como serviço de mensageria intermediário entre os processos. A arquitetura que de fato foi implementada é mostrada na figura 3.8.

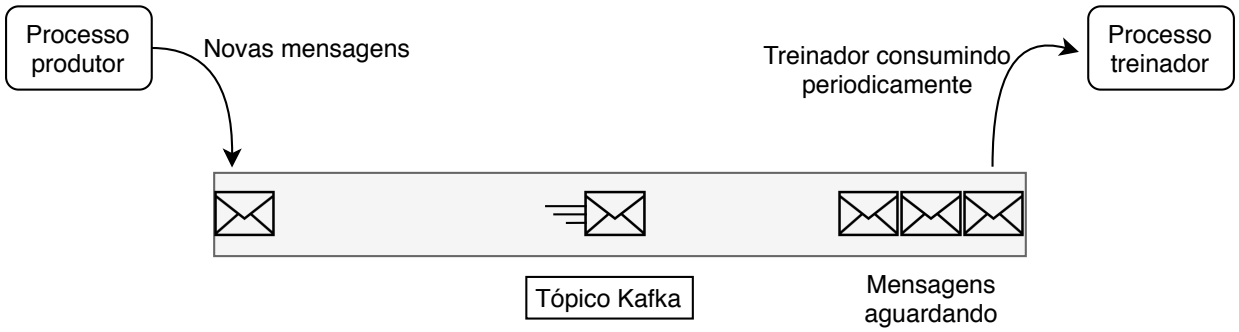

Figura 3.8: Troca de mensagens entre produtor e treinador através do Apache Kafka. [Fonte: Autor]

A figura 3.8 ilustra o processo de troca de mensagens entre o produtor e o treinador. O produtor envia as mensagens do tráfego de rede para o tópico Kafka à medida em que elas ficam disponíveis para transmissão (seja no modo *online* ou *offline*). O Kafka, por sua vez, se encarrega de armazenar essas mensagens até o momento em que o treinador esteja pronto para recebê-las e processá-las. Dessa forma, é possível perceber que a ferramenta age como *buffer* entre produtor e consumidor, aceitando mensagens a qualquer momento e entregando-nas ao consumidor apenas quando este estiver pronto [62].

Conforme também pode ser visualizado na figura 3.8, a implementação da arquitetura de troca de mensagens com um serviço de mensageria é significativamente menos propensa a erros que a cliente-servidor, pois não há necessidade de sincronização entre as partes, visto que cada módulo apenas precisa de um processo correspondente para funcionar e a complexidade de tratar o assincronismo das mensagens é terceirizada para o Apache Kafka.

3.7 TREINADOR

A partir do momento em que as mensagens encontram-se disponíveis no serviço Kafka, elas podem ser consumidas pelo treinador que, nesse contexto, exerce o papel de um consumidor Kafka. Este pode ser configurado para executar em intervalos de tempo Δt e, a cada nova execução, as mensagens mais recentes são lidas do serviço de mensageria e utilizadas para gerar um novo modelo de *clustering*, com base no algoritmo K-Prototypes. Dessa forma, os modelos utilizados para inferência são continuamente atualizados de acordo com a variação do fluxo de rede.

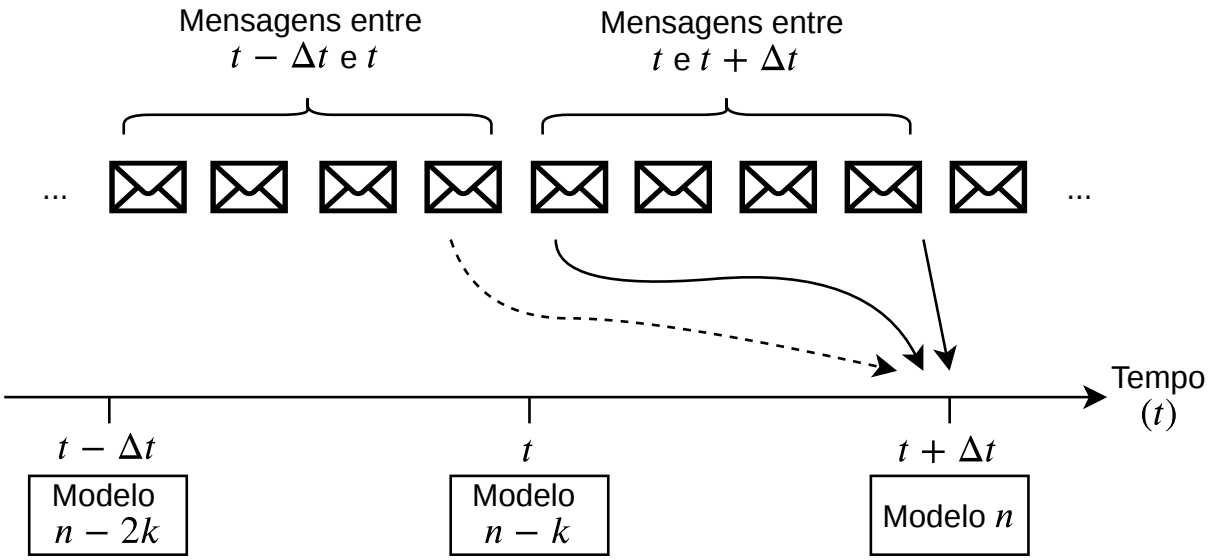

Figura 3.9: Ilustração da periodicidade de treinamento do modelo de clusterização. [Fonte: Autor]

A figura 3.9 ilustra o processo de treinamento ao longo do tempo. É possível ver que um determinado modelo n, treinado em $t + \Delta t$, utiliza todas as mensagens recebidas desde o último modelo gerado (em t) em conjunto a algumas mensagens da janela de tempo anterior (entre $t - \Delta t$ e t), com o objetivo de se manter uma memória do aprendizado passado.

Pode-se comparar o esquema de ingestão de mensagens utilizado pelo treinador como a janela deslizante do protocolo TCP, explicado na sessão 2.2.2. O início e o fim da janela de leitura se incrementam com valores configuráveis, seguindo a equação

$$t_1 = t_2 + \Delta t + \tau, \tag{3.1}$$

onde t_2 é o início da janela de ingestão, Δt é o intervalo de tempo que caracteriza um ciclo, τ é um parâmetro temporal que permite dilatar o tamanho da janela e t_1 é o fim da janela de ingestão. O resultado da equação 3.1 é a janela de ingestão de mensagens ilustrada na figura 3.10. Nela, $t_1 = t + 2\Delta t$ e $t_2 = t + \Delta t - \tau$.

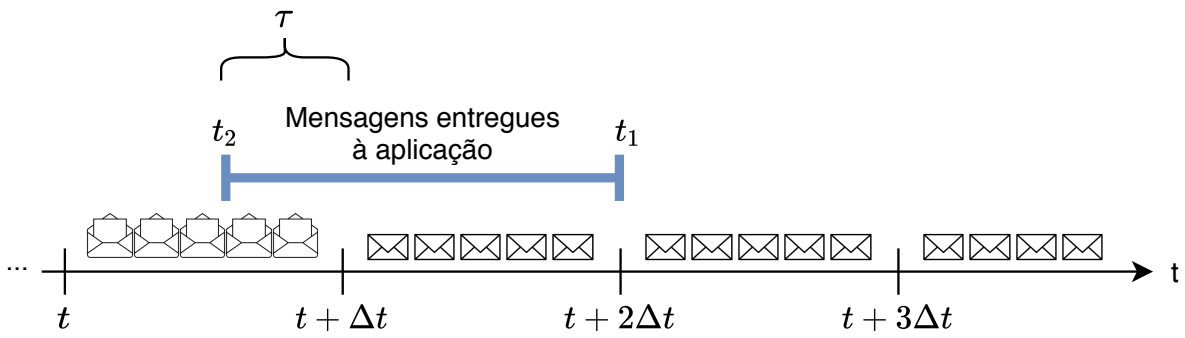

Figura 3.10: Janela de ingestão de mensagens do treinador. [Fonte: Autor]

É possível perceber a partir da figura 3.10 que o fator responsável por fazer o treinador utilizar mensagens referentes ao ciclo passado de mensagens é o τ, de forma que, se esse parâmetro for igual a zero, a aplicação somente irá considerar as mensagens recebidas dentro do último intervalo Δt. Para ilustrar o avanço da janela de ingestão do treinador, a figura 3.11 mostra o ciclo de treinamento que sucede àquele mostrado na figura 3.10, onde $t1 = t + 2\Delta t + \Delta t = t + 3\Delta t$ e $t2 = t + \Delta t - \tau + \Delta t = t + 2\Delta t - \tau$.

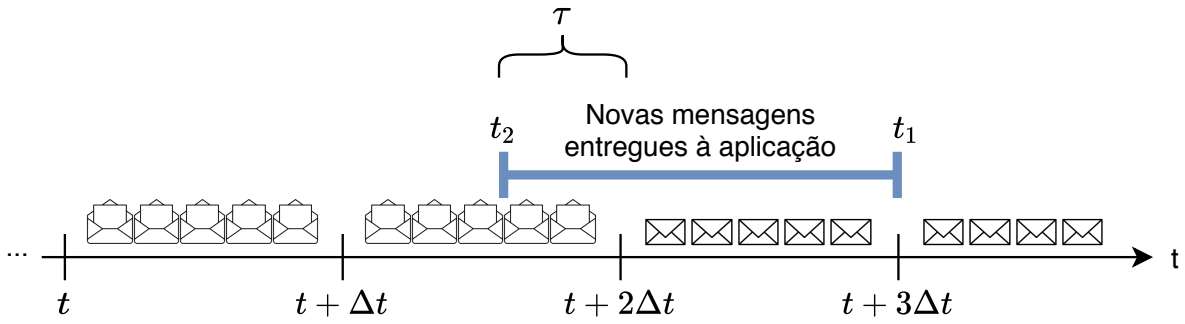

Figura 3.11: Janela de ingestão de mensagens do treinador após Δt. [Fonte: Autor]

As informações extraídas do serviço Kafka são transformadas em um objeto RDD onde cada mensagem trocada entre cliente e servidor é representada por uma linha contendo diversos campos, de tal forma que a quantidade de linhas equivale ao total de pacotes de rede trafegados no período.

Para a implementação do algoritmo K-prototypes em Python, utilizou-se a biblioteca K-modes, em que o autor baseou sua construção nos *papers* científicos originais que deram origem a este algoritmo, sendo eles [46] e [47]. A maioria dos parâmetros necessários para calcular os *clusters* de tráfego de rede obtiveram bons resultados em seus valores padrão que já vem pré-configurado na biblioteca. Precisou-se, porém, calcular corretamente o parâmetro K, que é o número de *clusters* que o programa irá encaixar os pontos.

Devido à grande relevância deste parâmetro, utilizou-se o método do cotovelo [7] para calcular o melhor valor de K para cada uma das implementações realizadas nas sessões 4.1 e 4.2. Este método consiste em calcular diversos modelos distintos, cada um deles com um número de K diferente (geralmente em ordem crescente) e posteriormente calcular a soma das distâncias entre os pontos e os centróides de seus respectivos *clusters*. Tendo esses valores, escolhe-se o primeiro K a obter uma queda brusca em relação ao K anterior. Este processo é ilustrado na figura 3.12.

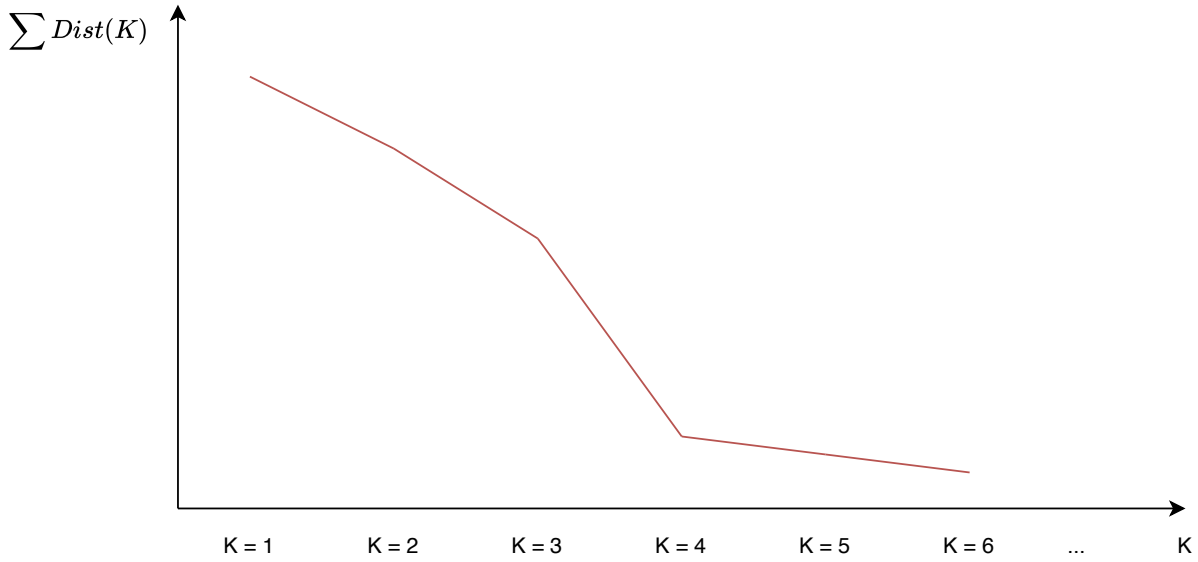

Figura 3.12: Demonstração do método do cotovelo. Adaptado de [7]

Conforme pode ser visto na figura 3.12, a primeira queda mais significativa na soma das distâncias dos pontos em relação aos seus respectivos centróides ocorre entre K = 3 e K = 4. Dessa forma, para este cenário em particular, escolher o número de clusters igual a 4 potencialmente geraria o melhor modelo.

Após a clusterização através do método K-Prototypes, um objeto Python é gerado contendo o melhor modelo dentre as iterações realizadas durante o treinamento. Adota-se o número padrão indicado pela ferramenta, que é de 100 [63]. Cada iteração corresponde a uma inicialização distinta dos centróides e a escolha do melhor modelo é feita ao selecionar aquele de menor soma das distâncias intra-clusters. O resultado do treinamento é um objeto Python, que então é armazenado em um sistema de arquivos persistente.

Para armazenar o modelo pronto, a linguagem *Python* oferece a biblioteca *pickle*, que é capaz de serializar objetos e salvá-los de maneira persistente. Foram implementadas duas maneiras de salvar estes arquivos: no disco local da máquina que está instanciando o *driver* do Apache Spark ou diretamente no *Hadoop Distributed File system*, mostrado na figura 3.13.

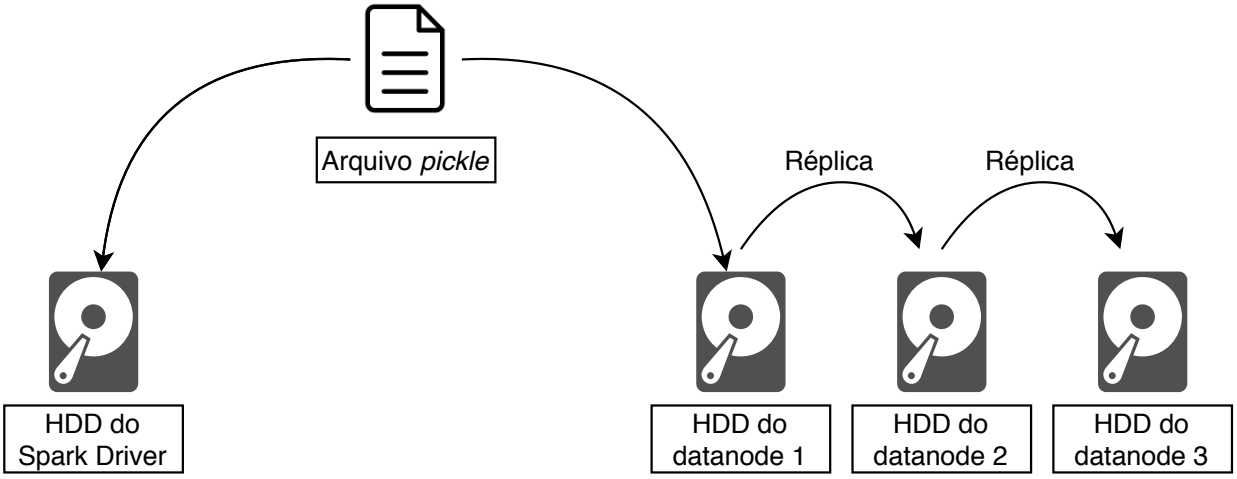

Figura 3.13: Processos de gravação do arquivo *pickle*. [Fonte: Autor]

Conforme pode ser visto na figura, há dois caminhos distintos que um arquivo *pickle* pode tomar: o primeiro é ser armazenado no disco rígido do servidor que está rodando o *driver* do Apache Spark e o segundo é ser armazenado no HDFS. A primeira opção foi utilizada nas etapas iniciais de desenvolvimento da ferramenta, pois esta opção é mais simples de ser implementada e também a operação de escrita no disco local é mais rápida que no HDFS, permitindo assim que esforços de desenvolvimento fossem priorizados em outras áreas do código mais críticas.

A segunda abordagem, por sua vez, foi implementada posteriormente e possui uma série de características intrínsecas ao ecossistema Hadoop. Ao ser armazenado no HDFS, o arquivo é gravado em um *datanode* e, automaticamente replicado em outros dois *datanodes*. Essa operação, apesar de gastar mais tempo em comparação à gravar no disco local, garante que o arquivo possui redundância suficiente para não ser perdido em caso de mal funcionamento de um servidor.

Por fim, é importante ressaltar que o arquivo *pickle* gerado, independente do destino da escrita ser disco local ou HDFS, possui em seu nome uma identificação temporal. O padrão escolhido é mostrado na figura 3.14.

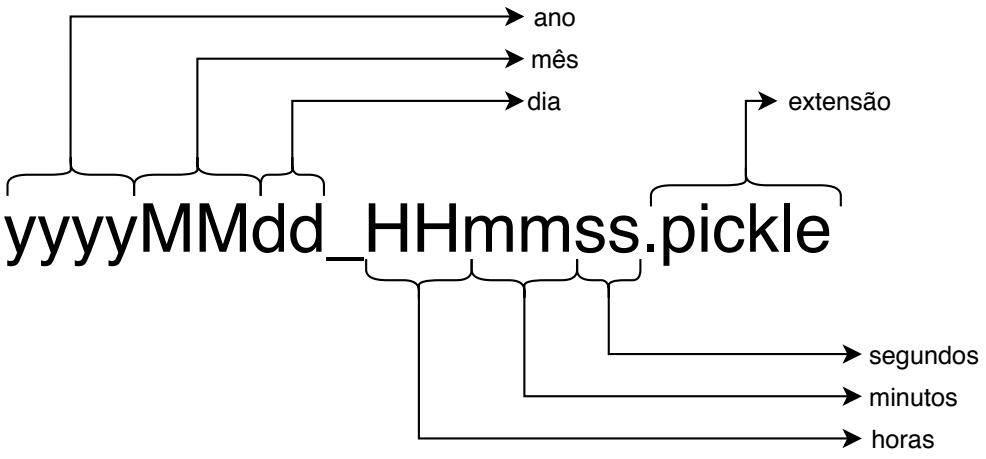

Figura 3.14: Padrão de nomenclatura dos arquivos *pickle*. [Fonte: Autor]

O padrão ilustrado na figura 3.14 foi escolhido pois garante que cada novo modelo gerado possua um nome único. Se, por exemplo, um modelo foi gerado às 14 horas, 40 minutos e 32 segundos do dia 15 de Julho de 2020, então o nome do arquivo gerado será 20200715_144032.pickle. Dessa forma, é possível garantir que os novos arquivos gerados não sobreponham os anteriores, o que permite também guardar o histórico desses arquivos para comparações futuras em intervalos maiores, como hora, dia, semana e mês. Ademais, esse sistema de nomenclatura serve de guia para que o módulo de predição seja capaz de escolher sempre o arquivo mais atualizado para suas predições.

3.8 PREDIÇÃO

A etapa posterior da arquitetura é a predição do tráfego de rede que é recebido, utilizando como base os modelos gerados pelo treinador . Este módulo também é escrito na linguagem Python, e executado no *cluster* através do *framework* PySpark. Similarmente ao treinador, o preditor também é executado de maneira periódica, porém este processo é configurado para executar independentemente do anterior, em intervalos de tempo $\Delta t'$, em que $\Delta t' < \Delta t$. A figura 3.15 é o complemento da figura 3.9 e ilustra como os modelos gerados pelo treinador são utilizados no módulo de predição.

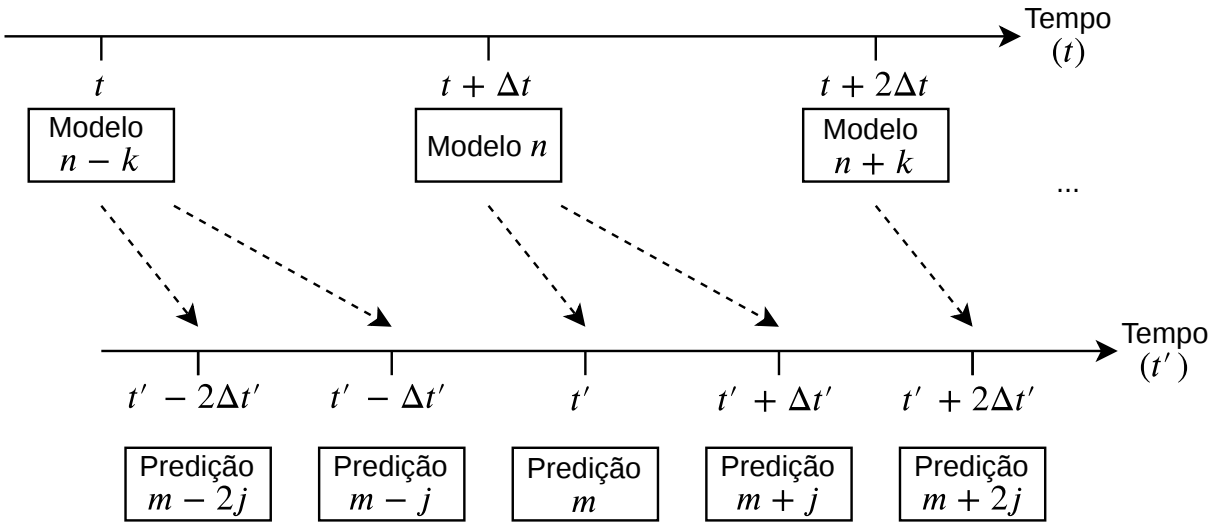

Figura 3.15: Ilustração do processo contínuo do preditor. [Fonte: Autor]

Conforme pode ser visto na figura, a predição acontece de maneira mais frequente que o treinamento de novos modelos, visto que a detecção requer resultados mais rapidamente e com tempos de processamento menores.

Para cada nova execução do preditor, busca-se o modelo mais recentemente treinado seguindo o padrão de nomenclatura explicado na sessão 3.7 e coletam-se todas as novas mensagens que chegaram ao serviço Apache Kafka desde a execução anterior. É importante lembrar que o preditor precisa estar configurado para buscar os arquivos de modelo no mesmo local em que o treinador os está salvando, seja no HDFS ou no disco local, visto que o descasamento dessas configurações pode acarretar na utilização de modelos defasados ou até mesmo erros na aplicação, que não conseguiu encontrar arquivos de modelos para ler. A figura 3.16 mostra os dois caminhos que o módulo de predição pode percorrer ao tentar ler um arquivo de modelo.

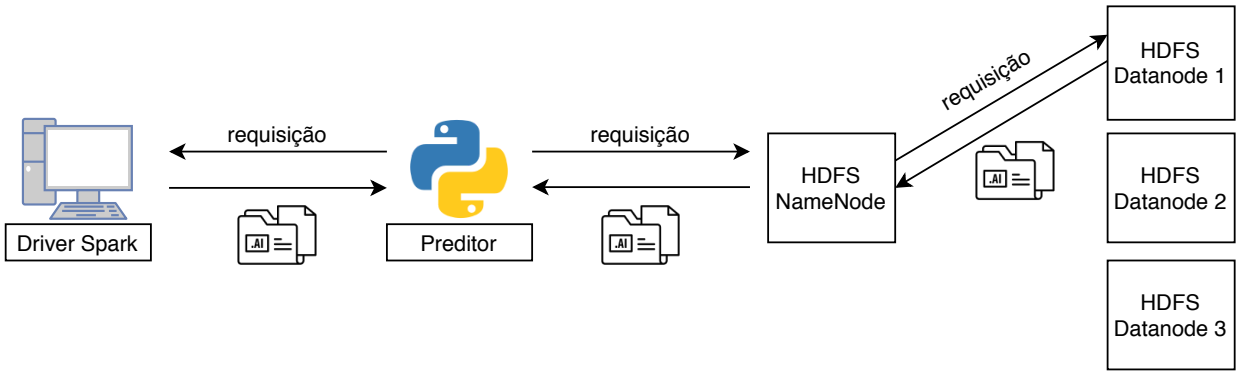

Figura 3.16: Processo de obtenção de um arquivo de modelo. [Fonte: Autor]

Conforme ilustrado na figura 3.16, o preditor pode requisitar um arquivo de modelo tanto para o disco local do *driver* Spark, quanto para o HDFS. Caso ele seja configurado para ler do HDFS, ele manda uma requisição para o serviço do NameNode, que serve de *proxy* para os DataNodes e entrará em contato com um dos três disponíveis para reaver uma das cópias espalhadas pelo *cluster*. Este método, apesar de mais demorado, é mais resiliente, pois se beneficia dos mecanismos de proteção contra perda de dados do ecossistema Hadoop.

O preditor, após obter o arquivo de modelo mais atualizado, coleta todas as novas mensagens que chegaram ao serviço Apache Kafka desde a execução anterior. Dessa forma, ele classifica os novos pacotes recebidos com base em uma regra de decisão configurável: se o pacote não pertence a um dos *clusters* identificados na etapa de treinamento, isto é, caso seu vetor de atributos possua distância ao centróide de qualquer cluster maior que um limiar de decisão pré-configurado, ele é classificado como anômalo e incluído no relatório criado pela ferramenta após cada ciclo. Para facilitar a compreensão, a figura 3.17 exemplifica o processo de decisão projetado em um plano bidimensional.

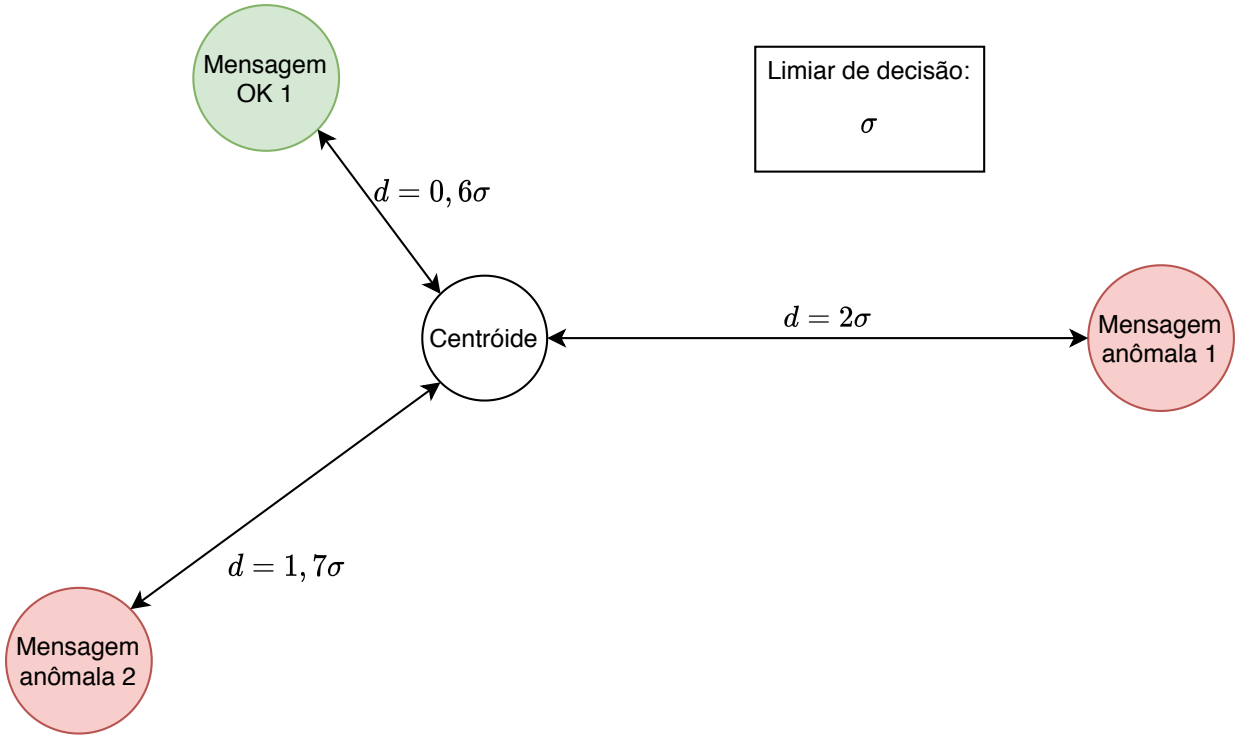

Figura 3.17: Ilustração do processo de detecção de anomalias. [Fonte: Autor]

Na figura 3.17 o limiar de decisão que separa o tráfego anômalo do normal é σ. Conforme esperado, as mensagens cuja distância ao centróide mostrado foi menor que σ são consideradas normais, ao passo que aquelas em que suas distâncias calculadas até o centróide foram maior que σ foram consideradas anomalias.

Por fim, a ferramenta reúne todas as mensagens que foram classificadas como anomalias e extrai um RDD contendo apenas mensagens com essa classificação. Esse objeto é então persistido no banco de dados noSQL do ecossistema Hadoop, o HBase. Escolheu-se esse modo de persistência para o preditor conseguir manter estado de quais pacotes já foram classificados como anomalias. Para economizar espaço no banco de dados, ao invés de armazenar a mensagem inteira com todos os campos utilizados para a predição, faz-se um hash da concatenação de todos os caracteres presentes no campo, de modo a gerar uma *string* única que identifica cada pacote considerado anômalo.

3.9 VISÃO DETALHADA

Uma visão detalhada da arquitetura montada neste trabalho, contendo todos os componentes listados no capítulo 3 é mostrada nas figuras 3.18 e 3.19.

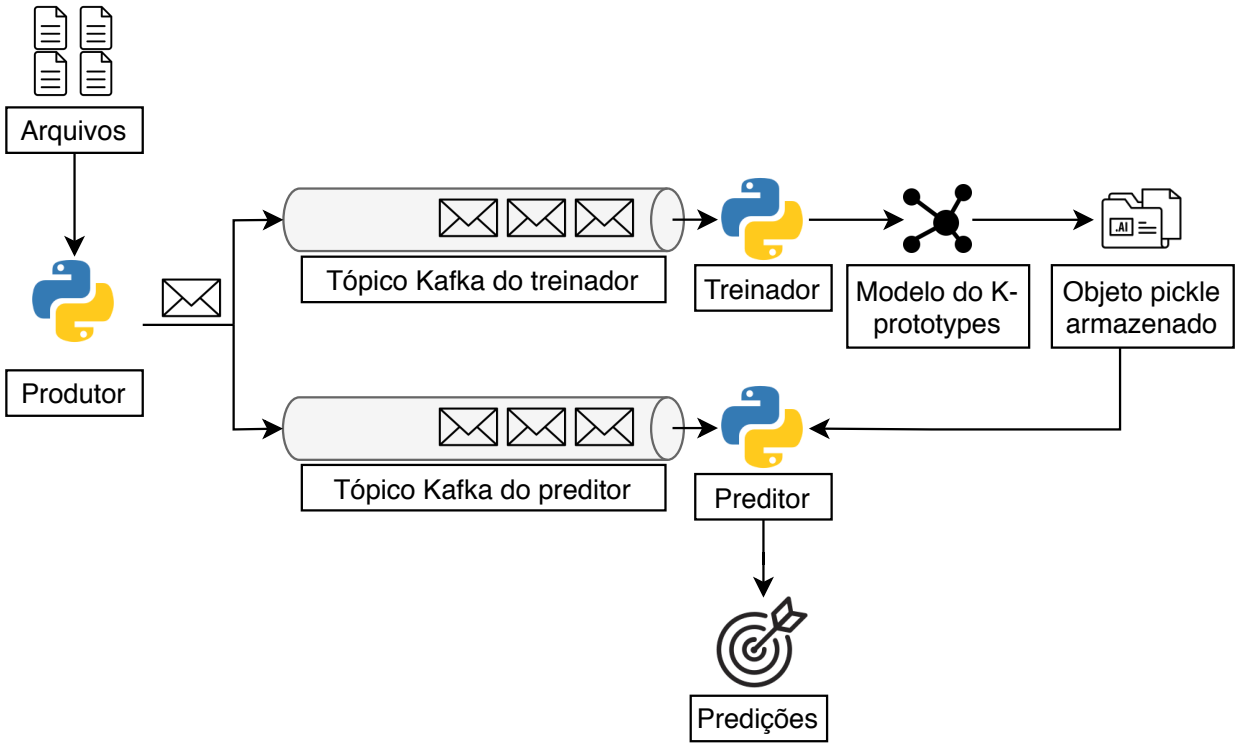

Figura 3.18: Arquitetura completa *offline*. [Fonte: Autor]

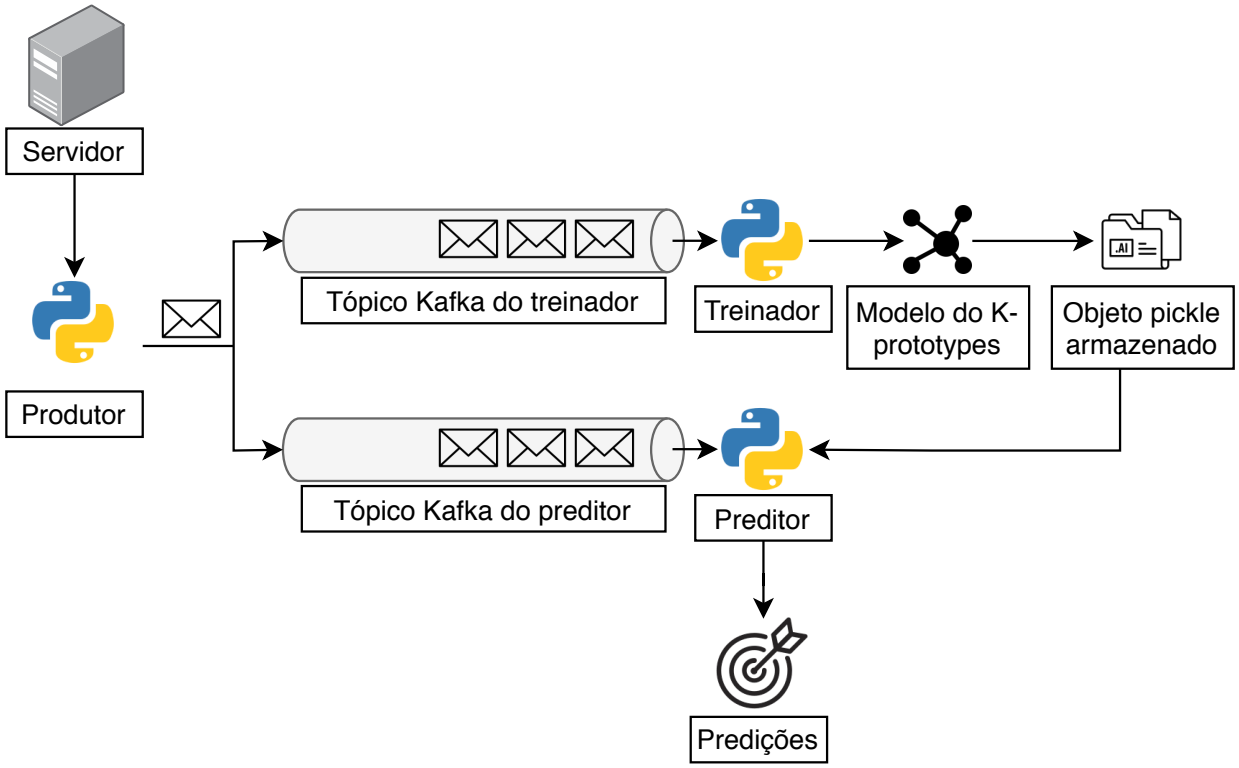

Figura 3.19: Arquitetura completa *online*. [Fonte: Autor]

Conforme pode ser visto nas figuras, grande parte dos módulos desenvolvidos durante a implementação deste trabalho são aproveitados em ambos os casos *offline* e *online*. Somente o produtor precisou de duas implementações distintas para contemplar os casos simulados nos experimentos do capítulo 4.

4 VALIDAÇÃO EXPERIMENTAL

Neste capítulo são realizados diversos experimentos com o intuito de testar a arquitetura proposta e também a eficácia da linha de produção construída. É importante lembrar que a proposta deste trabalho é construir uma arquitetura em *Big Data* capaz de detectar em tempo quase real (*near real time*) anomalias em tráfego de rede através do método de clusterização K-prototypes. Outro objetivo a ser atingido é garantir que os modelos utilizados para a detecção de anomalias seja periodicamente atualizado, para que os resultados obtidos não sejam defasados em relação à situação atual do tráfego de rede monitorado.

Dessa forma, os experimentos realizados foram divididos em dois grandes grupos: primeiramente, analisou-se o comportamento e desempenho da arquitetura ao processar arquivos pré-rotulados. É importante começar com esse tipo de abordagem pois ela permite que os resultados obtidos pelo preditor sejam comparados com a classificação real do tráfego. Com base nos resultados obtidos por esse método e também as métricas coletadas durante e após o processamento, executou-se o segundo lote de experimentos, em que tráfego real foi analisado pela arquitetura.

Por fim, com os resultados dos conjuntos de experimentos realizados na arquitetura, foram feitas análises sobre os valores calculados e as métricas obtidas e comparações com outros trabalhos com proposta similar.

4.1 ANÁLISE COM DADOS PRÉ-ROTULADOS

O primeiro experimento a ser realizado tem como premissa validar o correto funcionamento da arquitetura e seus componentes, bem como avaliar o desempenho do algoritmo *K-prototypes* em detectar anomalias de redes. Dessa forma, foi escolhido um *dataset* pré-rotulado para ser analisado, visto que os resultados calculados poderiam ser comparados com os valores reais fornecidos.

O conjunto de dados utilizados nessa etapa foi o *Network Intrusion Detection*, disponibilizado em 2018 na ferramenta Kaggle [9]. Optou-se por esse *dataset* principalmente por ser recente, visto que outros *datasets* mais conhecidos na literatura são em sua maioria antigos, tais como os fornecidos pela DARPA [64]. Por ser atual, esse conjunto de dados representa o funcionamento do tráfego de rede moderno mais eficientemente que àqueles gerados por volta do ano de 1999.

O conjunto de dados em questão consiste de uma ampla rede local (*Local Area Network*) militar simulada. O objetivo desta simulação era parecer com um ambiente real das forças aéreas dos Estados Unidos, de modo a coletar informações brutas de tráfego TCP/IP. As conexões armazenadas nos arquivos disponibilizados são uma sequência de pacotes que se inicia e termina em algum ponto dentro do intervalo de tempo coberto no experimento. As conexões descrevem origem e destino do tráfego, e contém diversos tipos de tráfego e protocolos de camada 4 e 5, conforme explicado na sessão 2.1. A tabela 4.1 contém informações sobre os protocolos contidos no *dataset*.

Protocolos de transporte	Protocolos de Aplicação
	HTTP
	TELNET
TCP	POP3
	SMTP
	PRIVATE
UDP	PRIVATE

Tabela 4.1: Principais protocolos e aplicações contidos no dataset *Network Intrusion Detection*.

Conforme pode ser visto na tabela, a maior parte dos protocolos da camada de aplicação utilizam como protocolo de transporte o TCP. Há, porém, casos em que a aplicação não especificou qual protocolo estava sendo utilizado e por isso o conjunto de dados o classificou como privado (*private*). Os protocolos privados foram utilizados tanto em aplicações que rodam em TCP quanto UDP.

Cada conexão contida no *dataset* em questão está classificada ou como normal, caso esteja se tratando de tráfego legítimo, ou como um ataque, podendo ser quatro tipos distintos. Por fim, cada linha dos registros consiste de aproximadamente 100 *bytes*.

Os quatro tipos de ataques presentes neste *dataset* são: dos, probe, u2r e r2l. A seguir, é explicado o funcionamento básico de cada um deles.

- **dos - Denial of Service:** Os ataques de negação de serviço – DoS – consistem em exaurir os recursos de um determinado servidor de maneira que novos usuários legítimos não sejam capazes de abrir novas conexões [8]. Devido ao fato desses ataques se originarem de servidores remotos, localizar a origem de tais ataques pode ser complexo. É possível também que esse ataque seja realizado simultaneamente de diversas origens distintas, caracterizando um ataque de negação de serviço distribuído – o DDoS. A figura 4.1 ilustra uma simplificação da versão distribuída desse ataque.

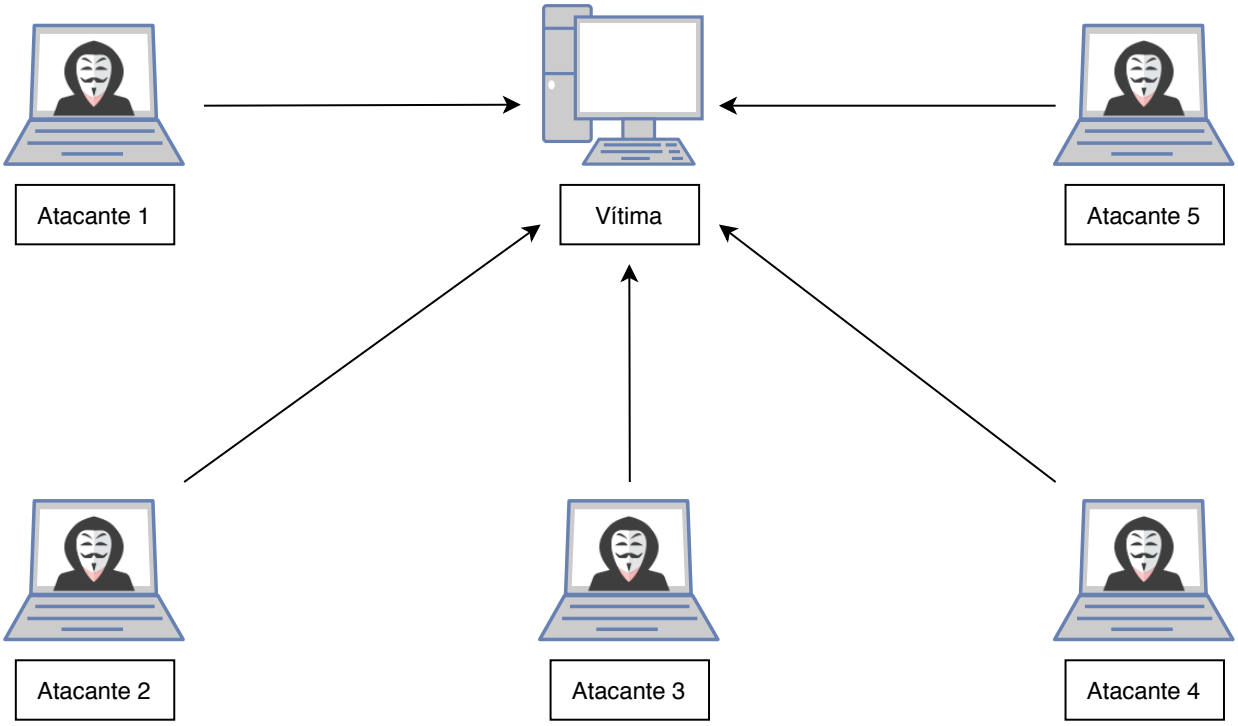

Figura 4.1: Simplificação de um ataque DDoS. Adaptado de [8]

Caso um usuário tente iniciar uma conexão nova para com o servidor alvo dos ataques, esta não será completada pois todos os recursos do sistema operacional que tratam novas conexões estão sendo utilizados pelos atacantes.

- **probe:** Os ataques do tipo *probe* consistem em um conjunto de ações por parte dos atacantes para fazer o reconhecimento de um determinado segmento de rede a ser atacado [65]. São realizados diversos tipos de varredura, com o intuito de descobrir informações sobre as máquinas conectadas e quais as possíveis vulnerabilidades que elas possam apresentar. Uma das formas mais comuns de executar esse ataque é realizando varreduras de ping, em que diversos pacotes do protocolo ICMP são disparados para todos os endereços IP disponíveis em determinada faixa de rede e os computadores que responderem ao pacote podem ser vítimas em potencial que o atacante descobriu. Feito isso, um segundo tipo de varredura é executado, em que as portas abertas de cada computador descoberto no passo anterior são verificadas para descobrir quais serviços estão ativos no sistema de destino e assim utilizar essa informação para explorar vulnerabilidades inerentes a processos específicos.

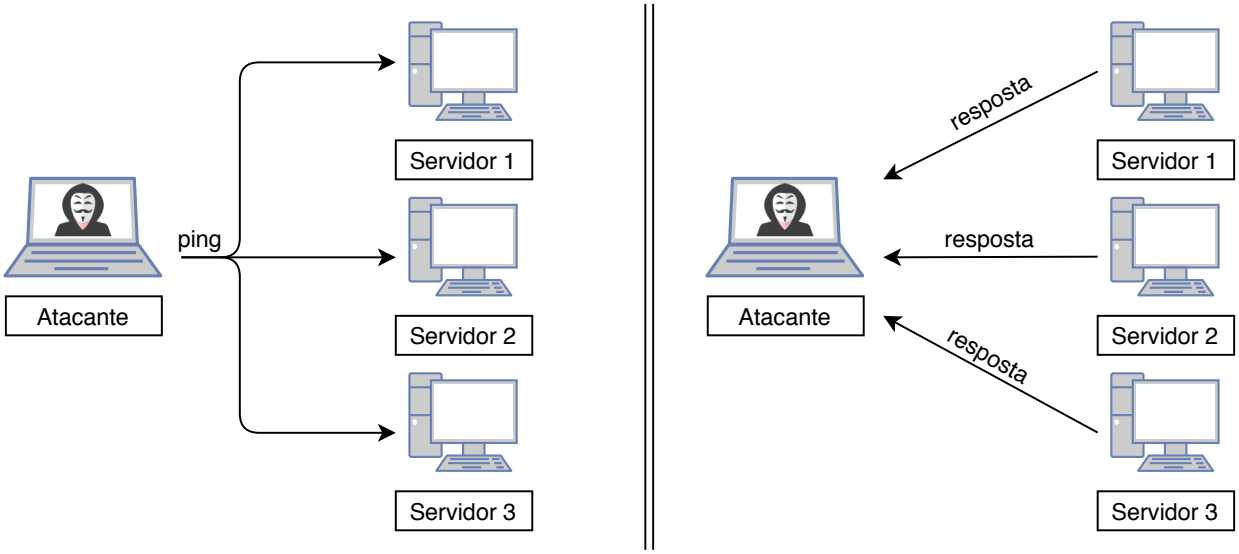

Figura 4.2: Simplificação de um ataque probe. [Fonte: Autor]

A figura 4.2 ilustra o processo inicial de reconhecimento de um atacante. Nesse momento, para descobrir quais computadores se encontram nessa faixa de IP, ele dispara pacotes de ICMP para todos os endereços disponíveis e, com base na resposta, sabe quantos alvos há naquele segmento de rede.

- **u2r - User to root:** Em ataques do tipo u2r, um processo do sistema operacional é comprometido por atacantes, mas não possui ainda permissões de super-usuário. Dessa forma, esse processo está limitado ao nível de privilégios associados à conta de usuário que o está executando [66]. Para conseguir aproveitar por completo os recursos disponibilizados pelo alvo do ataque, o processo tenta elevar o usuário de execução para *root*, podendo assim executar quaiquer comandos no sistema operacional.

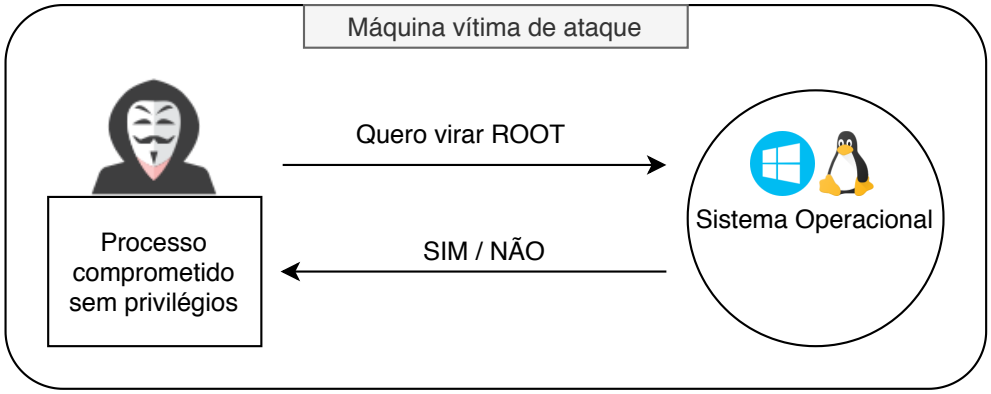

Figura 4.3: Exemplo de um ataque U2R. [Fonte: Autor]

Conforme mostrado na figura 4.3, a tarefa de obter as permissões de super usuário não é simples, visto que o sistema operacional normalmente não permite que usuários comuns do sistema virem *root* sem algum processo de autenticação antes, normalmente uma senha.

- **r2l - Root to local:** Ataques do tipo r2l consistem em uma máquina comprometida por atacantes lançando ataques em direção à outro computador no mesmo segmento de rede, tentando comprometê-lo também [66]. Por via de regra, o processo que assume o controle dos ataques na máquina comprometida possui privilégios de super ussário, visto que diversos recursos do sistema operacional de camada baixa precisam ser utilizados para enviar ataques, tais como *sockets*. A figura 4.4 ilustra o funcionamento de um ataque r2l.

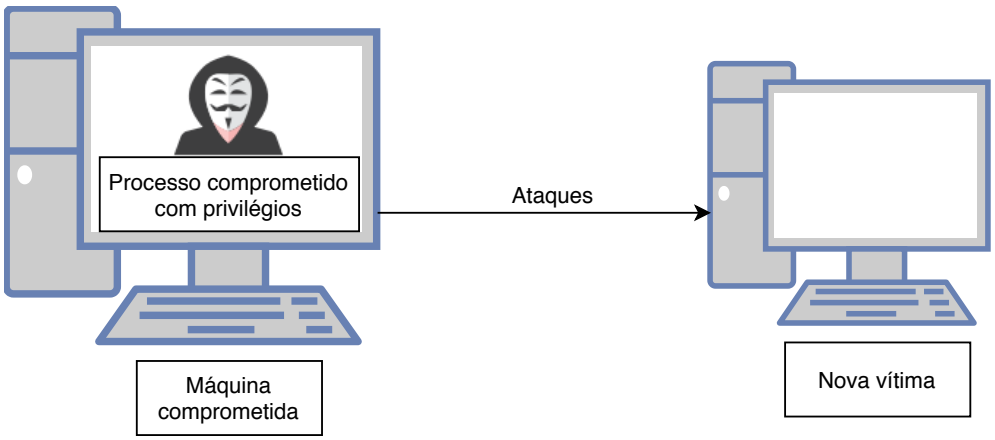

Figura 4.4: Exemplo de ataque R2L. [Fonte: Autor]

Um fato importante de ser destacado é a natureza destes ataques explicados previamente. Dentre eles, dos, *probe* e r2l utilizam o tráfego de rede para realizar as ações necessárias, ao passo que o ataque u2r fica contido dentro do sistema operacional comprometido. Análises que utilizam apenas parâmetros de rede para identificar padrões de ataques, de modo geral, tendem a ser menos eficientes para classificar ataques que ocorrem exclusivamente dentro do sistema.

O *dataset Intrusion Detection* é dividido em dois arquivos distintos: um arquivo CSV contendo aproximadamente 126 mil exemplos para ser usado como conjunto de treinamento e outro, de mesma extensão, com cerca de 10 mil registros para servir de conjunto de teste. Ambos os arquivos disponibilizados possuem 42 campos distintos, mostrados na tabela 4.2.

Intrusion Detection		
duration	su_attempted	same_srv_rate
protocol_type	num_root	diff_srv_rate
service	num_file_creations	srv_diff_host_rate
flag	num_shells	dst_host_count
src_bytes	num_access_files	dst_host_srv_count
dst_bytes	num_outbound_cmds	dst_host_same_srv_rate
land	is_host_login	dst_host_diff_srv_rate
wrong_fragment	is_guest_login	dst_host_same_src_port_rate
urgent	count	dst_host_srv_diff_host_rate
hot	srv_count	dst_host_serror_rate
num_failed_logins	serror_rate	dst_host_srv_serror_rate
logged_in	srv_serror_rate	dst_host_rerror_rate
num_compromised	rerror_rate	dst_host_srv_rerror_rate
root_shell	srv_rerror_rate	xAttack

Tabela 4.2: Campos do *dataset* Intrusion Detection [9].

É possível perceber a partir dos campos mostrados a presença de dois grandes grupos de atributos: aqueles inerentes ao sistema operacional da máquina onde foi extraído, tais como *root_shells*, *su_attempted*, *num_access_files*, *is_guest_login* e *num_failed_logins*; e os que contém informações sobre o tráfego de rede, tais como *dst_bytes*, *protocol_type*, *service* e *dst_host_count*. Há também a presença do campo xAttack, que não faz parte do tráfego em si, mas contém a classificação da linha como sendo normal ou um dos 4 tipos de ataques explicados.

Foi necessário realizar uma análise qualitativa dos campos fornecidos e filtrá-los para deixar o tráfego mais similar às informações extraídas pelo método *online*. Dessa forma, foram selecionados majoritariamente apenas os campos que contém informações referentes ao tráfego de rede, visto que apenas esse tipo de informação é extraída ao se analisar as interfaces de rede de um computador. Os atributos que foram preservados são mostrados na tabela 4.3

Intrusion Detection (atributos de rede)		
duration	serror_rate	dst_host_srv_count
protocol_type	srv_serror_rate	dst_host_same_srv_rate
service	rerror_rate	dst_host_diff_srv_rate
flag	srv_rerror_rate	dst_host_same_src_port_rate
src_bytes	same_srv_rate	dst_host_srv_diff_host_rate
dst_bytes	diff_srv_rate	dst_host_serror_rate
count	srv_diff_host_rate	dst_host_srv_serror_rate
srv_count	dst_host_count	dst_host_rerror_rate
dst_host_srv_rerror_rate		

Tabela 4.3: Atributos de rede do *dataset* Intrusion Detection.

A distribuição das classes contidas no dataset de treinamento é mostrada na tabela 4.4. É possível perceber que, individualmente, as quatro classes de ataques compõem frações do *dataset* significativamente menores que o tráfego com classificação normal. Porém, considerando apenas a classificação binária de ser ou não um ataque, essa distribuição se torna mais similar, sendo 53,45% normal e 46,55% ataques.

Classificação	Qte de mensagens	Fração do dataset
normal	67343	53,45%
dos	45927	36,45%
probe	11656	9,25%
u2r	52	0,05%
r2l	995	0,80%

Tabela 4.4: Distribuição das classes de treinamento no *dataset* Intrusion Detection.

Por sua vez, a distribuição das classificações no *dataset* de testes é mostrada na tabela 4.5. Novamente, a distribuição dos ataques se prova menor que a de trafego legítimo. Porém, ao se analisar binariamente as linhas como ”ataque”ou ”normal”, tem-se uma distribuição de 43,29% para tráfego legítimo e 57,71% para ataques.

Classificação	Qte de mensagens	Fração do dataset
normal	4329	43,29%
dos	3332	33,32%
probe	1053	10,53%
u2r	87	0,87%
r2l	1199	11,99%

Tabela 4.5: Distribuição das classes de teste no *dataset* Intrusion Detection.

Como o objetivo da ferramenta é gerar modelos que representam a situação de normalidade de um determinado *host* e detectar situações que divirjam do que é considerado normal, filtrou-se o *dataset* de treinamento para que sobrassem apenas os registros de classificação ”normal”. Dessa forma, é possível concluir que o modelo gerado represente informações apenas do tráfego normal e, durante os testes, espera-se que ele seja capaz de identificar quaisquer um dos quatro tipos de ataque como sendo anomalias.

Após as etapas anteriores de seleção de atributos e filtragem dos registros de classificação normal, foi necessário definir o número de *clusters* para caracterizar o modelo gerado. Para calcular o melhor valor desta variável, utilizou-se o método do cotovelo, explicado na seção 3.7. O resultado é mostrado na figura 4.5.

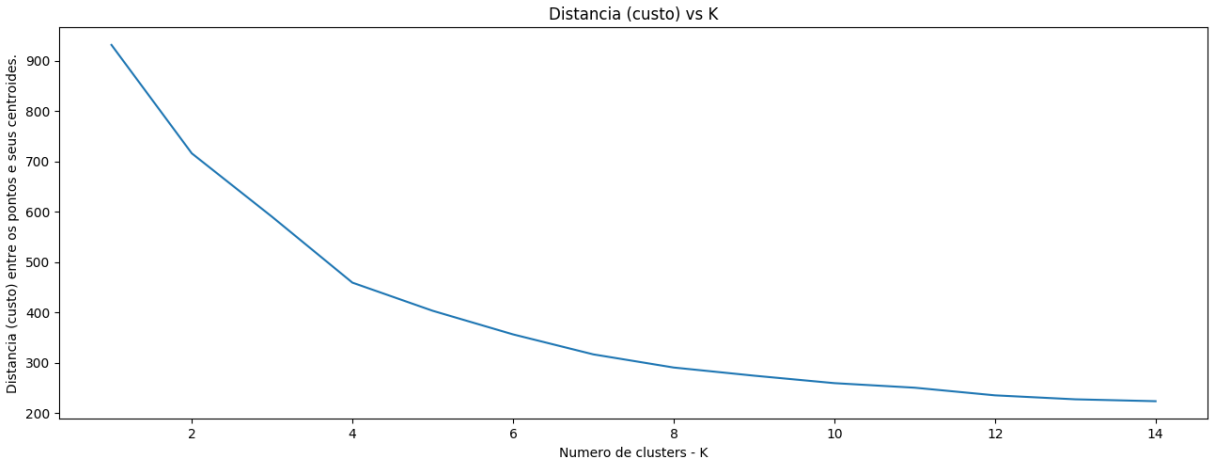

Figura 4.5: Aplicação da regra do cotovelo no experimento *offline*. [Fonte: Autor]

Conforme pode ser visualizado na figura, o número 4 é uma boa estimativa para a quantidade de *clusters* utilizados durante o treinamento do modelo.

A próxima variável a ser configurada foi o limiar de decisão, isto é, a distância máxima até o centróide de um *cluster* para que se considere que um ponto pertença àquele grupo, conforme explicado na seção 3.8. Utilizou-se o dobro da distância intracluster, isto é, a média da distância entre os pontos de um cluster e seu respectivo centróide (2σ).

Para uma simulação mais próxima do real, o produtor *offline*, explicado na seção 3.5, foi configurado para dividir o *dataset* em n partes e submeter, ao poucos, as mensagens ao serviço Kafka de modo que o módulo treinador fosse capaz de gerar vários modelos ao longo do tempo, com o preditor capaz de classificar novos blocos de mensagens por meio do modelo mais recente gerado. Uma simplificação do processo é ilustrada na figura 4.6.

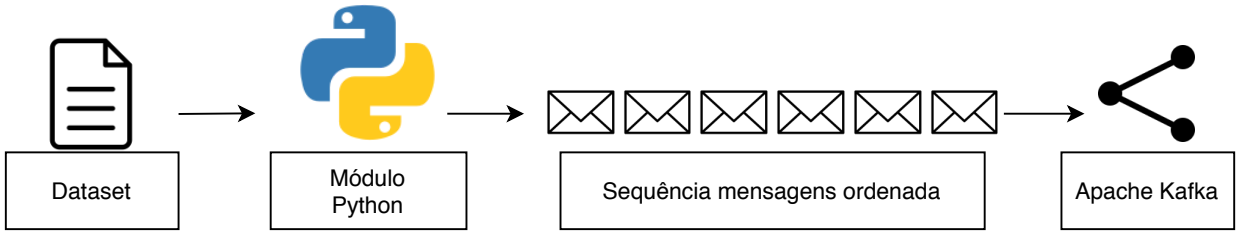

Figura 4.6: Utilização do produtor *offline* durante o experimento. [Fonte: Autor]

Para testar o desempenho da arquitetura em diversos cenários distintos, as configurações temporais de treinamento e predição foram alteradas diversas vezes, para aumentar a frequência com que essas operações são realizadas. Ao todo, foram formulados 4 cenários, em que a frequência de treinamento e predição foram dobradas entre cada um deles. A tabela 4.6 mostra a quantidade de ciclos de treinamento e predição realizados, conforme explicado nas seções 3.7 e 3.8, para cada um dos cenários testados.

Cenário	Qte de ciclos	Nº de modelos gerados	Nº de predições realizadas
1	1	1	2
2	2	2	4
3	4	4	8
4	8	8	16

Tabela 4.6: Informações sobre os cenários do experimento *offline*.

Para entender como a distribuição do *dataset* foi feita ao longo do tempo, a figura 4.7 ilustra os cenários, onde T_m^n indicam as frações do *dataset* de treinamento, P_j^k são as frações do *dataset* de predição, m_x são os modelos gerados ao longo do tempo e p_y retratam as predições realizadas nas simulações.

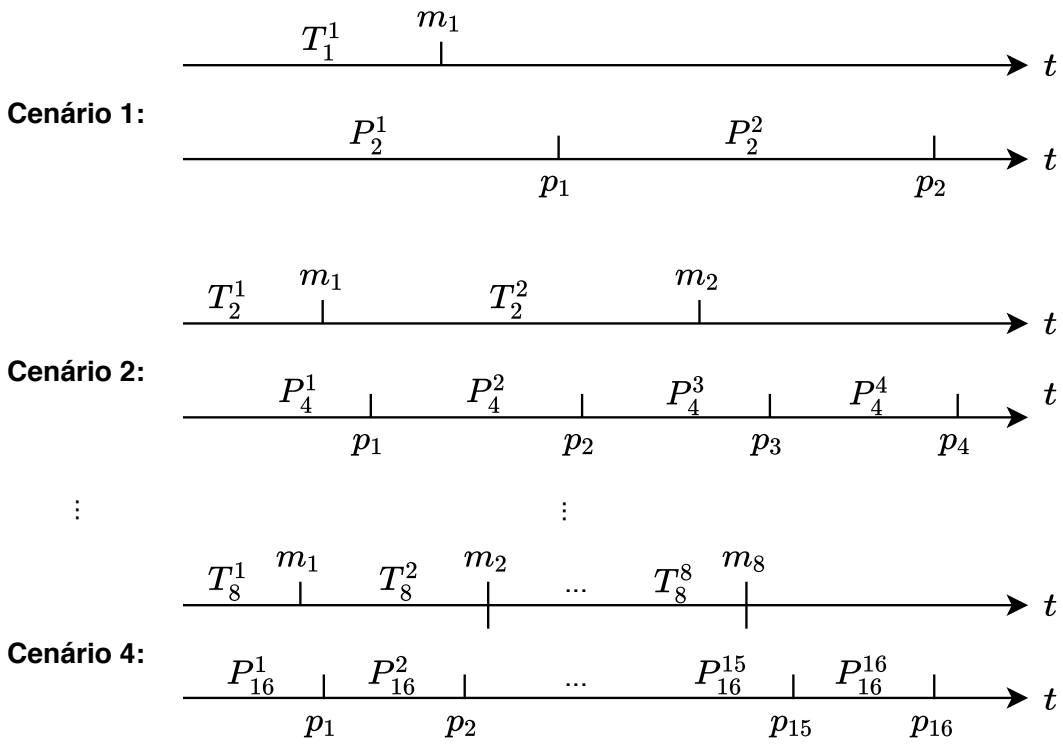

Figura 4.7: Ilustração dos cenários testados durante o experimento. [Fonte: Autor]

Conforme mostrado na figura, ao todo foram testados quatro cenários com um crescente particionamento dos *datasets*, de modo que cada ciclo de operação do Spark tivesse que processar uma quantidade menor de mensagens. Durante os testes, manteve-se a proporção de dois ciclos de predição para cada novo modelo treinado. Os resultados obtidos são mostrados nas tabelas 4.7, 4.8 e 4.9.

Cenário	Qte de ciclos	Fração do dataset por ciclo	Tempo médio	Msgs por ciclo
1	1	T/1	1600 s	67343
2	2	T/2	725 s	33671
3	4	T/4	335 s	16835
4	8	T/8	170 s	8417

Tabela 4.7: Métricas de **treinamento** para o *dataset Intrusion Detection*.

Cenário	Qte de ciclos	Fração do dataset por ciclo	Tempo médio	Msgs por ciclo
1	2	T/2	0,598 s	5000
2	4	T/4	0,419 s	2500
3	8	T/8	0,267 s	1250
4	16	T/16	0,140 s	625

Tabela 4.8: Métricas de **predição** para o *dataset Intrusion Detection*.

Primeiramente temos, nas Tabelas 4.7 e 4.8, os resultados em termos de custo computacional dos diferentes cenários para o treinamento de novos modelos e as predições, respectivamente. Como pode ser visto, o tempo de predição é bem menor que os tempos de treinamento, comprovando a vantagem para o sistema que ambas as entidades (treinador e preditor) estejam em módulos separados e independentes. Também é possível perceber que dividir a quantidade de mensagens pela metade faz com que o treinamento tenha seu tempo reduzido aproximadamente na mesma proporção. Porém, é importante garantir que o intervalo entre retreinos não seja muito curto para permitir que uma quantidade suficientemente grande de mensagens seja utilizada para gerar modelos de qualidade. Enquanto isso, diminuir o intervalo de predição mostrou-se vantajoso para a arquitetura, pois o sistema é capaz de retornar os resultados mais rapidamente.

Cenário	Acurácia média	Precisão média	Recall médio	Especificidade média	F1 Score médio
1	79,84%	87,91%	73,54%	87,62%	80,32%
2	80,34%	88,58%	74,78%	87,74%	82,46%
3	80,52%	88,96%	74,66%	87,81%	81,16%
4	81,09%	88,89%	76,15%	87,55%	82,01%

Tabela 4.9: Métricas calculadas para o experimento *offline*.

Por sua vez, a Tabela 4.9 contém os resultados em termos de desempenho na detecção, para os quatro cenários. No comparativo entre os quatro cenários, é possível perceber que não há degradação nos resultados obtidos ao se aumentar a frequência da inferência. Os resultados são discutidos mais profundamente ao final do experimento online, em que comparações com outros trabalhos também são feitas.

4.2 ANÁLISE EM REGIME ONLINE

Utilizando a análise com dados pré-rotulados, foi possível validar o funcionamento de cada componente individualmente e também da arquitetura como um todo, além de obter métricas de performance e processamento. Após analisar os resultados obtidos no final da sessão 4.1, partiu-se para validar a parte *online* da arquitetura.

Para validar o funcionamento *online* da arquitetura, analisou-se o tráfego de rede do servidor de *proxy*-reverso do laboratório pertencente ao departamento de Engenharia Elétrica da Universidade de Brasília, o LATITUDE. A arquitetura de rede simplificada pode ser visualizada na figura 4.8.

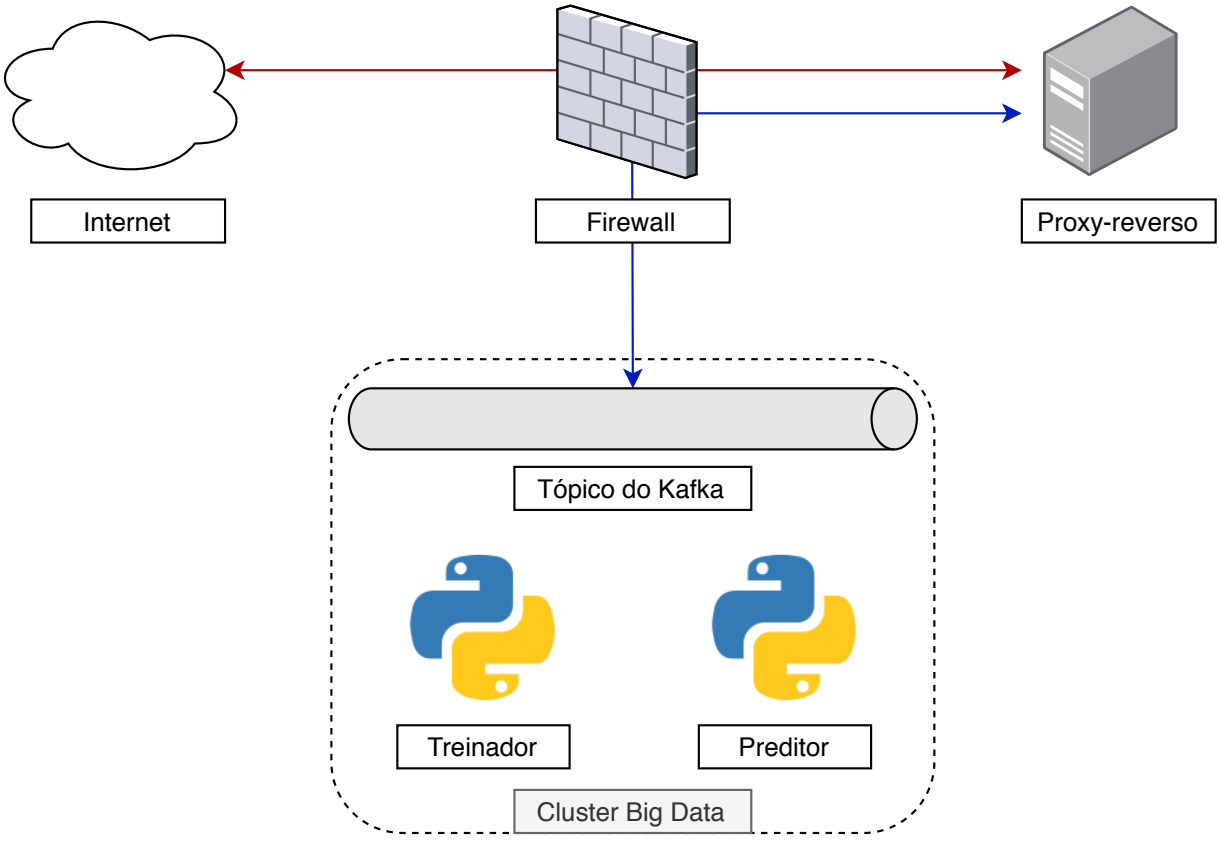

Figura 4.8: Distribuição do Firewall, *proxy*-reverso e *cluster Big Data* no LATITUDE. [Fonte: Autor]

Na figura o tráfego externo é representado pela seta vermelha (superior) ao passo que o tráfego interno à rede do laboratório é mostrado em azul (linha inferior que dobra). Graças ao *Firewall*, em momento algum o tráfego externo é capaz de chegar diretamente ao *cluster* de Big Data. De fato, o tráfego que chega ao *cluster* é uma cópia do que é capturado pela interface de rede do *proxy*. O *Big Data* se encontra em um segmento de rede diferente do servidor de *proxy* e, portanto, todo o tráfego é intermediado pelo *Firewall*, que age também como roteador neste cenário.

O *Firewall*, por sua vez, possui regras para descartar pacotes suspeitos (destinados a portas específicas que não englobam o protocolo HTTP e seus derivados), então se assume que o tráfego coletado deste servidor pode ser classificado como não-malicioso.

Conforme explicado na sessão 3.5, a biblioteca *t-shark* é utilizada para capturar o tráfego das interfaces de rede do servidor de *proxy*-reverso. Essa biblioteca é capaz de destrinchar os pacotes em todas as camadas do modelo OSI e, para cada camada, é capaz de extrair os cabeçalhos correspondentes. Dessa forma, tem-se grande flexibilidade dos parâmetros que serão utilizados para o treinamento pois potencialmente qualquer cabeçalho pode ser extraído.

Devido à maneira com que os campos são extraídos através do *t-shark*, cada campo do cabeçalho de um determinado protocolo é utilizado como *feature* durante o treinamento dos modelos. Por ser um servidor exclusivamente de *proxy*, o tráfego mais abundante que pode ser capturado nas interfaces dele é o HTTP e seu correspondente criptografado, o HTTPS. Dessa forma, escolheu-se por realizar a análise *online* com base nesses dois protocolos de camada de aplicação.

Conforme [45], os endereços IP dos pacotes e seus números de porta são boas *features* para se utilizar na classificação de tráfego de rede. Como todo o tráfego é HTTP ou HTTPS, os números de porta são, necessariamente, 80 ou 443. Portanto, não se utilizou esse campo como *feature* pois todos os pacotes teriam um entre os dois valores citados. A tabela 4.10 mostra os campos da camada de rede, explicados na seção 2.2.1, que foram escolhidos para a análise.

Cabeçalhos do protocolo IP
ip.src
ip.proto
ip.ttl
ip.hdr_len
ip.len

Tabela 4.10: Campos do cabeçalho do protocolo IP escolhidos para a análise.

É importante ter cautela na hora de escolher os cabeçalhos da camada de aplicação pois alguns têm o potencial de serem criptografados. Caso o protocolo seja HTTP, todos os campos serão capturados em texto claro e nenhuma informação será perdida. Porém, o HTTPS é construídos para funcionar com criptografia ponta-a-ponta e, dessa forma, qualquer tráfego interceptado durante o caminho vai possuir o *payload* criptografado, efetivamente impossibilitando a análise desse campo. Com base nisso, escolheu-se os campos do cabeçalho da camada de aplicação que não possuem criptografia, mesmo se capturados de pacotes HTTPS. Os campos selecionados do protocolo HTTP, explicados na seção 2.2.3, são mostrados na tabela 4.11.

Cabeçalhos do protocolo HTTP		
http.connection	http.request	http.request_number
http.content_encoding	http.request.full_uri	http.response
http.content_length	http.request.method	http.response.code
http.content_length_header	http.request.uri	http.response.phrase
http.host	http.request.version	http.response_number
http.referer	http.request_in	http.server
http.time	http.user_agent	

Tabela 4.11: Campos do cabeçalho do protocolo HTTP escolhidos para a análise.

Com base nos campos escolhidos como *features*, foi possível então realizar o método do cotovelo para determinar o melhor número de *clusters* para os modelos gerados. A figura 4.9 mostra os resultados coletados.

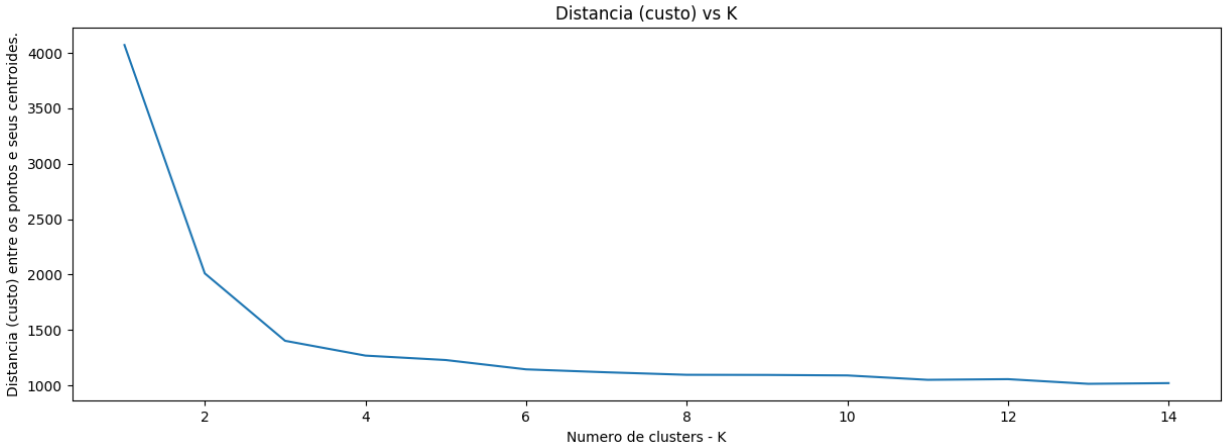

Figura 4.9: Aplicação do método do cotovelo no experimento *online*. [Fonte: Autor]

Conforme mostrado na figura 4.9, encontrou-se que o número de *clusters* é $K = 3$. Com as etapas preliminares realizadas, passou-se à execução do processo treinador, de forma a gerar novos modelos periodicamente. Com base na quantidade de tráfego recebida pelo servidor, escolheu-se o período de 1 hora para retreino, visto que em média aproximadamente 6000 pacotes eram trocados por ele neste intervalo de tempo. Com base no experimento anterior, mostrado na sessão 4.1, essa quantidade de pacotes é capaz de gerar um novo modelo em um intervalo relativamente curto ($< 170s$).

Para testar os modelos gerados pelo treinador, enviou-se tráfego capturado da *Honeynet* em conjunto com o tráfego legítimo normalmente enviado pro *proxy*. Dessa forma, foi possível submeter o *cluster* a ataques de rede sem de fato comprometer a integridade e segurança do laboratório, suas aplicações e serviços. O processo de ingestão de tráfego malicioso é mostrado na figura 4.10.

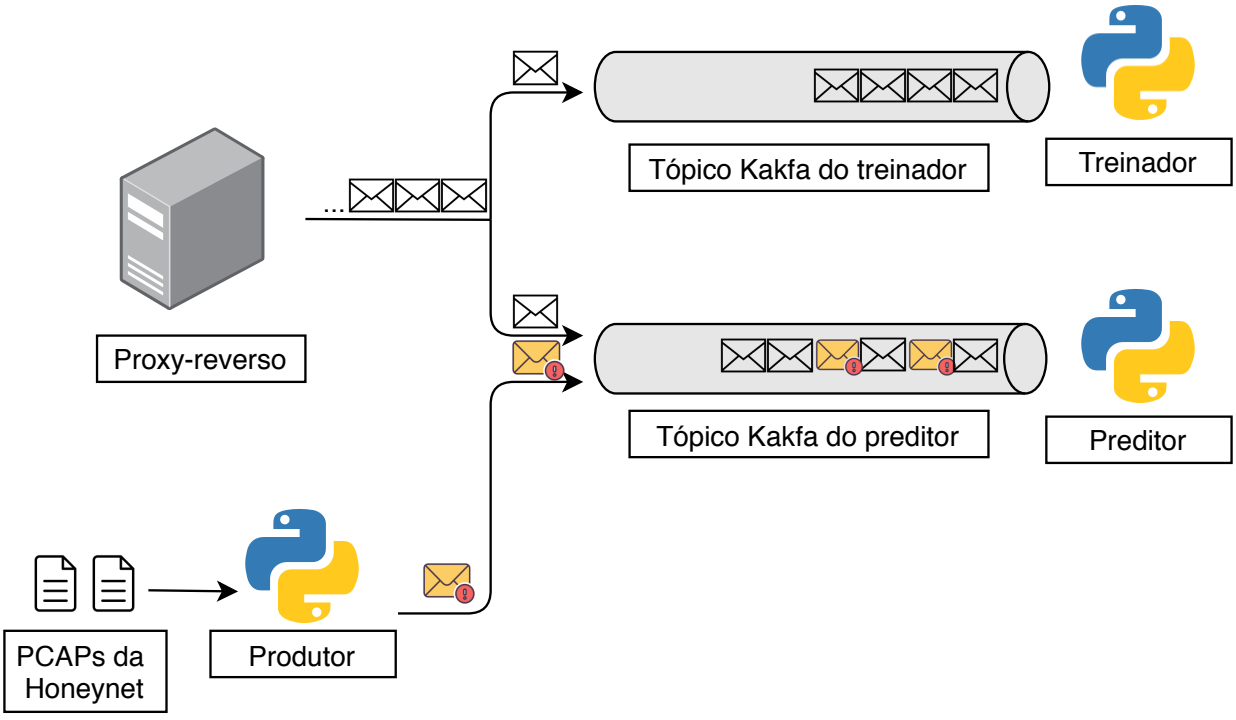

Figura 4.10: Simulação dos ataques na arquitetura. [Fonte: Autor]

Conforme pode ser visto na figura, o tráfego copiado no servidor de *proxy* é enviado simultaneamente para dois tópicos do Kafka. Um desses tópicos alimenta exclusivamente o treinador, ao passo que o outro alimenta o preditor. Dessa forma, com auxílio da ferramenta *tshark* em conjunto com um *script* Python análogo ao produtor offline, o tráfego malicioso pode ser submetido independentemente para ambos os módulos, tomando o cuidado de respeitar a variação dos timestamps originais.

Na primeira interação desta arquitetura, o treinador não recebeu nenhum pacote extraído da *Honeynet*, com o intuito de gerar o primeiro modelo contendo apenas tráfego considerado legítimo. A partir do segundo ciclo, porém, iniciou-se o processo de ingestão do tráfego malicioso nos tópicos do preditor e do treinador. Dessa forma, o preditor recebe tanto tráfego da *honeynet* quanto tráfego legítimo, e utilizará o modelo gerado anteriormente utilizando apenas o tráfego legítimo. Espera-se, com isso, que o preditor seja capaz de identificar os pacotes maliciosos e armazená-los no HBase, conforme explicado na sessão 3.8.

Ao final do segundo ciclo, o treinador também terá recebido tanto mensagens maliciosas quanto mensagens legítimas. Porém, ao iniciar o treinamento do segundo modelo, ele terá ao seu dispor a base de mensagens maliciosas atualizada pelo preditor, e ao consultá-la, é capaz de deixar as anomalias de fora do treinamento do novo modelo, para assim conseguir gerar um modelo o mais ”puro”possível.

Obteve-se, então, um ciclo de retroalimentação entre o treinador e o preditor, de modo que o treinador utiliza a base de anomalias identificada pelo preditor para filtrar os pacotes maliciosos dos novos modelos e o preditor carrega sempre os modelos mais atualizados gerados pelo treinador para conseguir identificar novas anomalias de maneira eficiente.

Conforme citado anteriormente, escolheu-se o intervalo de retreino do treinador como sendo uma hora. Para o preditor, escolheu-se metade desse período, ou seja, 30 minutos. Com isso, é possível rodar duas rodadas de predição e atualização da base de mensagens anômalas para cada novo modelo criado. Deixou-se o experimento rodar por 16 ciclos de predição, até as métricas estabilizarem, e calculou-se a média desses valores. Os resultados são compilados na tabela 4.12.

T_{treino} médio	$T_{predicao}$ médio	Acurácia média	Precisão média	Recall médio	Specifity média	F1 Score médio
71 s	1,5289 s	89,69%	73,87%	87,33%	90,30%	77,54%

Tabela 4.12: Resultados da predição no tráfego do proxy reverso.

Conforme pode ser visto na tabela, o tempo médio necessário para fazer a predição das mensagens recebidas em cada ciclo do preditor foi de 1,53 segundos. Com esse resultado, é possível reforçar a importância de manter o módulo de predição independente, visto que aumentar a frequência de execução pode gerar resultados mais rapidamente, e aproximar o funcionamento da ferramenta do tempo real. É possível perceber, então, que pode-se configurar esse módulo para rodar em intervalos muito menores que 30 minutos, por exemplo, a cada 1 minuto.

É possível perceber também que as métricas de classificação foram condizentes com aquelas encontradas no experimento anterior, apesar de serem dois *datasets* distintos. Conforme esperado, este experimento levou menos tempo em média para gerar os modelos, pois cada ciclo possuía menos mensagens que o Cenário 4 testado na Seção 4.1. É importante ressaltar que, apesar do tempo médio de treinamento dos modelos ser relativamente baixo, não é interessante diminuir a janela de retreino para aumentar a frequência de atualização do modelo, pois o volume de mensagens utilizadas para calcular os modelos tenderia a ficar baixo, o que impactaria a qualidade das predições.

4.3 COMPARAÇÕES

Com os resultados obtidos nos experimentos *offline* e *online*, descritos nas sessões 4.1 e 4.2, é possível compará-los com valores obtidos por outros trabalhos de linha de pesquisa similar. Os datasets utilizados nos experimentos citados no decorrer desta sessão não foram os mesmos utilizados neste trabalho, e portanto não podem ser utilizados para tirar conclusões diretas. A ideia dos comparativos a seguir é apresentar resultados de outras técnicas para avaliar se a proposta feita neste trabalho apresentou resultados com patamares parecidos a outros trabalhos de proposta similar.

No primeiro trabalho comparado, Syarif et al. [10] analisaram o *dataset* DARPA/Lincoln *Laboratory offline evaluation dataset* [64], mencionado na seção 4.1, com diversos algoritmos não supervisionados de clusterização e compilaram os resultados. As métricas calculadas são mostradas na tabela 4.13.

Algoritmo	Acurácia
k-Means	57.81%
improved k-Means	65.40%
k-Medoids	76.71%
EM clustering	78.06%
Distance-based outlier detection	80.15%

Tabela 4.13: Resultados obtidos por Syarif et al. [10].

De acordo com a tabela, foram testados cinco algoritmos diferentes de clusterização para analisar o *dataset* da DARPA. Vale ressaltar que, para esses algoritmos, os autores fizeram um mapeamento dos atributos categóricos para valores numéricos durante a fase de preparação para obter sucesso no processo de clusterização.

É importante ressaltar que, das cinco métricas calculadas em ambos os experimentos feitos nas sessões 4.1 e 4.2, Syarif et al. apenas calculou a acurácia dos resultados. A tabela 4.14 compara os valores mostrados na tabela 4.13 com a média das acurácias calculadas no experimento *offline* e a acurácia obtida no experimento *online*.

Algoritmo	Acurácia
k-Means	57.81%
improved k-Means	65.40%
k-Medoids	76.71%
EM clustering	78.06%
Distance-based outlier detection	80.15%
K-prototypes (experimento offline)	80.45%
K-prototypes (experimento online)	89.69%

Tabela 4.14: Comparativo entre os resultados de Syarif et al. e as métricas calculadas.

Conforme pode ser visto na tabela, as acurácias obtidas neste trabalho foram promissoras, principalmente no experimento *online*. Não obstante, mesmo no experimento *offline*, a média das acurácias obtidas nos quatro cenários testados foi maior que aquela obtida pelo algoritmo que melhor performou no trabalho de Syarif et al. para detecção de anomalias.

A segunda comparação a ser realizada é com o trabalho publicado por Zhong et al. [56]. Neste trabalho foi analisado o mesmo *dataset* da DARPA utilizado no trabalho de Syarif et al., e também foram feitos diversos testes com algoritmos de clusterização para detectar anomalias de rede.

Os autores também não processam os dados categóricos de maneira direta, mas sim fazem um mapeamento entre os atributos categóricos e valores numéricos. As métricas calculadas no trabalho são a acurácia e a precisão. O comparativo entre os valores mostrados por Zhong et al. e aqueles calculados nos experimentos das seções 4.1 e 4.2 são mostrados na tabela 4.15.

Algoritmo	Acurácia	Precisão
k-Means	92.6%	72.1%
kmo	93.7%	72.7%
SOM	92.4%	72.2%
Neural-Gas	93.5%	72%
MOSG	91.5%	71.4%
K-prototypes (experimento offline)	80,45%	88,59%
K-prototypes (experimento online)	89.69%	73.87%

Tabela 4.15: Comparativo entre os resultados de Zhong et al. e os calculados nos experimentos.

Conforme pode ser visto na tabela, Zhong et al. alcançou melhor acurácia que ambos os experimentos *online* e *offline*. Porém, a precisão calculada neste trabalho para o experimento *online* foi condizente com os valores de Zhong et al. Vale ressaltar que a precisão encontrada para o experimento *offline* foi superior a todos os outros algoritmos utilizados.

Por fim, a última comparação a ser feita é com o trabalho produzido por Gupta et al. [59]. Ao contrário de ambos os trabalhos anteriores, que eram focados em algoritmos de clusterização não supervisionados (mais especificamente, *clustering*), este trabalho foca em criar um *framework* de processamento rápido para detecção de anomalias de rede em Big Data com algoritmos supervisionados.

No trabalho de Gupta et al. há quatro métricas que também foram calculadas neste, e que podem ser diretamente comparadas, são elas: acurácia, especificidade, tempo de treinamento e tempo de predição. A tabela 4.16 mostra o comparativo desses quatro resultados.

Algoritmo	Acurácia	Especificidade	T_{treino}	T_{predio}
Logistic Regression	91.56%	98.4%	289.105 s	12.909 s
SVM	78.84%	30.48%	479.124 s	10.085 s
Naive Bayes	90.68%	57.85%	79.552 s	12.750 s
Random Forest	88.65%	63.97%	155.64 s	15.650 s
GB Tree	91.13%	99%	294.74 s	22.250 s
K-prototypes (experimento offline)	80,45%	90.30%	707.5 s	0.356 s
K-prototypes (experimento online)	89.69%	87.68%	71 s	1.529 s

Tabela 4.16: Comparativo entre os resultados de Gupta et al. e os calculados nos experimentos.

Os valores de acurácia encontrados nas sessões 4.1 e 4.2 são condizentes com os calculados por Gupta et al., visto que ficam entre o menor valor (SVM) e o maior valor (*logistic regression*). Porém, ao se comparar as especificidades de ambos os trabalhos, é possível perceber que o algoritmo K-prototypes obteve um desempenho mediano próximo a 90%, ao passo que os outros algoritmos variaram bastante nesse parâmetro.

O *dataset* utilizado por Gupta et al. também foi o da DARPA [59]. A distribuição das mensagens é ilustrada na tabela 4.17.

	Amostras para treino	%	Amostras para teste	%
Normal	972781	19.85	60593	19.48
DoS	3883390	79.27	231455	74.41
Probe	41102	00.83	4166	01.33
u2r	52	0.001	245	0.07
r2l	1106	0.02	14570	4.68
Total	4898431	100	311029	100

Tabela 4.17: Distribuição das amostras do *dataset* KBB 99, da DARPA. Adaptado de [11]

Com base nas informações contidas na tabela, é possível calcular os tempos normalizados de treinamento e predição para os cenários testados por Gupta et al. e os cenários testados neste trabalho. Para tal, dividiu-se a quantidade de mensagens pelo tempo total que os processos levaram para executar. Os resultados calculados são mostrados na tabela 4.18.

Algoritmo	T_{treino} / mensagem (s)	T_{pred} / mensagem (s)
Logistic Regression	0.00005902	0.000041504
SVM	0.000097812	0.000032425
Naive Bayes	0.00001624	0.000040993
Random Forest	0.000031773	0.000050317
GB Tree	0.00006017	0.000071537
K-prototypes (offline)	0.010505917	0.000035596
K-prototypes (online)	0.0118333	0.000127417

Tabela 4.18: Tempo normalizado de processamento das mensagens.

A partir da análise da tabela é possível perceber que o K-prototypes não escala tão bem quanto os algoritmos supervisionados que foram utilizados. Durante o experimento *online*, em que cerca de 6000 mensagens eram utilizadas para gerar novos modelos, levou-se apenas 71 segundos em média para cada um. Porém, o experimento *offline* levou, no pior dos cenários, 1600 segundos para processar aproximadamente 70000 mensagens. Esse valor é muito maior que qualquer um obtido por Gupta et al. durante seus experimentos, cujo pior algoritmo levou aproximadamente 479 segundos para gerar o modelo. É importante ressaltar que foram utilizados *hardwares* distintos durante os experimentos conduzidos e, portanto, a comparação entre os valores temporais de processamento e predição são utilizados apenas para verificar se os resultados obtidos neste trabalho estão condizentes com os resultados obtidos por Gupta et al.

Porém, em contraste com o tempo de treinamento, o tempo de predição obtido pelo K-prototypes foi condizente com os cenários testados por Gupta et al. Dessa forma, é possível ver a importância de separar o módulo de treinamento do módulo de predição, vista a tamanha discrepância de tempo que cada um leva para terminar suas respectivas execuções.

5 CONCLUSÃO

Este trabalho propôs construir uma arquitetura em Big Data capaz de detectar anomalias em tráfego de rede. A arquitetura proposta deve ser capaz de processar dados não-rotulados, com o intuito de evitar a necessidade de utilizar *datasets* classificados previamente. Ademais, tanto atributos numéricos quanto atributos categóricos devem fazer parte do escopo de processamento desse sistema. Quanto ao tempo de processamento, propôs-se que os resultados fossem gerados em *near real time*, ou seja, de maneira rápida e com baixos níveis de latência. Por fim, o último objetivo proposto da arquitetura era ela ser capaz de manter os seus modelos gerados sempre atualizados para acompanhar a constante variação no fluxo de rede, para que as predições realizadas sejam as mais acuradas possível.

O primeiro dos objetivos propostos foi concluído através da utilização de diversas ferramentas pertencentes ao ecossistema de *Big Data*. Foi utilizado o *framework* de processamento distribuído Apache Spark, que permitiu realizar as operações necessárias em memória primária dos nós do *cluster*, dando grande agilidade ao processo. O serviço de mensageria Apache Kafka permitiu que os diversos módulos desenvolvidos em Python se comunicassem de maneira transparente, sem precisar implementar arquiteturas cliente-servidor tradicionais. É interessante ressaltar que, devido à característica distribuída das ferramentas de Big Data, o sistema resultante ganhou o mesmo nível de resiliência e escalabilidade que o *cluster* possui.

Todos os componentes de Big Data utilizados e *scripts* criados foram desenvolvidos para serem modulares e permitir assim a manutenção e evolução da ferramenta por partes, sem precisar alterar todo o código fonte a cada mudança.

Ambos os objetivos de processar dados não rotulados e utilizar tanto atributos numéricos quanto atributos categóricos foram atingidos através do uso do algoritmo de *clusterização* K-prototypes. Essa técnica provou ser bastante eficiente para processar altas quantidades de tráfego de rede com baixa latência, permitindo assim que o tráfego recebido pelo Spark fosse processado rapidamente. Ademais, essa técnica, ao contrário do K-Means tradicional, permite utilizar *features* categóricas diretamente (fator relevante pois parte dos campos dos protocolos de rede contém valores textuais), fazendo com que este trabalho se destacasse de outros trabalhos já existentes de proposta similar, tais como [10], [56] e [59].

Conseguiu-se obter baixos tempos de processamento através da separação dos módulos de treinamento e predição. Dessa forma, o objetivo de realizar o processamento em *near real time* foi concluído com êxito, visto que, conforme mostrado no experimento 4.2, a arquitetura obtida foi capaz de realizar predições sobre o tráfego de rede do servidor de *proxy*-reverso do laboratório Latitude em aproximadamente 1.5 segundos.

O último dos objetivos propostos também foi cumprido, pois o módulo de treinamento é configurado para rodar de maneira periódica e, para cada nova execução, gerar um novo modelo com base no tráfego de rede capturado desde a sua última execução. Dessa forma, o módulo preditor é capaz de realizar suas análises com base em modelos atuais, evitando assim que os resultados fiquem obsoletos e não reflitam mais o estado da rede sendo analisada.

Os comparativos feitos com trabalhos de proposta similar mostraram que os resultados obtidos nesse trabalho foram promissores, e que, mesmo propondo combinar clusterização com retreinos periódicos em ferramentas de Big Data, tanto as métricas de predição quanto as de velocidade de processamento foram condizentes com tais trabalhos.

5.1 TRABALHOS FUTUROS

Trabalhos subsequentes poderão analisar outros tipos de servidores para gerar modelos baseados em outros protocolos da camada de aplicação, tais como o DNS, o Telnet e o SSH. Comparar o desempenho do K-prototypes para os diferentes protocolos citados também tem potencial de trazer resultados interessantes. Após validar o funcionamento com *datasets* coletados, pode-se também testar os modelos gerados com o auxílio de ferramentas que simulam ataques de rede reais, tais como o HPing.

Devido a modularidade obtida na arquitetura proposta, há também o potencial de implementação de novos algoritmos não supervisionados para comparar o desempenho com o K-propotypes e, potencialmente, obter melhores níveis de detecção de anomalias em redes de computadores.

Por fim, adicionar mecanismos de visualização, tais como o Kibana, para usuários pode facilitar a análise dos resultados obtidos e possivelmente levar a novos *insights*.

REFERÊNCIAS

[1] IETF, “Rfc 791 – internet protocol,” 1981. [Online]. Available: https://tools.ietf.org/html/rfc791

[2] ——, “Rfc 793 – transmission control protocol,” 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[3] ——, “Hypertext transfer protocol – http/1.1,” 1999. [Online]. Available: https://tools.ietf.org/html/rfc2616#page-31

[4] Cloudera. (2018) Introduction to spark streaming. [Online]. Available: https://www.cloudera.com/tutorials/introduction-to-spark-streaming/.html

[5] T. Point, “Hbase - overview,” 2017. [Online]. Available: https://www.tutorialspoint.com/hbase/hbase_overview.htm

[6] G. A. d. Oliveira Júnior, “Honeyselk: um ambiente para pesquisa e visualização de ataques cibernéticos em tempo real,” 2016.

[7] T. Sarkar. (2019) Clustering metrics better than the elbow-method. [Online]. Available: https://towardsdatascience.com/clustering-metrics-better-than-the-elbow-method-6926e1f723a6

[8] S. Weisman. (2020) What are denial of service (dos) attacks? dos attacks explained. [Online]. Available: https://us.norton.com/internetsecurity-emerging-threats-dos-attacks-explained.html

[9] Kaggle. (2017) Intrusion detection. [Online]. Available: https://www.kaggle.com/what0919/intrusion-detection

[10] I. Syarif, A. Prugel-Bennett, and G. Wills, “Unsupervised clustering approach for network anomaly detection,” in *International conference on networked digital technologies*. Springer, 2012, pp. 135–145.

[11] A. Özgür and H. Erdem, “A review of kdd99 dataset usage in intrusion detection and machine learning between 2010 and 2015,” *PeerJ Preprints*, vol. 4, p. e1954v1, 2016.

[12] N. T. S. C. Checkpoint. (2020) Cyber secutiry report 2020. [Online]. Available: https://www.ntsc.org/assets/pdfs/cyber-security-report-2020.pdf

[13] J. Oden, “The seven layers of networking – part i,” Boson Holdings, LLC. All rights reserved, 2013. [Online]. Available: http://blog.boson.com/bid/86999/The-Seven-Layers-of-Networking-Part-I

[14] IBM, “Network layer, layer 3,” IBM Inc.., 2010. [Online]. Available: https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.znetwork/znetwork_31.htm

[15] O. Bonaventure, “The network layer,” Computer Networking : Principles, Protocols and Practice, 2014. [Online]. Available: http://cnp3book.info.ucl.ac.be/1st/html/network/network.html

[16] G. Fairhurst, “Transport layer protocol,” 2009. [Online]. Available: http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/transport.html

[17] J. F. K. Keith W. Ross, “Transport layer services and principles,” 2000. [Online]. Available: http://www2.ic.uff.br/~michael/kr1999/3-transport/3_01-transport_layer.htm

[18] K. Acheson, “The seven layers of networking – part ii,” Boson Holdings, LLC. All rights reserved, 2014. [Online]. Available: http://blog.boson.com/bid/102793/The-Seven-Layers-of-Networking-Part-II

[19] T. Project, “Tor: Overview,” 2017. [Online]. Available: https://www.torproject.org/about/overview.html.en

[20] C. M. Kozierok, “Application layer (layer 7),” 2005. [Online]. Available: http://www.tcpipguide.com/free/t_ApplicationLayerLayer7.htm

[21] R. M. Bezerra, “A camada de aplicação,” CEFET/BA, 2008. [Online]. Available: http://www2.ufba.br/~romildo/downloads/ifba/aplicacao.pdf

[22] Intel, “Going beyond deep packet inspection (dpi) software on intel® architecture,” *Intel White Paper*, 2012.

[23] P. Podila, “Http: The protocol every web developer must know - part 1,” 2013. [Online]. Available: https://code.tutsplus.com/tutorials/http-the-protocol-every-web-developer-must-know-part-1--net-31177

[24] K. Watson, “Honeynet tutorial,” 2010. [Online]. Available: http://homes.cerias.purdue.edu/~kaw/research/honeynet/HoneynetTutorial/honeypots.html

[25] P. G. R. Jorge Guilherme Silva Santos, “Proposta de prova de conceito de rede para internet das coisas (iot),” *Universidade de Brasília*, p. 109, 2016.

[26] T. H. Project, “Know your enemy: Honeynets,” 2001. [Online]. Available: https://www.symantec.com/connect/articles/know-your-enemy-honeynets

[27] T. Point, “Big data analytics,” 2017. [Online]. Available: https://www.tutorialspoint.com/big_data_analytics/big_data_analytics_lifecycle.htm

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[29] A. R. e Jeffrey D. Ullman, *Mining of Massive Datasets*, Universidade de Stanford, 2010, 2011.

[30] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[31] D. Kovachev. (2019) A beginner’s guide to apache spark. [Online]. Available: https://towardsdatascience.com/a-beginners-guide-to-apache-spark-ff301cb4cd92

[32] A. Spark. (2019) Resilient distributed datasets (rdds). [Online]. Available: https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds

[33] E. F. Z. Santana. (2016) Introdução ao apache spark. [Online]. Available: https://www.devmedia.com.br/introducao-ao-apache-spark/34178

[34] S. Goyal. (2019) Introduction to apache spark. [Online]. Available: https://towardsdatascience.com/introduction-to-apache-spark-207a479c3001

[35] Apache, “Apache hadoop,” 2016. [Online]. Available: http://hadoop.apache.org/

[36] H. Works, “What is apache hadoop?” 2014. [Online]. Available: http://hortonworks.com/apache/hadoop/

[37] Apache, “Apache kafka for beginners,” 2016. [Online]. Available: http://blog.cloudera.com/blog/2014/09/apache-kafka-for-beginners/

[38] L. Johansson, “Part 1: Apache kafka for beginners - what is apache kafka?” Cloudekarafka, 2016. [Online]. Available: https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html

[39] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[40] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, “Anomaly-based network intrusion detection: Techniques, systems and challenges,” *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.

[41] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin, “Intrusion detection by machine learning: A review,” *expert systems with applications*, vol. 36, no. 10, pp. 11 994–12 000, 2009.

[42] L. Portnoy, “Intrusion detection with unlabeled data using clustering,” Ph.D. dissertation, Columbia University, 2000.

[43] A. Al-Masri. (2019) How does k-means clustering in machine learning work? [Online]. Available: https://towardsdatascience.com/how-does-k-means-clustering-in-machine-learning-work-fdaaaf5acfa0

[44] M. Goggin. (2018) Machine learning in construction: How clustering data can improve processes (part 2 of 2). [Online]. Available: https://towardsdatascience.com/how-does-k-means-clustering-in-machine-learning-work-fdaaaf5acfa0

[45] H. G. Kayacik, A. N. Zincir-Heywood, and M. I. Heywood, “Selecting features for intrusion detection: A feature relevance analysis on kdd 99 intrusion detection datasets,” in *Proceedings of the third annual conference on privacy, security and trust*, vol. 94, 2005, pp. 1723–1722.

[46] Z. Huang, “Clustering large data sets with mixed numeric and categorical values,” in *Proceedings of the 1st pacific-asia conference on knowledge discovery and data mining,(PAKDD)*. Singapore, 1997, pp. 21–34.

[47] ——, “Extensions to the k-means algorithm for clustering large data sets with categorical values,” *Data mining and knowledge discovery*, vol. 2, no. 3, pp. 283–304, 1998.

[48] N. J. de Vos, “Kmodes categorical clustering library,” https://github.com/nicodv/kmodes, 2015–2020.

[49] S. P. Bhuvanagiri, S. Pichika, R. Akkur, K. Chaganti, R. Madhusoodhanan, and S. V. Pusapati, “Chapter 9 - integrated approach for modeling coastal lagoons: A case for chilka lake, india,” in *Integrated Population Biology and Modeling, Part A*, ser. Handbook of Statistics, A. S. Srinivasa Rao and C. Rao, Eds. Elsevier, 2018, vol. 39, pp. 343 – 402. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169716118300105

[50] R. R. Sokal, “A statistical method for evaluating systematic relationships,” *Univ. Kansas, Sci. Bull.*, vol. 38, pp. 1409–1438, 1958.

[51] E. Bloedorn, A. D. Christiansen, W. Hill, C. Skorupka, L. M. Talbot, and J. Tivel, “Data mining for network intrusion detection: How to get started,” Citeseer, Tech. Rep., 2001.

[52] M. Ahmed, A. N. Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016.

[53] J. Jabez and B. Muthukumar, “Intrusion detection system (ids): anomaly detection using outlier detection approach,” *Procedia Computer Science*, vol. 48, pp. 338–346, 2015.

[54] K. Leung and C. Leckie, “Unsupervised anomaly detection in network intrusion detection using clusters,” in *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, 2005, pp. 333–342.

[55] G. Gu, R. Perdisci, J. Zhang, and W. Lee, “Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection,” 2008.

[56] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, “Clustering-based network intrusion detection,” *International Journal of reliability, Quality and safety Engineering*, vol. 14, no. 02, pp. 169–187, 2007.

[57] L. Wang and R. Jones, “Big data analytics for network intrusion detection: A survey,” *International Journal of Networks and Communications*, vol. 7, no. 1, pp. 24–31, 2017.

[58] S. Suthaharan, “Big data classification: Problems and challenges in network intrusion prediction with machine learning,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 4, pp. 70–73, 2014.

[59] G. P. Gupta and M. Kulariya, “A framework for fast and efficient cyber security network intrusion detection using apache spark,” *Procedia Computer Science*, vol. 93, pp. 824–831, 2016.

[60] G. A. de Oliveira Júnior, R. T. de Sousa Júnior, and D. F. Tenório, “Desenvolvimento de um ambiente honeynet virtual para aplicação governamental,” 2015.

[61] L. Chappell, *Wireshark network analysis*. Podbooks. com, Llc, 2012.

[62] K. M. M. Thein, “Apache kafka: Next generation distributed messaging system,” *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[63] N. de Vos. (2019) kmodes 0.10.1. [Online]. Available: https://pypi.org/project/kmodes/

[64] J. McHugh, “Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 262–294, 2000.

[65] V. Shmatikov and M.-H. Wang, “Security against probe-response attacks in collaborative intrusion detection,” in *Proceedings of the 2007 workshop on Large scale attack defense*, 2007, pp. 129–136.

[66] A. Alharbi, S. Alhaidari, and M. Zohdy, “Denial-of-service, probing, user to root (u2r) & remote to user (r2l) attack detection using hidden markov models,” *International Journal of Computer and Information Technology*, 2018.