1 Introdução

The Internet of Things (IoT) brings together a wide variety of devices, including mobile, wearable, consumer electronics, automotive and sensors of various types. Such devices can either be accessed by users through the Internet or connect to other devices, servers and applications, with little human intervention or supervision [TKS20, AMBB19, HDP+19, SS15]. Security and privacy is a major concern in the IoT, especially regarding devices having access to user personal data like location, health and many other sensitive data [SRB20]. Furthermore, if compromised, such devices can also be used to attack other devices and systems, steal information, cause immediate physical damage or perform various other malicious acts [KKSV17]. As an additional concern, IoT devices likely have a long lifespan, less frequent software patches, growing diversity of technologies combined with lack of control over the software and hardware of such devices by the host organization (where they are deployed), which considerably increases the attack surface.

Because most IoT devices have limited resources (i.e., battery, processing, memory and bandwidth), configurable and expensive algorithm-based security techniques are not usual, giving way to network based approaches [ZCDV17]. Machine Learning (ML) techniques, for instance, have been studied for years to detect attacks from known patterns or to discover new attacks at an early stage [BG16, MC14]. A recent survey [TKS20] shows that ML based methods are a promising alternative which can provide potential security tools for the IoT network making them more reliable and accessible than before.

Despite the promising use of ML to secure IoT systems, studies found in the literature [BG16, MC14, TKS20] are limited to traditional ML methods that use static models of traffic behavior. Most existing ML solutions for network-based intrusion detection cannot maintain their reliability over time when facing evolving attacks [VSBN19, LDP19]. Unlike traditional methods, stream mining algorithms can be applied to intrusion detection with several advantages, such as:

(*i*) processing traffic data with a single read;

(*ii*) working with limited memory (allowing the implementation in small devices commonly employed in edge services);

(*iii*) producing real-time response; and

(*iv*) detecting novelty and changes in concepts already learned.

Given the recent [VSBN19, LDP19, dCPL+19] use of Data Stream Novelty Detection (DNFD) in network data streams, this paper shows the effects of adapting these mechanisms to edge services for use in IoT environments. Our proposal, called sistema M-FOG, adapted the IDSA-IoT architecture [CSDB19] using the DNFD algorithm MINAS [FGC13, FPdLFCG15], making it suitable to run on a distributed system composed of small devices with limited resources on the edge of the network. Using our newer version of the MINAS algorithm, we have experimentally evaluated how the distribution affects the capability to detect changes (novelty) in traffic patterns and its impact on the computational efficiency. Finally, some distribution strategies and policies for the data stream novelty detection system are discussed.

A Internet das Coisas (*Internet of Things* - IoT) é um sistema global de dispositivos (máquinas, objetos físicos ou virtuais, sensores, atuadores e pessoas) com capacidade de comunicação pela Internet, sem depender de interação com interface humano-computador tradicional. Outra característica de dispositivos IoT são os recursos computacionais dimensionados, para propósitos específicos que limitam a capacidade de computar outras funções além da função original do dispositivo. O número de dispositivos categorizados como IoT na última década teve crescimento sem precedentes e, proporcionalmente, cresceu o volume de dados gerados por esses dispositivos. A análise desses dados pode trazer novos conhecimentos e tem sido um tema frequentemente abordado por trabalhos de pesquisa. Contudo, além dos dados de sensores e atuadores, esses dispositivos se subvertidos, podem gerar tráfego maligno , como o gerado pela *botnet* mirai em 2016 [KKS17]. Nesse cenário, fatores que podem favorecer a subversão dos dispositivos incluem a falta de controle sobre a origem do hardware e software embarcado nos dispositivos, além da falta das cruciais atualizações de segurança.

Com milhares de dispositivos em redes distantes gerando dados (diretamente ligados às suas funções originais ou metadados produzidos como subproduto) em volumes e velocidades consideráveis, formando fluxos contínuos de dados (*Data Stream* - DS), técnicas de mineração de fluxos

de dados (*Data Stream Mining*) são amplamente necessárias. Nesses cenários, essas técnicas são aplicadas, por exemplo, em problemas de monitoramento e classificação de valores originários de sensores para tomada de decisão tanto em nível micro, como na modificação de atuadores remotos, ou macro, na otimização de processos industriais. Analogamente, as mesmas técnicas de classificação podem ser aplicadas para os metadados gerados pela comunicação entre esses nós e a Internet, detectando alterações nos padrões de comunicação num serviço de detecção de intrusão (*Network Intrusion Detection System*, NIDS).

Técnicas de Mineração de Fluxo de Dados (*Data Stream Mining*) envolvem mineração de dados (*Data Mining*), aprendizado de máquina (*Machine Learning*) e, dentro destes tópicos, detecção de novidades (*Novelty Detection*, DNFD). Dentre as técnicas de mineração de fluxo de dados, classificadores podem ser utilizados para detectar padrões conhecidos e, em conjunto com algoritmos de detecção de novidades ou detecção de anomalias, detectar novos padrões. Essa capacidade é relevante em especial para o exemplo de detecção de intrusão, onde novidades na rede podem distinguir novas funcionalidades (entregues aos dispositivos após sua implantação em campo) de ataques por agentes externos, sem assinaturas existentes em bancos de dados de ataques conhecidos.

Análises como *Data Stream Mining* e DNFD são geralmente implementadas sobre o paradigma de computação na nuvem (*Cloud Computing*) e, recentemente, sobre paradigmas como computação em névoa (*Fog Computing*). Para *fog*, além dos recursos em *cloud*, são explorados os recursos distribuídos pela rede desde o nó remoto até a *cloud*. Processos que dependem desses recursos são distribuídos de acordo com características como sensibilidade à latência, privacidade, consumo computacional ou energético.

1.1 Motivação

Um problema recente que une, em um único contexto, os métodos de computação em névoa, processamento de fluxo de dados e detecção de novidades nesses fluxos é a detecção de intrusão em redes de dispositivos IoT. Para tratar esse problema, a arquitetura IDSA-IoT, recentemente proposta por [CSDB19], aplica ao problema algoritmos atuais de detecção de novidades em fluxos, executando esses algoritmos em ambiente próximo aos dispositivos e avaliando-os quanto à detecção de intrusão.

Na arquitetura proposta, [CSDB19] avaliou os algoritmos ECSMiner [MGK+11], AnyNovel [AGSK16] e MINAS [FPdLFCG15], sendo que o último mostrou resultados promissores. A arquitetura proposta foi avaliada com o conjunto de dados (*data set*) *Kyoto 2006+*, composto de dados coletados de 348 *Honeypots* (máquinas isoladas equipadas com diversos softwares com vulnerabilidades conhecidas expostas à Internet com propósito de atrair ataques) de 2006 até dezembro 2015. O *data set Kyoto 2006+* contém 24 atributos, 3 etiquetas atribuídas por detectores de intrusão comerciais e uma etiqueta distinguindo o tráfego entre normal, ataque conhecido e ataque desconhecido [CSDB19].

Contudo, o algoritmo MINAS ainda não foi implementado e avaliado com paralelismo, multiprocessamento ou distribuição computacional, que são necessários para tratar fluxos de dados com grandes volumes e velocidades. O tratamento de distribuição em ambiente *fog computing* é essencial para aplicação deste algoritmo ao problema de detecção de intrusão em redes IoT, pois esta aplicação requer tempo de resposta mínimo e mínima comunicação entre nós distantes, como aqueles na borda e na nuvem. Ainda observando o algoritmo MINAS, destaca-se a possível divisão em três partes semi-independentes, sendo elas treinamento, classificação e detecção de novidade; a classificação é o elemento central cujos resultados são utilizados para a identificação de intrusões.

Ainda no contexto de DNFD como método de detecção de intrusão, outras propostas tratam do caso de fluxos com grandes volumes e velocidades, como é o caso de [VSBN19], que apresenta o *BigFlow* no intuito de detectar intrusão em redes do tipo *10 Gigabit Ethernet*, que podem produzir um volume considerável, atualmente impossível de ser processado em um único núcleo de processador (*single-threaded*). Essa implementação foi feita sobre uma plataforma distribuída processadora de fluxos (*Apache Flink*) executada em um cluster com até 10 nós de trabalho, cada um com 4 núcleos de processamento, totalizando 40 núcleos, para atingir taxas de até $10,72$ *Gbps*.

Os trabalhos de [CSDB19] e [VSBN19] abordam detecção de intrusão em redes utilizando algoritmos de ND em DS, porém com perspectivas diferentes. O primeiro investiga *IoT* e processamento em *fog* e baseia-se em um algoritmo genérico de detecção de novidade. O segundo trabalho trata de *backbones* e processamento em *cloud* e implementa o próprio algoritmo de detecção de novidade. Essas diferenças deixam uma lacuna onde, de um lado, tem-se uma ar-

quitetura mais adequada para o ambiente *fog* com um algoritmo estado da arte de detecção de novidades, porém sem paralelismo e. Do outro lado da lacuna, tem-se um sistema escalável de alto desempenho porém almejando outro ambiente (*cloud*) e com um algoritmo menos preparado para os desafios de detecção de novidades.

1.2 Objetivos

Como estabelecido na Seção 1.1, a lacuna no estado da arte observada é a ausência de uma implementação de algoritmo de detecção de novidades que trate adequadamente os desafios de fluxo de dados contínuos (como volume e velocidade do fluxo, evolução e mudança de conceito) e considere o ambiente de computação em névoa aplicada à detecção de intrusão. Seguindo a comparação entre algoritmos desse gênero realizada por [CSDB19], esta pesquisa escolheu investigar o algoritmo MINAS [FPdLFCG15] para receber o tratamento necessário para adequá-lo ao ambiente de névoa e para fluxos de grandes volumes e velocidades.

Portanto, seguindo os trabalhos do Grupo de Sistemas Distribuídos e Redes (GSDR) da Universidade Federal de São Carlos (UFSCar), propõem-se a construção de uma

aplicação que implemente o algoritmo MINAS de maneira escalável e distribuível para ambientes de computação em névoa e a avaliação dessa implementação com experimentos baseados na literatura usando conjunto de dados públicos relevantes. O resultado esperado é uma implementação compatível em qualidade de classificação ao algoritmo MINAS e passível de ser distribuída em um ambiente de computação em névoa aplicado à detecção de intrusão.

Com foco no objetivo geral, alguns objetivos específicos são propostos:

- Implementar o algoritmo MINAS de maneira distribuída sobre uma plataforma de processamento distribuída de fluxos de dados;
- Avaliar a qualidade de detecção de intrusão em ambiente distribuído conforme a arquitetura IDSA-IoT;
- Avaliar o desempenho da implementação em ambiente de computação em névoa.

1.3 Proposta Metodológica

Para cumprir os objetivos citados na Seção 1.2, foi identificada a necessidade de um processo exploratório seguido de experimentação. Tal processo inclui a revisão da literatura, tanto acadêmica quanto técnica, seguida da experimentação através de implementação de aplicação e testes.

O foco da revisão da literatura acadêmica é em trabalhos que abordem processamento de fluxos de dados, classificação de fluxo de dados, detecção de novidades em fluxo de dados e processamento distribuído de fluxo de dados. O objetivo da revisão é o estabelecimento do estado da arte desses assuntos, de forma que alguns desses trabalhos sirvam para comparações e relacionamentos. Além disso, desses trabalhos buscam-se métricas de qualidade de classificação (por exemplo, taxa de falso positivo e matriz de confusão) e métricas de escalabilidade (como taxa de mensagens por segundo e escalabilidade vertical ou horizontal).

A revisão da literatura técnica será focada em plataformas, ferramentas e técnicas para realizar a implementação proposta. Portanto, são selecionadas plataformas de processamento distribuído de DS e técnicas de aprendizado de máquina associadas a elas. Dessa revisão também serão obtidas técnicas ou ferramentas necessárias para extração das métricas de avaliação, bem como *data sets* públicos relevantes para detecção de novidades em DS.

Uma vez definidos o estado da arte, as ferramentas técnicas e os *data sets*, o passo seguinte é a experimentação. Nesse passo, será desenvolvida uma aplicação na plataforma escolhida que, com base no algoritmo MINAS [FPdLFCG15], irá classificar e detectar novidades em DS. Também nesse passo, a implementação será validada comparando os resultados de classificação obtidos com os resultados de classificação do algoritmo original MINAS. Posteriormente, serão realizados experimentos com a implementação e variações em *data sets* e cenários de distribuição em *fog*, coletando as métricas de classificação e escalabilidade.

Ao final, a aplicação, resultados, comparações e discussões serão publicados nos meios e formatos adequados, como repositórios técnicos, eventos ou revistas acadêmicas.

1.4 Organização do trabalho

O restante desse trabalho segue a estrutura: Capítulo 2 aborda conceitos teóricos e técnicos que embasam esse trabalho; Capítulo 3 enumera e discute trabalhos relacionados e estabelece o estado da arte do tema detecção de novidade em fluxos de dados e seu processamento; Capítulo 4 descreve a proposta de implementação, discute as escolhas de plataformas e resultados esperados. Também são discutidos no Capítulo 5 os desafios e resultados preliminares encontrados durante o desenvolvimento do trabalho. Capítulo 6 Capítulo **??** adiciona considerações gerais e apresenta o plano de trabalho e cronograma até a defesa do mestrado.

2 Fundamentos Científicos e Tecnológicos

Este Capítulo aborda conceitos que embasam esse trabalho, conceitos teóricos de ambientes e arquiteturas de computação distribuída e detecção de novidade e conceitos técnicos, como plataformas de processamento distribuído de fluxo de dados e o algoritmo MINAS.

2.1 Ambientes de Computação Distribuída

Esta Seção relaciona três ambientes de computação distribuída habitualmente utilizados para o processamento de dados massivos relacionados a redes de dispositivos IoT, entre outras aplicações. A computação em nuvem (*cloud computing*) é aplicada a vários problemas e neste trabalho seu papel em sistemas IoT é fornecer vastos recursos e garantias e em que dispositivos enviam todos dados relevantes ao sistema. O segundo e terceiro ambiente são computação de borda (*edge computing*) e a computação em névoa (*fog computing*), que utiliza os recursos computacionais distribuídos presentes em nós localizados entre os dispositivos de borda e a nuvem, com diversas intenções, desde privacidade até redução de latência.

A computação em nuvem (*cloud computing*), ou simplesmente nuvem (*cloud*), habilita o acesso através da rede a um grupo compartilhado de recursos de computação configuráveis, como servidores, redes, aplicações, armazenamento, etc. Tais recursos podem ser provisionados ou liberados sob demanda rapidamente com o mínimo esforço de gerenciamento e mínima interação com o provedor destes recursos [MG12].

As principais características do ambiente *cloud computing*, segundo [MG12] são: Serviço sob Demanda, Amplo acesso à rede, Agrupamento de recursos, Elasticidade e Serviço mensurado. Segundo, [MG12], a implantação da Computação em Nuvem pode ocorrer através dos seguintes modelos: privada, comunitária, pública, híbrida. Das implantações, a pública é a mais comum, sendo gerenciada e operada por um provedor de nuvem e a infraestrutura é provisionada e oferecida para uso público.

A computação de borda (*edge computing*) refere-se às tecnologias que permitem que a computação seja executada na borda da rede. Define-se borda ou *edge* como qualquer recurso de computação e de rede ao longo do caminho entre as fontes de dados e os data centers da nuvem [SCZ+16]. Na borda, é possível fazer armazenamento, processamento e descarregamento de dados, assim como distribuir as requisições e entregar os serviços das nuvens aos usuários. [SCZ+16] ressalta que essas capacidades (dentre outras) dos nós da borda (*edge nodes*) possibilitam que a computação de borda reduza a latência na resposta da nuvem, pré-processando os dados nos nós da borda, aproveitando melhor a banda e a transmissão de dados, e também consumindo menos recursos de computação na nuvem. Além disso, o autor ainda acrescenta que a computação de borda pode aumentar a privacidade dos dados, uma vez que eles podem ser processados no próprio dispositivo final.

A computação de borda tenta trazer a computação mais próxima das fontes de dados. Como é observado na figura, os componentes desse tipo de computação podem ser tanto produtores como consumidores, não só requisitando serviços e conteúdo da nuvem, mas também realizando tarefas da nuvem. Algumas aplicações da computação de borda incluem: análise de vídeo; em sistemas críticos para redução de latência; descarregar a nuvem de parte da computação; privacidade dos dados produzidos, mantendo-os fora de ambientes públicos; redução das cargas de dados na rede e processamento distribuído de sensoriamento massivo em cidades inteligentes [SCZ+16].

[DB16] e [IEE18] mencionam que a enorme massa de dados gerados por ambientes IoT pode ser processada em nuvem, entretanto a latência produzida pela transferência desses dados para a nuvem e o retorno do resultado pode não ser toleradas por sistemas críticos que sejam sensíveis

a latência (monitoramento de saúde e resposta a emergências). [IEE18] ainda acrescenta que enviar tantos dados à nuvem para processamento e armazenamento pode ser ineficiente e não escalável, devido à saturação de dados na rede. O ambiente *edge computing* foi proposto para trazer o processamento e armazenamento para os dispositivos de borda tentando solucionar esses problemas. Porém, dispositivos de borda comumente não podem lidar com várias aplicações IoT competindo pelos seus recursos limitados, o que poderia causar a contenção dos recursos e o aumento na latência do processamento [DB16]. Portanto, para solucionar estas questões de latência e capacidade limitada dos dispositivos de borda, a computação em névoa foi proposta.

A computação em névoa (*fog computing*) é um paradigma que distribui as capacidades de computação, armazenamento e rede entre os nós próximos das fontes dados e dos dispositivos finais, mas não necessariamente localizados na borda, dando a esses nós características de uma nuvem [BMZA12, DB16, IEE18]. Esse tipo de computação evita a sobrecarga dos dispositivos de borda. [BMZA12] e [DB16] consideram computação em névoa como complementar da computação em borda, podendo a computação em névoa aproveitar os recursos da nuvem e da borda. [IEE18] considera que a principal diferença entre esses dois tipos de computação está no número de camadas. Enquanto *edge computing* tem camadas menores, pois atua só nos dispositivos de borda, *fog computing* tem mais camadas e um modelo hierárquico, pois não atua só na camada de borda.

Segundo [BMZA12] e [DB16], as principais características da computação em névoa são:

- **Mobilidade:** é essencial que as aplicações *fog* sejam capazes de se comunicar com dispositivos móveis, por exemplo, utilizando protocolos que considerem a mobilidade dos nós;
- **Heterogeneidade:** os nós nesse tipo de paradigma possuem configurações e formatos diferentes e podem estar implantados em ambientes distintos;
- **Baixa Latência:** foi proposta para atender aplicações que requeiram baixa latência (monitoramento de saúde, jogos, realidade aumentada, etc.);
- **Distribuição geográfica:** computação em névoa pode possuir milhares de sensores e dispositivos distribuídos geograficamente, com consciência de suas localizações (*location awareness*);
- **Alto número de nós:** seguindo os ambientes IoT, a computação em névoa pode ser composta por milhares de nós;
- **Interoperabilidade e federação:** os componentes da computação em névoa devem ser capazes de interoperar, e o serviços devem ser federados ;
- **Uso de fluxo de dados e aplicações em tempo real:** a computação em névoa pode envolver aplicações que processam em lote, mas na maior parte das vezes envolve aplicações com requisito de processamento em tempo real, e para isso fazem o uso de fluxo de dados. Por exemplo, os sensores de um rede IoT escrevem a informação no fluxo de dados, a informação é processada, ações são inferidas e traduzidos em ações nos componentes atuadores.

Algumas aplicações para computação em névoa são: cidades inteligentes e semáforos inteligentes que enviam sinais de alerta aos veículos e coordenam os sinais verdes com outros semáforos através de sensores (veículos, pedestres, ciclistas); na área de saúde, para monitorar e prever situações de pacientes que estão conectados a sensores; em prédios inteligentes, que são dotados de sensores de umidade, temperatura, qualidade do ar, ocupação, sendo que a partir das informações deles, é possível alertar os ocupantes do prédio em algum caso de emergência.

2.2 Mineração de Dados e Fluxo de Dados

A Mineração de Dados é o processo de descoberta de padrões em conjuntos de dados utilizando métodos derivados de aprendizagem de máquina, estatística e banco de dados [GZK05]. Além de mineração de dados tradicional, *Big Data* trata de conjuntos de dados que não podem ser processados em tempo viável, devido a limitações como memória ou armazenamento principal.

Definição 1. *Um Fluxo de Dados S é uma sequência massiva, potencialmente ilimitada de exemplos multi-dimensionais $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n, \ldots$ recebida em instantes $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n, \ldots$ [AHWY03].*

Além da dimensão de armazenamento, outra dimensão que afeta a maneira como dados são modelados e manipulados é o tempo. Técnicas e algoritmos de mineração de fluxo de dados atendem a esses desafios utilizando restrições como apenas uma leitura do conjunto de dados e baixo tempo de processamento na construção de seus algoritmos [GR07, GZK05].

As características de fluxos de dados e mineração de dados e os requisitos de seu processamento regularmente superam as capacidades computacionais de um único nó computacional convencional, de forma que a distribuição dos requisitos em múltiplos nós computacionais em um sistema distribuído pode ser necessária [GZK05].

Computação distribuída é a área da ciência da computação que estuda sistemas em que os componentes são localizados em diferentes computadores (nós), que comunicam-se apenas por troca de mensagens e, para que o objetivo do sistema seja atingido, a cooperação entre os nós é necessária. Outras propriedades de um sistema distribuído são a concorrência entre os nós e possibilidade de falhas em partes independentes [TVS18].

Para a construção de sistemas que apliquem técnicas de mineração de fluxos de dados são necessárias bibliotecas e plataformas (*frameworks*) que são abordadas na Seção 2.3.

2.3 Arquiteturas e Plataformas de Processamento de Fluxos

Tradicionalmente, aplicações foram construídas com um sistema gerenciador de banco de dados (SGBD) relacional ou não-relacional associado. Essa arquitetura, nomeada de "arquitetura totalmente incremental" por [MW15], foi evoluída e simplificada iterativamente durante décadas de uso, porém ela não é adequada para sistemas em tempo real, como os sistema de fluxo de dados. O volume e a velocidade de dados em um *Data Stream* leva à necessidade de distribuir o processamento, acrescentando poder computacional a cada nó adicionado. Porém, desafios como comunicação eficiente e sincronização de estado entre os nós, assim como tolerância a falhas, aumentam a complexidade de construção de um sistema distribuído em relação a um sistema tradicional.

Para mitigar problemas associados à construção de sistemas *Big Data* e *Data Streams*, arquiteturas de processamento de fluxo de dados distribuído foram propostas, como a arquitetura *Lambda* [MW15] e *Kappa* [Kre14], além de diversas plataformas, tanto de *Big Data* com características de tempo real, como especializadas em fluxo de dados.

MapReduce é a primeira plataforma de processamento de conjuntos massivos de dados que atingiu uso generalizado. Nessa implementação, uma biblioteca gerencia a distribuição, paralelização, tolerância a falhas e balanceamento de carga. Ao usuário da biblioteca resta implementar duas funções: *Map*, que recebe um par ordenado $(chave, valor)$ e emite um conjunto de pares intermediários na mesma estrutura; *Reduce*, que recebe uma chave e um conjunto de valores gerado pelo agrupamento de pares com essa mesma chave [DG04].

Em prática, um *cluster MapReduce* tem centenas de processadores e o conjunto de dados é armazenado em um sistema de arquivos distribuído que é lido pela plataforma com programas escritos por usuários sendo executados sob supervisão de um nó mestre. Essa implementação tem esquema geral de processamento em lotes que não atende o requisito de baixa latência. *MapReduce* é uma das principais influências na criação da arquitetura *Lambda* [MW15].

Apache Hadoop é uma coleção de ferramentas, incluindo: *Hadoop Distributed File System* (HDFS, um sistema de arquivos distribuído), *Hadoop YARN* um gerenciador de recursos em cluster e escalonador de trabalhos e, *Hadoop MapReduce*, um sistema baseado em *YARN*, implementando o modelo *MapReduce* [Apa20b].

Apache Spark, analogamente ao *Hadoop*, é um *framework* para construção de sistemas de computação distribuída em *cluster*, com garantias de tolerância a falhas. No entanto, o modelo de processamento diverge significativamente do tradicional *MapReduce*, utilizando em lugar do HDFS um multiconjunto imutável distribuído (*Resilient Distributed Dataset* - RDD) com um escalonador de trabalhos representados por grafos acíclicos direcionados (*directed acyclic graph* - DAG), otimizador de consultas e motor de execução [Apa20c].

Uma das extensões de *Apache Spark* é *Spark Streaming*, que é um sistema de processamento de fluxo de dados escalável e tolerante a falhas [ZXW+16a, ZXW+16b]. *Spark Streaming* implementa processamento incremental de fluxo de dados usando o modelo de fluxos discretizados em que dividem-se os dados de entrada em micro-lotes (ex: a cada 100 milissegundos) e combinam-se regularmente com o estado nos RDDs para produzir novos resultados [ZXW+16a]. Essa estratégia traz benefícios sobre os sistemas de fluxos de dados distribuídos tradicionais, pois permite a

consistência e recuperação de falhas rapidamente, devido à (*RDD lineage*) e à combinação do fluxo de dados com consultas em lotes e interativas [ZXW+16b, AL18].

Apache Storm é um sistema de computação tolerante a falhas em tempo real que de fluxo de dados [Fou20, AL18]. Ao invés de executar trabalhos (*jobs*) como algumas ferramentas citadas anteriormente, *Apache Storm* . Os *jobs* eventualmente finalizam, e as topologias executam continuamente até serem finalizadas por comandos. Uma topologia constitui-se de processos trabalhadores (*workers*) sendo executados em um *cluster* de nós que são gerenciados pelo nó mestre que além de coordenar e distribuir execução, monitora falhas. Uma topologia pode ser representada por um grafo de computação direcionado acíclico (DAG).

O *Apache Flink* é uma plataforma de processamento distribuído para computação com estado gerenciado (*stateful*) sobre fluxo de dados limitados (têm início e fim) e ilimitados (não têm fim definido) [Apa20a]. Essa plataforma segue um paradigma que abrange o processamento de fluxos de dados contínuos e o processamento em lote [CKE+15, AL18]. O *Apache Flink* pode ser integrado a vários gerenciadores de *cluster* comuns, como *Hadoop Yarn*, *Apache Mesos*, e *Kubernetes*, mas também pode ser configurado para ser executado como um *cluster stand-alone*. Já o acesso programático a essa plataforma pode ser feito através das linguagens Java, Scala ou Python.

2.3.1 *OpenMPI*

MPI é um padrão com algumas implementações que permite a construção de um sistema distribuído com um executável único (monolito) utilizando com abstração a passagem de mensagens.

2.4 Detecção de Novidade

No âmbito de classificação de dados, parte da área de aprendizado de máquina, os métodos de detecção de novidade (*Novelty Detection*, DNFD) lidam com o reconhecimento e a classificação de exemplos que diferem de exemplos anteriores [Per07, GR10]. Esses métodos tratam da classificação em fluxos de dados que evoluem com o tempo, levando em consideração as características desse tipo de fluxos.

Tratando-se de fluxos de dados contínuos, são características dos padrões observados: evolução de conceito (*Concept Evolution*) em que novos padrões podem surgir; desaparecimento ou recorrência de conceito, em que padrões podem desaparecer e também podem reaparecer; mudança de conceito (*Concept Drift*, também nomeado deriva ou desvio) onde um padrão gradualmente se transforma; presença de ruído e *outliers* [GR10].

Os métodos de DNFD são aplicados a diversos problemas como detecção de intrusos [CBSB03, SdLFdCG08, VSBN19, CSDB19], detecção de falhas [ZYZH06], diagnósticos médicos [Per09], detecção de regiões de interesse em imagens [SM04], detecção de fraudes [WFYH03, AMZ16], filtros de spam [HH10] e detecção de variações comportamentais em um jogador [VFdMdC13].

Alguns métodos de DNFD utilizam tratam de novidades como uma classificação de uma ou duas classes (binariamente) onde um conceito representa a classe normal e as anomalias são representadas pela falta de conceito no modelo ou como um segundo conceito no modelo. Além da abordagem de classificação binária, múltiplos conceitos em um mesmo conjunto de dados, para isso é necessário abordar DNFD como classificação multi-classe. Alguns métodos que abordam DNFD como classificação multi-classe não atendem completamente características de conjuntos com evolução temporal, como *Concept Evolution* e *Concept Drift*, deixando de detectar múltiplos padrões que surgem simultaneamente num intervalo de avaliação [FGdCG15, GR10].

A maioria dos métodos de DNFD são construídos seguindo a abordagem de aprendizado *Offline-Online*. Essa abordagem estabelece que o método seja dividido em duas fases: a primeira fase (*Offline*) usa um conjunto de exemplos rotulados para deles extrair conceitos conhecidos e gerar um modelo; a segunda fase (*Online*) consome um conjunto ou fluxo de exemplos não rotulados e detecta padrões-novidade. Além de detectar padrões-novidade, alguns algoritmos classificam cada exemplo em um dos conceitos do modelo, ou marca o exemplo como desconhecido. Ainda na segunda fase, para atualizar o modelo, os exemplos marcados como desconhecidos são utilizados para a extração de novos conceitos ou variações em conceitos conhecidos [GR10].

Dentre os métodos de DNFD que baseiam-se em aprendizado *Offline-Online*, muitos são baseados em algoritmos de agrupamento não supervisionados, tanto para construção do modelo inicial como na extração de novos conceitos dos exemplos não explicados pelo modelo marcados como desconhecidos [SdLFdCG09, MGK+11, FGGC13].

2.4.1 O algoritmo MINAS

Um algoritmo de DNFD que tem recebido atenção nos últimos anos é o algoritmo MINAS, originalmente proposto por [FGGC13], refinado por [FPdLFCG15] e recentemente aprimorado por [dS18], com o uso de conceitos *Fuzzy*, e expandido por [Cos19], para tratar problemas multi-rótulo além dos problemas multi-classe já tratados na versão original. Esse algoritmo segue a abordagem de duas fases no modelo *Offline-Online* e usa por base algoritmos de agrupamento não supervisionados como *K-means* e *CluStream*.

O algoritmo MINAS em sua fase *Offline* consome um conjunto de treinamento contendo exemplos etiquetados. Esse conjunto de treinamento é dividido em grupos usando como chave a etiqueta, e para cada grupo de exemplos o método de agrupamento (*clustering*) é executado. O método de agrupamento objetiva resumir um conjunto maior de exemplos em um conjunto menor de *micro-clusters*.

Um *micro-cluster* é uma tupla de quatro components $(N, \mathbf{LS}, \mathbf{SS}, T)$ derivados dos exemplos representados por este *micro-cluster*, onde: N número de exemplos, **LS** soma linear dos exemplos, **SS** soma quadrada dos exemplos, T instante de chegada do último exemplo adicionado ao *micro-cluster*. Deste sumário extrai-se, entre outras estatísticas, o centro e raio que são utilizados na operação de classificação da fase *Online*. A cada *micro-cluster* é adicionada a etiqueta do grupo original e todos *micro-clusters* são arranjados em um único conjunto formando o modelo de decisão.

Na fase *Online*, listada no Algoritmo 1, o algoritmo MINAS opera com três operações: classificação de novos exemplos, detecção de padrões-novidade e atualização do modelo de decisão [FPdLFCG15]. O primeiro método é o de classificação, onde exemplos do fluxo de dados são consumidos e avaliados pelo modelo de decisão. O modelo de decisão avalia cada exemplo calculando a distância euclidiana entre o exemplo e todos *micro-clusters* do modelo, selecionando o *micro-cluster* de menor distância. Se a distância entre o exemplo e o centro do *micro-cluster* for menor que o raio do *micro-cluster*, o exemplo é classificado com a etiqueta do *micro-cluster* e o sumário estatístico do *micro-cluster* é atualizado. Caso a distância (mínima no modelo) seja maior que o raio, o exemplo é marcado como desconhecido e armazenado em conjunto próprio [FPdLFCG15].

O segundo método da fase *Online* é a detecção de padrões novidade, que é executada quando o tamanho do conjunto de desconhecidos é maior que um parâmetro predefinido. Esse método executa o agrupamento (*clustering* descrito na fase *Offline*) e valida os *micro-clusters* gerados verificando sua representatividade e coesão.

MINAS [FPdLFCG15] is an offline-online DNFD algorithm, meaning it has two distinct phases. The first phase (offline) creates an initial model set with several clusters based on a clustering algorithm with a training set. Each cluster can be associated with only one class of the problem, but each class can have many clusters.

During its online phase, which is the main focus of our work, MINAS performs three tasks in (near) real-time, in summary, classification, novelty detection, and model update tasks in a potentially infinite data stream, as shown in Algorithm 1.

MINAS attempts to classify each incoming unlabeled instance according to the current decision model. Instances not explained by the current model receive an *unknown* label and are stored in an unknowns-buffer. When the unknowns-buffer reaches a preset threshold, MINAS executes the Novelty Detection function. After a set interval, samples in the unknowns-buffer are considered to be noise or outliers and removed. The algorithm also has a mechanism to forget clusters that became obsolete and unrepresentative of the current data stream distribution, removing them from the Model and storing in a Sleep Model for possible recurring pattern detection [FPdLFCG15].

The Novelty Detection function, illustrated in Algorithm 2, groups the instances to form new clusters, and each new cluster is validated to discard the non-cohesive or unrepresentative ones. Valid clusters are analyzed to decide if they represent an extension of a known pattern or a completely new pattern. In both cases, the model absorbs the valid clusters and starts using them to classify new instances.

Para atribuição de etiquetas aos *micro-clusters* gerados, o algoritmo MINAS encontra no modelo atual o *micro-cluster* mais próximo pela distância euclidiana e classifica em dois tipos de conceito. Se a distância é menor que um parâmetro predefinido, o novo *micro-cluster* gerado recebe como etiqueta o valor de extensão de conceito conhecido. Caso contrário, se o novo *micro-cluster* está mais distante, um novo conceito foi encontrado e a etiqueta marca um padrão novidade. Após a atribuição da etiqueta do novo *micro-cluster*, ele é adicionado ao modelo de decisão.

```
   Input: ModelSet, inputStream
   Output: outputStream
   Parameters: cleaningWindow, noveltyDetectionTrigger
 1 Function MinasOnline(ModelSet, inputStream):
 2 |  UnkownSet ← ∅, ModelSleepSet ← ∅ ;
 3 |  lastCleanup ← 0 , noveltyIndex ← 0;
 4 |  foreach sample_i ∈ inputStream do
 5 |  |  nearest ← nearestCluster (sample, ModelSet);
 6 |  |  if nearest.distance < nearest.cluster.radius then
 7 |  |  |  sample.label ← nearest.cluster.label;
 8 |  |  |  nearest.cluster.lastUsed ← i ;
 9 |  |  else
10 |  |  |  sample.label ← unknown;
11 |  |  |  UnkownSet ← UnkownSet ∪ sample;
12 |  |  |  if | UnkownSet | ≥ noveltyDetectionTrigger then
13 |  |  |  |  novelties ← NoveltyDetection (ModelSet ∪ ModelSleepSet,
   |  |  |  |   *UnkownSet);
14 |  |  |  |  ModelSet ← ModelSet ∪ novelties;
15 |  |  |  |  if i > ( lastCleanup + cleaningWindow) then
16 |  |  |  |  |  ModelSet ← moveToSleep (ModelSet, *ModelSleepSet, lastCleanup);
17 |  |  |  |  |  UnkownSet ← removeOldSamples (UnkownSet, lastCleanup);
18 |  |  |  |  |  lastCleanup ← i;
19 |  |  outputStream.append(sample);
```

Algorithm 1: Our interpretation of MINAS baseado em [FPdLFCG15]

O algoritmo MINAS, como já foi discutido na Seção 2.4.1, classifica exemplos e detecta novidades em DS e considera em sua composição *concept drift* e *concept evolution*, sendo capaz de classificar como extensão de classe conhecida e identificar padrões novidade sem intervenção de especialista [FPdLFCG15]. Neste trabalho, consideram-se algoritmos derivados do algoritmo MINAS aqueles apresentados em trabalhos publicados após 2016, que estendem a implementação original seguindo sua estrutura básica.

Algoritmo Extensão FuzzyND

O algoritmo FuzzyND, derivado do MINAS foi proposto por [DSDD18]. FuzzyND incrementa o algoritmo original, aplicando a ele teorias de conjuntos *fuzzy* pela modificação da representação dos *clusters*. A modificação afeta o método de construção de *clusters*, método de classificação de exemplos e método de detecção de novidades de acordo com a nova representação.

A avaliação do algoritmo FuzzyND foi feita por meio de experimentos usando 3 *data sets* sintéticos (*MOA3*, *RBF*, *SynEDC*) e por comparação com o MINAS. O método de avaliação utili-

```
   Parameters: minExamplesPerCluster, noveltyFactor
 1 Function NoveltyDetection(Model, Unknowns):
 2 |  newModelSet ← ∅;
 3 |  foreach cl in clustering (Unknowns) do
 4 |  |  if | cl.sampleSet | ≥ minExamplesPerCluster then
 5 |  |  |  (distance, near) ← nearestCluster (cl, Model);
 6 |  |  |  if distance < near.radius × noveltyFactor then
 7 |  |  |  |  cl.label ← near.label;
 8 |  |  |  |  cl.type ← extension;
 9 |  |  |  else
10 |  |  |  |  cl.label ← noveltyIndex;
11 |  |  |  |  noveltyIndex ← noveltyIndex +1;
12 |  |  |  |  cl.type ← novelty;
13 |  |  |  Unknowns ← Unknowns − cl.sampleSet;
14 |  |  |  newModelSet ← newModelSet ∪ cl;
15 |  return newModelSet;
```

Algorithm 2: MINAS [FPdLFCG15] Novelty Detection task.

zado baseia-se na matriz de confusão incremental descrita por [FGdCG15], extraindo dessa matriz duas métricas: acurácia (*Macro F-Score*) [SL09] e taxa de desconhecidos (*UnkR*) [FPdLFCG15]. Em geral, o algoritmo FuzzyND detecta melhor novidades e, consequentemente, é mais robusto a valores atípicos (*outlier*), porém perde a capacidade de reconhecer padrões recorrentes.

Algoritmo Extensão MINAS-LC e MINAS-BR

O algoritmo MINAS-LC foi proposto por [Cos19] e trata a classificação multi-rótulo, porém não trata evoluções de conceito (*Concept Evolution*). As alterações fundamentais propostas são: a representação de *cluster* onde MINAS-LC troca a etiqueta, que era única, por uma multi-rótulo; a transformação de problema aplicada ao conjunto de treinamento para transformá-lo de um conjunto multi-rótulo para um conjunto multi-classe (simplificação) em duas variações *Label Powerset* e *Pruned Sets* com mineração de conjunto de itens frequentes.

Já o trabalho de [JFS+19], estende o algoritmo original para que classifique um exemplo com uma ou mais etiquetas usando a transformação *Binary Relevance*, o que deu origem ao algoritmo MINAS-BR. O algoritmo modifica a representação do modelo, originalmente conjunto de *clusters*, para um grupo de *clusters* por classe (etiqueta). Também modifica o método de agrupamento, substituindo a inicialização do algoritmo *K-means*, originalmente aleatória, pelo algoritmo *Leader Incremental Clustering* [VMS04].

O algoritmo MINAS-BR também é experimentalmente avaliado com 4 *data sets* sintéticos: *MOA-3C-5C-2D*, *MOA-5C-7C-2D*, *MOA-5C-7C-3* da ferramenta MOA [BHKP10] e *4CRE-V2* [1] gerados pelo método *Radial Basis Function* [SSGB15, JFS+19]. O algoritmo MINAS-BR foi comparado com 7 algoritmos da literatura também disponíveis na ferramenta MOA [BHKP10], diferente da avaliação do FuzzyND que compara diretamente com MINAS. Para análise, os 7 algoritmos foram divididos em dois grupos [JFS+19]. O primeiro grupo de 3 algoritmos com acesso às etiquetas corretas para atualização do modelo e com a técnica ADWIN (*ADaptive WINdowing*) para detectar mudanças de conceito (*Concept Drift*) O segundo grupo com os 4 algoritmos sem acesso às etiquetas corretas, ou seja, sem *feedback* externo, mesma condição do MINAS-BR [JFS+19].

A avaliação elencada por [JFS+19] leva em consideração que as classes contidas no conjunto de testes podem não ter correlação direta com os padrões identificados pelos algoritmos. Para tratar a divergência, uma estratégia baseada em proposta anterior por [FGdCG15] foi apresentada com alterações para exemplos multi-rótulo. Após associação entre padrões de novidade e classes novidade foi possível calcular métricas tradicionais. A estratégia é executada na fase de classificação seguindo as regras:

1. após o consumo do exemplo X_n;
2. para todo padrão P_i (etiqueta atribuída) identificado sem associação até o momento;
3. com classes novidade y_j (etiqueta real) presentes em exemplos antes X_n;
4. preenche-se a tabela de contingência $\mathbf{T}_{(i,j)}$ relacionando padrão P_i e classe y_j;
5. calcula-se o grau de dependência *F1* derivado da tabela de contingência $F1_{(i,j)} = f(\mathbf{T}_{(i,j)})$;
6. valores $F1_{(i,j)} = 0$ são descartados;
7. dentre os valores restantes: o padrão P_i é associado à classe y_j se $F1_{(i,j)}$ é máximo.

As métricas utilizadas por [JFS+19] após a associação de classes e padrões são as tradicionais taxa de desconhecidos (*UnkRM*) e *F1M*. Os resultados apresentados indicam que MINAS-BR capturou todas as novidades dos *data sets* sintéticos de teste e mostrou, como esperado, melhores métricas que os 4 algoritmos equivalentes da literatura ficando abaixo dos 3 com *feedback* externo.

Os trabalhos abordados nessa Seção 2.4, têm em comum, além do algoritmo base, as métricas de avaliação acurácia (*Macro F-Score* e *Macro F-Measure* F1M) e taxa de desconhecidos, aplicadas com devido tratamento. Também é comum entre eles o uso de *data sets* sintéticos. Outro potencial não explorado do MINAS é em aplicações reais, ou seja, consumindo além de *data sets* reais, fluxos realistas em ambientes simulados ou reais porém considerando uso de recursos computacionais.

Observando a arquitetura dos algoritmos abordados na Seção 2.4, nota-se as semelhanças: a fase offline centrada no processo de agrupamento e criação de modelo; a fase online dividida em

[1] A versão original do *data set* 4CRE-V2 está disponível em `https://sites.google.com/site/nonstationaryarchive/home`.

classificação (com atualização das estatísticas do modelo) e detecção de padrões, onde novamente o processo de agrupamento é central. Portanto, apesar de outros trabalhos expandirem o algoritmo com diferentes técnicas, seu núcleo continua relevante.

Propostas de modificação do algoritmo MINAS estão longe de serem exauridas. Não cabe ao presente trabalho expandir e validar conceitos de aprendizagem de máquina, porém alguns exemplos mencionados ainda não abordados são [DSDD18, dS18, JFS+19]:

a) diferentes métodos de cálculo de distância entre pontos além da distância euclidiana;

b) a mudança de representação de *clusters*, atualmente hiper-esferas [Cos19], para hiper-cubos tratando *data sets* onde as características representadas pelas dimensões são completamente independentes;

c) um modo interativo onde o *cluster* é formado, mostrado ao especialista que o classifica como inválido (ruído ou não representativo) ou válido, podendo conter uma ou mais classes e, se contiver mais que uma classe corte em grupos menores até conter somente uma classe;

d) ainda considerando interação com especialista, a possibilidade de injetar um exemplo não pertencente a uma classe, ou seja, marcar o exemplo como não pertencente a uma classe para mantê-lo na memória de desconhecidos e, eventualmente forçar criação de um *cluster* que represente uma classe geometricamente próxima mas semanticamente distinta;

e) na fase *offline* a verificação de sobreposição de *clusters* pertencentes a classes distintas e tratamento adequado.

3 Trabalhos Relacionados

Este Capítulo trata dos trabalhos relacionados e apresenta aspectos do estado da arte dos tópicos Detecção de Novidades em Fluxos de Dados, e Processamento Distribuído de Fluxos de Dados.

Nesta Capítulo, abordam-se trabalhos que aplicam em de fluxo de dados em tempo real. Um sumário dos trabalhos abordados pode ser visto na Tabela 3.1.

Tabela 3.1: Sumário dos trabalhos relacionados

Trabalho	Plataforma	Técnica	Conjunto de dados	Métricas
Ferramenta BigFlow [VSBN19]	*Python*, *flowtbag*, *Apache Kafka* e *Apache Flink*	*Hoeffding Tree*, *OzaBoosting*, *Leveraging Bag* e comitê	*MAWILab*	Acurácia (geral e por classe), Taxa de bytes
Ferramenta CATRACA [AL18]	*Virtual Network Function*, *Apache Kafka* e *Apache Spark*	PCA, SFS, e SVM-RFE	NSL-KDD, GTA/UFRJ e NetOp	Acurácia, precisão, sensibilidade e F1-score
Arquitetura IDSA-IoT [CSDB19]	Java, *Apache Kafka* e *Python*	ECSMiner, AnyNovel e MINAS	*Kyoto 2006+*	Fnew, Mnew e erro

3.1 Ferramenta BigFlow

Proposta por [VSBN19], a ferramenta BigFlow é um sistema de detecção de intrusão em rede (*Network Intrusion Detection System*, NIDS) baseado em detecção de anomalias. Duas abordagens, detecção por assinatura e detecção por anomalia, . Para a detecção de novos tipos de ataque (*zero day*), a abordagem de detecção por anomalia é vantajosa, em contraste com a abordagem de detecção por assinatura, devido ao tempo de resposta (que envolve a identificação e criação de uma assinatura), grande demais para prevenir esse tipo de intrusão.

A ferramenta BigFlow é composta pelos módulos de extração de atributos e de aprendizado confiável. O módulo de extração de atributos é responsável por coletar da rede monitorada, com estatísticas de comunicação e enviar informações desses fluxos como exemplos para o módulo de aprendizado confiável. O módulo de aprendizado confiável, é composto pelos submódulos: submódulo classificador, responsável por classificar exemplos; submódulo de verificação, responsável por verificar o resultado de classificação; submódulo de exemplos rejeitados, responsável por requisitar a um especialista etiquetas para exemplos rejeitados e; submódulo de atualização incremental, que atualiza e distribui o modelo aos classificadores.

[VSBN19] destaca que *data sets* adequados para NIDS são poucos, devido ao conjunto de qualidades que os mesmos devem atender, como realismo, validade, etiquetamento, grande variabilidade e reprodutividade (disponibilidade pública).

Para avaliar o desempenho de NIDS, o *data set* MAWIFlow é proposto por [VSBN19]. Este *data set* é derivado do *data set Packet traces from WIDE backbone, samplepoint-F*, composto por seções de captura de pacotes diárias de 15 minutos de um link de 1Gbps entre Japão e EUA, com início em 2006 continuamente até hoje, anonimizados e etiquetados por MAWILab [MAW20, FBAF10]. Desse *data set* original, o *data set* MAWIFlow utiliza apenas os eventos de 2016, dos quais 158 atributos são extraídos resultando em 7.9 TB de captura de pacotes. Além disso, os dados são estratificados para redução de seu tamanho a um centésimo, as proporções de etiquetas (Ataque e Normal), o compartilhamento e avaliação de NIDS, além de atender às qualidades anteriormente mencionadas.

Com o *data set* MAWIFlow reduzido a 62 atributos, foram avaliados quatro classificadores da literatura em dois modos de operação. O primeiro modo de operação usa somente a primeira semana do ano como conjunto de treinamento e as demais como conjunto teste. O segundo modo usa o conjunto da semana anterior como treinamento e o conjunto da semana seguinte como teste. Comparando os resultados entre os modos de operação, os autores demonstram que a qualidade da classificação reduz-se com o tempo, quando não há atualização frequente do modelo classificador.

Com base na avaliação dos classificadores da literatura, para a ferramenta BigFlow é proposta a utilização de 4 algoritmos de classificação com capacidade de atualização, sendo todos variações de árvore de decisão *Hoeffding* [VSBN19, DH00]. A avaliação da ferramenta foi executada de maneira semelhante à avaliação dos algoritmos da literatura, onde o conjunto de dados da primeira semana foi usado para treinamento e o conjunto de dados do restante do ano como conjunto de teste. Além do conjunto de treinamento, o modelo é atualizado semanalmente com base nas instâncias rejeitadas pelo submódulo de verificação.

Quanto à distribuição do processamento, a ferramenta BigFlow faz uso das plataformas *Apache Flink* e *Apache Kafka*. Em especial, destaca-se o uso do serviço gerenciador de trabalhos (*Job Manager*) e as múltiplas instâncias do serviço gerenciador de tarefas (*Task Manager*).

Em conclusão, a ferramenta BigFlow demonstra capacidade de classificação e detecção de anomalias em fluxos de dados de alta velocidade no contexto de detecção de intrusão.

3.2 Ferramenta CATRACA

O trabalho de [AL18] aborda a detecção de ameaças a redes de computadores em tempo real e, para atingir esse objetivo, propôs a ferramenta CATRACA[1]. A ferramenta CATRACA é composta de três camadas: captura, processamento e visualização.

Na camada de captura, pacotes são capturados da rede e são geradas informações sumário de fluxos por uma aplicação *Python* utilizando a biblioteca *flowtbag*[2]. Esses sumários são enviados para um tópico de um sistema de fila de mensagens (*Apache Kafka*) hospedado em nuvem. Essa aplicação *Python* é distribuída como uma função virtual de rede (*Network Function Virtualization*) executada em dispositivos de rede virtuais.

A camada de processamento consome o tópico de mensagens que contém os fluxos da camada de captura e extrai características dos fluxos, detecta e classifica ameaças, enriquece o resultado (com localização geográfica por exemplo) e envia para a próxima camada na arquitetura por meio de um banco de dados (SGBD). A última camada da ferramenta fornece uma interface gráfica que apresentada a visualização dos fluxos processados bem como os conhecimentos extraídos e armazenados no banco de dados (SGBD). Ambas as camadas de processamento e visualização são executadas em ambiente de computação em nuvem (*cloud computing*).

Para o desenvolvimento da ferramenta CATRACA, [AL18] avaliou e comparou as plataformas de processamento de fluxo de dados em tempo real disponíveis (*Apache Storm*, *Apache Flink*, *Apache Spark Streaming*). A avaliação extraiu a velocidade máxima, em mensagens por minuto, de cada plataforma, variando a configuração de paralelismo em dois programas. Ambos consumiam dados de um tópico de um sistema de fila de mensagens (*Apache Kafka*) e produziam para outro

[1] A ferramenta e sua documentação estão disponíveis em `http://gta.ufrj.br/catraca` e `https://github.com/tinchoa/catraca`.

[2] Disponível em `https://github.com/danielarndt/flowtbag` e `https://dan.arndt.ca/projects/netmate-flowcalc/`.

tópico. O primeiro programa consiste de um detector de ameças composto por uma rede neural classificadora escrito em *Java*, que foi testado com o conjunto de dados sintético UFRJ/GTA [AL18]. O segundo programa conta quantas repetições de uma palavra existem em um fluxo de dados, exemplo muito comum em tutoriais de plataformas desse gênero, e é avaliado com um conjunto de *Tweets*.

Para o modelo de classificação, a ferramenta CATRACA utiliza o método árvore de decisão, escolhido pelo rápido treinamento e pela alta precisão e acurácia[3]. O modelo é criado na fase *Offline* e utilizado na classificação binária (normal e ameaça) da fase *Online*, sendo recalculado quando uma ameaça é encontrada.

Pra avaliação da ferramenta CATRACA dois conjuntos de dados são utilizados. O primeiro conjunto, UFRJ/GTA, é sintético e foi criado por uma simulação de rede de computadores, contendo 214 200 fluxos de rede e totalizando 95GB de pacotes capturados, este *data set* é composto de 24 atributos e 16 classes. O outro conjunto, referido como NetOp, foi coletado de um operador de rede que atendia 373 residências na cidade do Rio de Janeiro em 2017. O conjunto NetOp é formado por 5 TB de pacotes capturados e etiquetados por um detector de intrusão comercial.

Também para a avaliação da ferramenta CATRACA, foram utilizadas as métricas de qualidade de classificação acurácia, precisão, sensibilidade e F1M, com intervalo de confiança de 95%. As métricas de qualidade, dependendo do tamanho do conjunto, foram extraídas por métodos de avaliação amplamente utilizados para avaliar modelos de aprendizado de máquina (*machine learning*) como validação cruzada com proporção 70% do conjunto base para treinamento e 30% para teste. Para as métricas de escalabilidade foram utilizadas a latência e fator de aceleração *speedup factor* (latência observada com paralelismo 1 dividida pela latência observada com paralelismo variável).

Em conclusão, a ferramenta CATRACA apresenta uma arquitetura dividida em camadas alocadas em ambientes de névoa (*fog computing*) e nuvem (*cloud computing*). Essa ferramenta foi avaliada com métricas de qualidade, métricas de escalabilidade e dois conjuntos de dados relevantes. No entanto, o algoritmo de detecção de anomalias desenvolvido para a ferramenta consiste de um modelo de classificação pelo método árvore de decisão e a atualização do modelo durante a fase *Online* depende de todos os exemplos do último intervalo de atualização. Esse tipo de algoritmo de detecção de anomalias de dados, como os descritos na Seção 2.4 (*Concept Drift*, *Concept Evolution*, limitado a ler o conjunto somente uma vez), que são atendidos por algoritmos de detecção de novidade.

3.3 Arquitetura IDSA-IoT

A arquitetura IDSA-IoT, proposta por [CSDB19], tem por objetivo monitorar uma rede local com dispositivos IoT e detectar tentativas de intrusão e alguma subversão do comportamento das transmissões destes dispositivos. O principal destaque da arquitetura é a distribuição de tarefas do sistema de detecção de intrusão entre nós na e nós em nuvem pública (*cloud computing*). O objetivo dessa distribuição é a redução de latência, que torna inviável a hospedagem de um sistema detector de intrusão somente em ambiente *cloud computing*, e também possibilitar a análise de grandes volumes de dados por algoritmos de maior complexidade, que são de custo computacional proibitivo para nós de borda.

A arquitetura proposta é avaliada com três algoritmos de detecção de novidade: ECSMiner [MGK+11], AnyNovel [AGSK16] e MINAS [FPdLFCG15]. A avaliação foi feita com o *data set Kyoto 2006+*, composto de dados coletados de 348 *Honeypots* (máquinas isoladas, equipadas com diversos softwares com vulnerabilidades conhecidas e expostas à Internet, com propósito de atrair ataques) de 2006 até dezembro 2015. Esse *data set* tem as características desejáveis de um conjunto para detectção de novidades como: realismo, validade, etiquetas previamente definidas, alta variabilidade, reprodutibilidade e disponibilidade pública. O *data set Kyoto 2006+* contém 24 atributos, 3 etiquetas atribuídas por detectores de intrusão comerciais e uma etiqueta distinguindo o tráfego entre normal, ataque conhecido e ataque desconhecido.

A avaliação da arquitetura foi realizada utilizando as métricas de qualidade Fnew, Mnew e erro. A métrica Fnew (ou Falso Positivo) é a fração dos exemplos de uma classe normal classificados com etiqueta novidade ou etiqueta extensão. A métrica Mnew (ou Falso Negativo) é a fração dos exemplos de uma classe novidade classificados com etiqueta normal. A métrica erro é a soma dos valores falso positivo e falso negativo dividida pelo número de exemplos classificados.

[3] A precisão e a acurácia do método árvore de decisão podem estar associadas à independência entre as características (*features*) de cada exemplo, típico de conjuntos derivados de pacotes de rede.

Além das métricas de qualidade de classificação tradicionais, também foi medida a quantidade de requisições de classificação por especialista.

Outra avaliação dos algoritmos foi a extração de métricas de uso de recursos computacionais e tempo total de processamento em dispositivos limitados. Essa avaliação envolveu dois computadores. Para tanto, um computador pessoal com recursos convencionais produzia exemplos e adicionava como mensagens em um tópico no sistema de fila de mensagens *Apache Kafka*; já o outro computador, com recursos limitados, consumia as mensagens do tópico e classificava os exemplos.

Ambas as avaliações demonstraram o equilíbrio entre qualidade de classificação e velocidade ou uso de recursos. O algoritmo ECSMiner mostrou melhor qualidade de classificação, porém com velocidade inferior e maior consumo de recursos comparado aos outros algoritmos. Já o algoritmo MINAS, apesar de maiores valores na métrica erro, mostrou-se adequado para dispositivos limitados com baixo consumo de recursos computacionais e manteve a métrica Fnew constante e baixa. O algoritmo AnyNovel não apresentou consistência nos resultados e o consumo de recursos computacionais (memória) foi elevado.

A distribuição das tarefas em serviços proposta abre oportunidades para a discussão de diferentes métodos de distribuição dessas tarefas em diferentes ambientes computacionais. Contudo, o algoritmo MINAS ainda não foi implementado e avaliado com ou , que são necessários para tratar fluxos de dados com grandes volumes e velocidades.

3.4 Conclusão

Em conclusão, os trabalhos discutidos nesse Capítulo têm temas complementares em áreas distintas. A área de aprendizado de máquina, com o tema detecção de novidades em fluxos de dados, preocupa-se em fornecer melhores previsões através de algoritmos classificadores que atendam as características de cada problema. A área de computação distribuída aborda os temas de processamento distribuído de fluxos contínuos em ambientes de computação em nuvem e em névoa, fornecendo métodos para processar grandes volume de dados com mínima latência.

Apesar de já existirem propostas que estabelecem o estado da arte separadamente em cada um dos temas, entre o estado da arte em de novidade e o estado da arte em de fluxos de dados, em especial para focado em relacionados a

4 Proposta e metodologia

Este Capítulo apresenta a proposta deste trabalho e a metodologia elegida para atingir os objetivos.

In this work, we investigate an appropriate architecture for performing DNFD at the edge, as a means of allowing small IoT devices to filter and detect undesirable network behavior. Our approach is based on the IDSA-IoT architecture [CSDB19] and DNFD techniques provide by the MINAS algorithm [FPdLFCG15]. Named sistema M-FOG, our distributed algorithm explores load balancing to enable low profile devices at the edge of the internet to also work on the classification and detection of unwanted traffic.

In this work, we propose and assess sistema M-FOG, a distributed data stream novelty detection system based on the algorithm MINAS for securing IoT networks. sistema M-FOG implements a distributed version of MINAS according to the IDSA-IoT architecture proposed in a previous work [CSDB19], to execute in the edge where small devices and constrained resources may be prevalent.

However, given the distributed nature and the typical use of small computing devices in IoT scenarios, new challenges arise:

(*i*) the classification phase of the algorithm must occur in parallel at different nodes;

(*ii*) the novelty detection phase, which provides the model evolution, must also be asynchronous;

(*iii*) the algorithm complexity (time and space) must allow it to be processed by modest computing devices (i.e., small memory and low processor performance).

NIDS monitor network traffic, and analyze the characteristics of each flow to identify any intrusion or misbehavior. However, this problem requires both fast and accurate response [dCPL$^+$19]: fast response is needed to have a proper reaction before harm can be cast to the network and to cope with the traffic without imposing loss or delay in the NIDS or observed network; accurate

response is required as not to misidentify, especially the case of false positive that leads to false alarms. To achieve those goals, we leverage fog computing.

In common IoT scenarios, data is captured by small devices and sent to the cloud for any compute or storage tasks, but this is not feasible in a NIDS scenario. Fog computing infrastructure aims to offload processing from the cloud providers by placing edge devices closer to end-users and/or data sources.

In our proposal, fog and cloud computing resources are combined to minimize the time elapsed between a flow descriptor ingestion and intrusion alarm, performing the classification step of MINAS running multiple classifier instances. After the initial classification, the resulting label can be used immediately, but if the sample is labeled as *unknown*, this sample must be stored and the novelty detection step will be triggered.

The overall sistema M-FOG architecture has two main modules, Classification and Novelty Detection, which implement the MINAS main tasks. The Classification Module performs the same task of the MINAS Online phase and is the focal point for parallelism and distribution in our proposal. It is replicated in the fog and runs on each cluster node, using a configurable number of threads (limited to the node CPU core count).

The Novelty Detection Module can also be replicated, the choice being one instance per local network, one global cloud instance, or both. This module also handles the homonymous task of MINAS Online phase, receiving all the samples labeled with *unknown*, storing them in an internal *unknown-buffer*, and, when this buffer is full, performing the MINAS Novelty Detection task (clustering followed by validation).

4.1 Polices

The design of our distributed DNFD architecture includes partitioning the functionalities of MINAS and establishing the appropriate data flows between different actors. Changes to placement and behavior can have different impacts and should be chosen with care. The decisions following these discussions can be organized in several policies, some of them were recurring during our implementation discussions and are:

- Regarding the allocation of the Novelty Detection Module:
  - At each fog node: patterns will be only detected if sufficient samples of them occur in the local observed network, use of the local node processing power, and a model synchronization mechanism between networks must be added;
  - In the cloud: detect patterns even when scattered on each local network, each sample with *unknown* label must be sent from edge to cloud implying increased internet link usage and increased delay between the appearance of a pattern, its detection and propagation to fog classifiers;
  - On both: local *unknown* buffer is maintained and novelty detection is local as well, once a sample is considered as noise or outlier it shall be sent to the cloud where the process repeats but with global data. This choice needs an even more complex model synchronization mechanism.
- Regarding the model cleanup (forget mechanism): Even when a global novelty detection is used, local models can be optimized for faster classification using the local model statistics by sorting by (or removing) least used clusters;
- Lastly, reclassification of *unknowns*: In the novelty detection task in MINAS, the *unknown* sample buffer is effectively classified using the new set of clusters. In Algorithm 2, at the line 13, the new cluster valid (novelty or extension) includes the set of samples composing that cluster, thus, if this new label assignment was put forth to the system output it would introduce delayed outputs, more recent and perhaps more accurate. Also, it would change the system data stream behavior from a *map* (meaning each input has one output) to a *flatMap* (each input can have many outputs).

A Internet das Coisas (IoT) é composta por vastas quantidades de dispositivos conectados à Internet e distribuídos geograficamente. Com capacidades diversas providas por elementos como sensores e atuadores, esses dispositivos produzem e consomem Fluxos Contínuos de Dados (*data streams*) com diversos objetivos. Alguns cenários de IoT envolvem a mineração desses fluxos (*data stream mining*) em busca de padrões para tomada de decisão e, por vezes requerem também baixa

latência. Para casos de baixa latência ou alta vazão, conexões adequadas para processamento em nuvem nem sempre são possíveis ou desejáveis; para esses casos, a computação em névoa (*fog computing*) é uma solução.

O tema de *data stream mining* envolve a classificação de novos elementos, que podem tanto estar relacionados aos dados ou aos metadados das comunicações, com base em um modelo. as classes contidas em um *data stream* não são todas previamente conhecidas. A identificação e classificação de novas classes em *data streams* é denominada Detecção de Novidades (*Novelty Detection*, DNFD) em *data streams*.

inerentes a *data stream mining*, são considerados na construção de um que computa *data streams* a taxa de eventos gerados por cada produtor e o número de produtores nesse sistema, totalizando o volume de eventos . Volumes elevados dificilmente são computados em apenas um nó (e muito menos em um único núcleo processador) e por isso, esses sistemas são distribuídos.

Sistemas que utilizam DNFD para *data streams* gerados por dispositivos IoT devem utilizar algoritmos que considerem os desafios inerentes a fluxos de dados (*Concept Evolution* e *Concept Drift*) para adequada detecção de novidades e, para tanto, requerem processamento em arquiteturas que atendam os requisitos de volume de mensagens e latência de detecção. O algoritmo MINAS é adequado para esse caso, pois trata os desafios de *data stream mining*, porém não tem ainda implementação que atenda os requisitos de volume e latência, especialmente para aplicações IoT onde um ambiente de *fog computing* é atrativo.

Para preencher a lacuna de algoritmo de DNFD em ambiente *fog computing*, propõem-se então o sistema M-FOG, uma implementação do algoritmo MINAS sobre a plataforma *Apache Flink*, que considera distribuição em um ambiente de *fog computing*. O sistema M-FOG descrito neste documento foi refinado com os resultados dos experimentos descritos na Seção 5.1 e poderá ser revisado ao longo da pesquisa conforme os resultados de outros experimentos evidenciarem obstáculos ou oportunidades de melhoria.

4.2 Descrição da Arquitetura Proposta

Nesta Seção, apresenta-se o sistema M-FOG, objeto proposta deste trabalho. O sistema M-FOG é composto de três módulos principais e dois auxiliares. Os módulos principais implementam o algoritmo MINAS, sendo eles: módulo treinamento (*Training Module*), módulo classificador (*Classification Module*) e módulo detector de novidades (*Novelty Detection Module*). Dois módulos auxiliares são utilizados para avaliação do sistema M-FOG: módulo auxiliar *source* (fonte) e módulo auxiliar *sink* (sorvedouro, consumidor final). Os módulos e as interações entre eles são ilustradas na Figura 4.1.

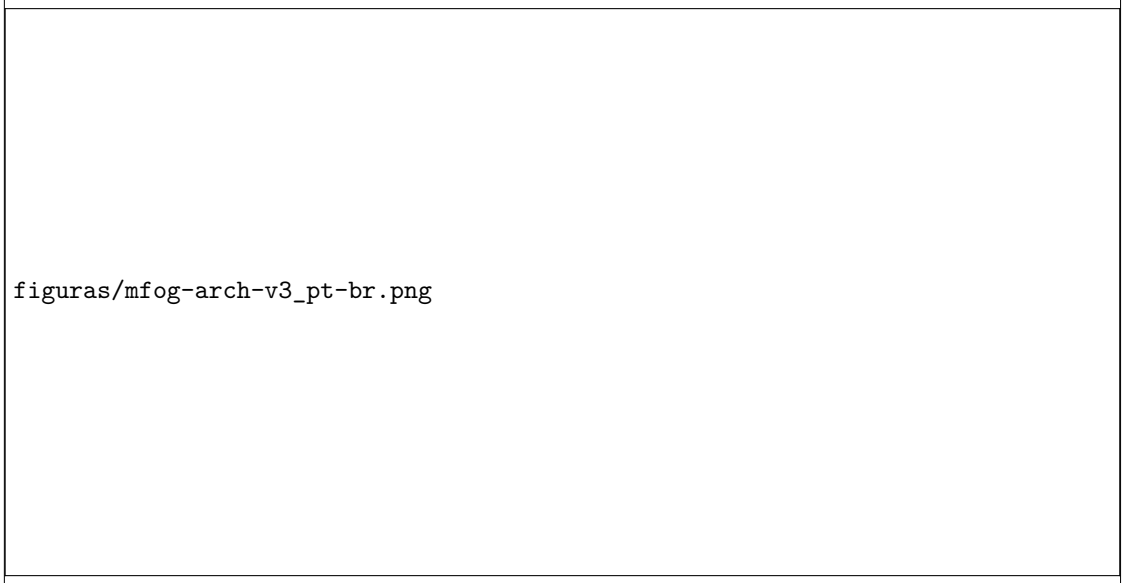

Figura 4.1: Arquitetura e fluxos de dados do sistema M-FOG.

A implementação do sistema M-FOG segue a arquitetura IDSA-IoT formalizada por [CSDB19], discutida na Seção 3.3. A arquitetura IDSA-IoT estabelece que um serviço de captura e tratamento de dados é instalado na borda de uma rede local com dispositivos IoT. Na presente implementação, esse serviço de captura e tratamento é representado pelo módulo auxiliar *source*.

O módulo auxiliar *source* é dependente da fonte de dados, executando a transformação dos formatos dos *data sets* para um fluxo de exemplos (representado por x na Figura 4.1) compatível com o restante da implementação. Além de fornecer exemplos tratados para o módulo classificador, o módulo auxiliar *source* também fornece exemplos com a classe original (representado por x, c na Figura 4.1) .

O módulo auxiliar *sink* é responsável por agregar todos resultados do sistema M-FOG e, juntamente com os valores do *data set* fornecidos pelo módulo auxiliar *source*, por computar as métricas de qualidade de classificação. Além disso, esse módulo também coleta e agrega métricas base para as avaliação de escalabilidade e métricas de uso de recursos computacionais.

Os dados resultantes do serviço de captura e tratamento (representado no sistema M-FOG pelo módulo auxiliar *source*) são ingeridos pela aplicação no módulo classificador. A ingestão é feita por meio de um operador fonte, fornecida pela plataforma *Apache Flink*. Na plataforma, com o modelo de classificação disponível, os exemplos são classificados seguindo o algoritmo MINAS original discutido na Seção 2.4.1. A etiqueta atribuída pela classificação, ou meta-etiqueta de desconhecido, juntamente com o exemplo original (representado por x, l na Figura 4.1) são enviados para o módulo auxiliar *sink*. Além disso, se o exemplo não for classificado, o exemplo e a meta-etiqueta de desconhecido (representado por x, u na Figura 4.1) são enviados para o módulo detector de novidades. Outra comunicação é o envio das modificações ao sumário estatístico do modelo de classificação (representado por *Summary* na Figura 4.1) do módulo classificador para o módulo detector de novidades.

O módulo detector de novidades é responsável por executar o processo de detecção de novidade, atualizando o modelo de classificação, e entregar o novo modelo às instâncias do módulo classificador, através do serviço de armazenamento de modelo (*Model Store* na Figura 4.1). O módulo detector de novidades também envia meta-informações sobre o processo de detecção de novidade (representado por *Log* na Figura 4.1) para o módulo auxiliar *sink*.

O sistema M-FOG utiliza em seus módulos a distribuição oferecida pela plataforma *Apache Flink* como paralelização, ou seja, utiliza uma instância de trabalho (*job*) por dispositivo de classificação, sendo que cada instância de trabalho aloca um gerenciador de tarefas por processador. Dessa forma, busca-se a escalabilidade no ambiente de *fog computing* para o módulo classificador. O módulo treinamento, por ser utilizado somente uma vez para gerar o modelo de classificação inicial, não tem impacto na escalabilidade geral do sistema. O módulo detector de novidades também é implementado na plataforma *Apache Flink* e, por ser hospedado em ambiente de *cloud computing*, herda as qualidades desse ambiente incluindo escalabilidade. O restante do sistema (módulo auxiliar *source*, módulo auxiliar *sink*, armazenamento de modelo) não é foco deste estudo e sua escalabilidade, desde que não afete a escalabilidade do módulo classificador e módulo detector de novidades.

4.3 Metodologia de Avaliação

A avaliação da proposta apresentada é feita por meio de métricas extraídas da literatura, divididas em duas partes: métricas de qualidade de classificação e métricas de escalabilidade. Métricas tradicionais de qualidade de classificação estabelecidas por trabalhos de aprendizado de máquina não são adequadas para avaliar detecção de novidades em *data streams* sem tratamento inicial. Felizmente, o tratamento necessário é estabelecido por [FGGC13] e expandido por [DSDD18, dS18, JFS+19, Cos19]. Além do tratamento estabelecido, as métricas tradicionais não são calculadas somente para o conjunto completo, e sim para cada exemplo classificado. Portanto, as métricas têm como índice o instante (n nas equações à seguir), informando a posição do exemplo em relação ao fluxo.

O tratamento estabelecido das métricas de qualidade para *data stream mining* define que as métricas sejam extraídas de uma matriz de erro de classificação multi-classe $\mathbf{E}_n$ (Equação 4.3), adaptada para detecção de novidade. A matriz de erro é preenchida com o número de eventos da classe c_i classificados com etiqueta l_j até o instante n. A Equação 4.1 representa o conjunto de classes presentes nos eventos do fluxo até o instante n e a Equação 4.2 representa o conjunto de etiquetas atribuídas pelo classificador a eventos até o mesmo instante.

$$\mathbf{C}_n = \{c_1, c_2, \cdots, c_M\} \tag{4.1}$$

$$\mathbf{L}_n = \{l_1, l_2, \cdots, l_J\} \tag{4.2}$$

$$\mathbf{E}_n = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,J} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,J} \\ \vdots & \vdots & \ddots & \vdots \\ e_{M,1} & e_{M,2} & \cdots & e_{M,J} \end{pmatrix} \tag{4.3}$$

As métricas de qualidade de classificação selecionadas para avaliar a implementação do sistema M-FOG serão taxa de desconhecidos ($UnkR$ na Equação 4.4) [FGGC13], acurácia média (acc na Equação 4.5) e Macro F-score ($Fscore$ na Equação 4.9, também referido na literatura por F1M) [SL09, dS18]. As métricas são extraídas para todos os exemplos classificados (instantes n) da respectiva matriz de erro $\mathbf{E}_n$.

$$UnkR_n = \frac{1}{M} \sum_{i=1}^{M} \frac{\#Unk_i}{\#ExC_i} \tag{4.4}$$

$$acc_n = \frac{1}{M} \sum_{i=1}^{M} \frac{tp_i + tn_i}{tp_i + fn_i + fp_i + tn_i} = \frac{1}{M} \sum_{i=1}^{M} \frac{\#Acc_i}{\#ExC_i} \tag{4.5}$$

$$Precision_n = \frac{1}{M} \sum_{i=1}^{M} \frac{tp_i}{tp_i + fp_i} \tag{4.6}$$

$$Recall_n = \frac{1}{M} \sum_{i=1}^{M} \frac{tp_i}{tp_i + fn_i} \tag{4.7}$$

$$Fscore\beta_n = (\beta^2 + 1) \cdot \frac{Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} \tag{4.8}$$

$$Fscore1_n = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{4.9}$$

A transformação do fluxo de saída em uma matriz de erro é realizada no módulo auxiliar *sink*, Esse módulo deve levar em consideração que pode haver reclassificação de um evento, previamente rotulado como desconhecido, em padrões oriundos de classe novidade ou extensão devido ao processo de detecção de novidades executado posteriormente ao surgimento do padrão em questão.

As métricas de escalabilidade selecionadas são: número de nós processadores, tipo de processadores, uso de memória, tempo de processamento, taxa de eventos processados e latência entre a produção e classificação de um evento.

Da implementação do sistema M-FOG é prevista a execução de experimentos com *data sets* diversos, em especial os *data sets* reais como *Kyoto 2006+*, que contenham evolução de conceitos. Os resultados desses experimentos irão conter as seguintes métricas:

a) Qualidade de classificação (taxa de desconhecidos, F1M);

b) Escalabilidade (número de processadores, volume processado, tempo decorrido);

c) Recursos computacionais utilizados (memória, tempo de processamento, operações de leitura e escrita).

Para a validação da corretude da implementação do sistema M-FOG com relação ao algoritmo MINAS original, as métricas de qualidade de classificação serão extraídas de ambas as Implementação e comparadas.

5 Implementação

5.1 Resultados preliminares

No desenvolvimento parcial desta pesquisa, algumas experimentações e algumas ferramentas de teste já foram desenvolvidas. Aspectos desses desenvolvimentos são descritos a seguir.

5.1.1 Implementação com *Python* e *Apache Kafka*

A primeira implementação e avaliação do sistema M-FOG realizada foi construída sobre a linguagem *Python* com o sistema de fila de mensagens *Apache Kafka* e a respectiva biblioteca de conexão. A escolha desse conjunto para a implementação ocorreu disponibilidade de bibliotecas de aprendizagem de máquina no ecossistema *Python* e, à simplicidade geral da linguagem. Na implementação desenvolvida, o sistema *Apache Kafka* recebe mensagens e as armazena em tópicos distribuídos em partições replicadas em nós de um *cluster*, gerenciados por um nó mestre e suportados pelo serviço de gerenciamento de configuração distribuída *Apache ZooKeeper*. A aplicação *Python* consome eventos através da interface *Consumer API*, que expõe a distribuição através da associação de um consumidor às partições mantidas pelo *Apache Kafka*.

Para essa implementação, havia a hipótese de que a distribuição de mensagens gerenciada pelo *Apache Kafka* se estenderia a processos consumidores, efetivamente distribuindo o volume de mensagens entre eles igualmente. No entanto, a hipótese foi refutada nos experimentos realizados. Os experimentos em questão foram compostos de 8 processos consumidores, um processo produtor, uma instância *Apache Kafka* com 8 partições em seu tópico principal e uma instância *Apache ZooKeeper* associada à instância *Apache Kafka*. A hipótese foi refutada quando observou-se que o número de mensagens consumidas por um dos 8 processos representava a maioria (mais de 80%) do volume introduzido no sistema, o restante sendo distribuído entre outros 3 processos e o restante dos processos não recebia nenhuma mensagem. Portanto, a iniciativa de implementar o algoritmo MINAS em *Python* com *Apache Kafka* e atingir os objetivos de distribuição falhou, o que levou à reconsideração das plataformas escolhidas.

5.1.2 Implementação com *Apache Flink*

A segunda alternativa explorada teve por inspiração o trabalho de [VSBN19] e, como outro grupo de pesquisa já estava explorando o algoritmo na plataforma *Apache Spark*, a segunda implementação foi baseada na plataforma *Apache Flink*.

A plataforma *Apache Flink* tem modelos de processamento tanto de fluxos como em lotes. O modelo em lotes é implementado como extensão do modelo de fluxos e, apesar de não ser foco desse trabalho, mostrou-se útil para a construção do módulo treinamento, já que o conjunto consumido por esse módulo é limitado.

Um desafio encontrado durante o desenvolvimento da implementação do sistema M-FOG foi a falta de bibliotecas na plataforma *Apache Flink* que disponibilizem versões adaptadas à plataforma de algoritmos base para o algoritmo MINAS. Em especial, a ausência dos algoritmos *K-means* e *CluStream* gerou carga imprevista sobre o processo de desenvolvimento resultando no atraso do processo de desenvolvimento.

Esta implementação segue a arquitetura descrita na Seção 4.2 e as avaliações e resultados esperados descritos neste Capítulo 4 referem-se à implementação do sistema M-FOG na plataforma *Apache Flink*.

5.2 Implementação com MPI

The original MINAS algorithm has a companion unpublished implementation (*Ref*) written in Java using MOA library base algorithms such as K-means and CluStream, but our implementation only used K-means. Another difference between *Ref* and sistema M-FOG is the calculus of the cluster radius from the distances of elements forming the cluster and the cluster's center. *Ref* uses the maximum distance while sistema M-FOG uses the standard deviation of all distances as described in [FPdLFCG15].

The stream formats for input and output are also of note. As input, the algorithm takes samples ($\vec{v}$), which are a sequence of numbers with dimension d. In addition to $\vec{v}$, for both training and evaluation, the class identifier is provided as a single character, along with a unique item identifier (*uid*), which can otherwise be determined from the sample index in the stream.

As its output, the algorithm returns the original sample $\vec{v}$ followed by the assigned label. Adjustments can easily be made to provide the output results as a tuple containing *uid* and the assigned label.

For evaluation purposes, an sistema M-FOG implementation was made using MPI (*Open MPI 4.0.4*). The program is organized in a single program multiple data (SPMD) programming model, so a single version of the sistema M-FOG program was initiated on all nodes, being that one of them would perform the root role, while the others ran as leaves, the program entry point is illustrated on Algorithm 3. On the root process, a sampler thread is responsible for distributing

```
   Parameters: mpiNodeRank as mpiRank
   Input: ModelSet, Sample Stream
 1 Function Mfog(ModelStream, InputStream, OutputStream):
 2 |   ModelSet = ∅;
 3 |   ModelSetLock = new Lock ();
 4 |   if mpiRank == 0 then root
 5 |   |   new Thread (Detector, [OutputStream, ModelSet, ModelSetLock]);
 6 |   |   Sampler (InputStream, ModelSet, ModelSetLock);
 7 |   else leaf
 8 |   |   new Thread (modelReceiver, [ModelSet, ModelSetLock]);
 9 |   |   Classifier (ModelSet, ModelSetLock);
```

Algorithm 3: MFOG: main MPI entry-point.

```
 1 Function Classifier(ModelSet, ModelSetLock):
 2 |   while True do
 3 |   |   sampe = receive (SampleType, root);
 4 |   |   if sample == EndOfStream then break;
 5 |   |   sample.label = unknown;
 6 |   |   with readLock (ModelSetLock)
 7 |   |   |   (distance, cluster) = nearestCluster (sample, ModelSet);
 8 |   |   if distance < cluster.radius then
 9 |   |   |   sample.label = cluster.label;
10 |   |   send (root, SampleType, sample);
11 Function modelReceiver(ModelSet, ModelSetLock):
12 |   while True do
13 |   |   cl = receive (ClusterType, root);
14 |   |   if cl == EndOfStream then break;
15 |   |   with writeLock(ModelSetLock)
16 |   |   |   ModelSet = ModelSet ∪ cl;
```

Algorithm 4: MFOG Leaf Tasks: Model Receiver and Classifier.

the sampled flow information ($\vec{v}$) to the classifier nodes, using a round-robin load balancing scheme. The other thread on the root process is responsible for receiving the classification results and for processing the unknown samples in the search for novelties. The root process functions are illustrated in Algorithm 5. Each leaf node runs a model adjustment thread and multiple (up to the number of cores) classifier threads. The leaf tasks are illustrated in Algorithm 4. The overall sequence of interactions is shown in Figure 5.1.

6 Experimentos e Resultados

6.1 Ambiente de Teste

Aiming to evaluate our proposal for the effects of distributed novelty detection in a IoT NIDS scenario, we implemented an experimental setup, composed of three Raspberry Pi 3 model B single board computers connected via Ethernet Switch. The idea was to create a simple cluster simulating an IoT network with constrained resources at the edge of the network. This cluster stored all source code, binaries (compiled and linked in place) and data sets. In our setup, the data set is stored in the root's node SD card and is read for each experiment. All experiments were executed in this cluster for isolation of otherwise unforeseen variations and for safe software comparison with constant hardware.

The data set used is the December 2015 segment of Kyoto 2006+ data set[1] (Traffic Data from Kyoto University's Honeypots) [STO+11a] containing 7 865 245 samples. From the original data set, we filtered only samples associated with normal traffic or known attack types identified by existing NIDS, and attack types with more than 10 000 samples for significance, as previously done by [CSDB19]. The remaining samples then were normalized so each feature value space (e.g., IP Address, Duration, Service) is translated to the Real interval $[0, 1]$.

The resulting derived data set is then stored in two sets, training set and test set, using the

[1] Available at `http://www.takakura.com/Kyoto_data/`

```
   Parameters: mpiClusterSize as mpiSize
 1 Function Sampler(InputStream, ModelSet, ModelSetLock):
 2 |   dest = 1;
 3 |   foreach sample from InputStream do
 4 |   |   if typeOf (sample) is Cluster then
 5 |   |   |   broadcast (ClusterType, sample, root);
 6 |   |   |   with writeLock (ModelSetLock)
 7 |   |   |   |   ModelSet = ModelSet ∪ sample;
 8 |   |   |   continue;
 9 |   |   send (dest, SampleType, sample);
10 |   |   dest = dest +1;
11 |   |   if dest > mpiSize then dest = 1;
   Parameters: cleaningWindow, noveltyDetectionTrigger
12 Function Detector(OutputStream, ModelSet, ModelSetLock):
13 |   lastCleanup ← 0;
14 |   while True do
15 |   |   sampe = receive (SampleType, any);
16 |   |   if sample == EndOfStream then break;
17 |   |   OutputStream.append(sample);
18 |   |   if sample.label == unknown then
19 |   |   |   UnkownSet = UnkownSet ∪ sample;
20 |   |   |   if | UnkownSet | ≥ noveltyDetectionTrigger then
21 |   |   |   |   novelties = NoveltyDetection (p, ModelSet, *UnkownSet);
22 |   |   |   |   with writeLock (ModelSetLock)
23 |   |   |   |   |   ModelSet = ModelSet ∪ novelties;
24 |   |   |   |   foreach cl in novelties do
25 |   |   |   |   |   broadcast (ClusterType, cl, root);
26 |   |   |   if sampe.uid > ( lastCleanup + cleaningWindow) then
27 |   |   |   |   UnkownSet ← removeOldSamples (UnkownSet, lastCleanup);
28 |   |   |   |   lastCleanup ← sampe.uid;
```

Algorithm 5: MFOG Root Tasks: Sampler and Detector.

holdout technique. However, for the training set we filter in only normal class resulting in 72 000 instances. For the test set we use 653457 instances with 206278 instances with "N" (normal) class and 447 179 instances with "A" (attack) class. Note that this choice results in possible overfitting for the normal class and, under-fitting for the attack class as the system first needs to detect a novel class and then add it to the model.

Para realização dos experimentos, diversas configurações de ambientes são propostas. Os ambientes selecionados são: local, . As configurações consistem na distribuição de módulos da implementação sistema M-FOG sendo executadas em combinações de ambientes nuvem e névoa com variada quantidade de nós.

O ambiente local é composto por um único nó computacional, consistindo de um computador pessoal equipado com um processador de 8 núcleos, $16GB$ de memória e armazenamento em estado sólido (SSD) usado para o desenvolvimento e referência em comparações. O ambiente nuvem é provido pela utilização da infraestrutura de nuvem da Universidade Federal de São Carlos (Cloud@UFSCar[2]). O ambiente de névoa (*fog computing*) é composto por computadores de única placa (*Single Board Computer*) equipados com processador de arquitetura ARM de 4 núcleos, $1GB$ de memória, armazenamento em cartão SD (*SD-card*) e conectados por rede sem fio.

A combinação de diferentes distribuições tem por objetivo e qualidade que podem afetar implantações em ambientes reais que não são geralmente destacados quando os experimentos são realizados em um único nó ou ambiente.

Faz parte também do ambiente de teste os conjuntos de dados (*data sets*) *KDD99* e *Kyoto 2006+* que foram selecionados por motivos distintos.

O *data set Kyoto 2006+* é o foco deste trabalho, pois contém dados ainda representativos (até 2015) e as característica desejáveis de um conjunto de dados (realismo, validade, etiquetas previamente definidas, alta variabilidade, reprodutibilidade e disponibilidade pública) são atendidas [STO20, STO$^+$11b].

[2]Disponível em `http://portalcloud.ufscar.br/servicos`

Figura 5.1: sistema M-FOG life line overview.

O *data set KDD99* é amplamente utilizado em trabalhos de detecção de anomalia. Porém, como não possui mais a característica de realismo, uma vez que foi construído em 1998, neste trabalho o *data set KDD99* é utilizado somente para que o leitor possa comparar com outros trabalhos [TBLG09, Pro18].

Os dois *data sets* mencionados e outros abordados em discussão e avaliados como relevantes são

Tabela 6.1: Sumário dos conjuntos de dados

Nome	Origem	Descrição	Acesso Público
KDD99 [TBLG09, Pro18]	Captura de Fluxos de rede com ataques simulados	41 atributos (sumário de fluxo), 23 classes, 4 898 431 instâncias, 709 MB	`https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`
Kyoto 2006+ [STO+11b, Pro18]	Captura de Fluxos de rede com HoneyPot	23 atributos (sumário de fluxo), 3 classes, 7 865 245 instâncias e 1.3 GB (dez-2015)	`https://www.takakura.com/Kyoto_data/new_data201704/`
CICIDS2017 [SLG18]	Captura de Fluxos de rede com ataques simulados com perfil de trafego de 25 usuários normais e de 6 perfis de ataques durante 5 dias (1º dia sem ataque)	80 atributos (sumário de fluxo extraído de CICFlowMeter), 15 classes, 2 830 751 instâncias e 1.2GB em arquivos *pcap* e *csv*	`https://www.unb.ca/cic/datasets/ids-2017.html`
Radial Basis Function (RBF) da biblioteca *Massive Online Analysis* (MOA) *4CRE-V2*	Sintético gerado por função RBF da biblioteca MOA com características de mudança e evolução de conceito	Atributos ($\mathbb{R}$), exemplos, classes, evoluções e mudanças configuráveis	`https://sites.google.com/site/nonstationaryarchive/home`

6.2 Métricas e Visualizações

We have used two types of evaluation measurements for each experiment: a measure of the full experiment execution time and, a set of qualitative measurements extracted by a Python

script.

Our evaluation script was build following reference techniques like multi-class confusion matrix with label-class association [FPdLFCG15] to extract classification quality measurements. This script takes two inputs, the test data set and the captured output stream, and outputs the confusion matrix, label-class association, final quality summary with: *Hits* (true positive), *Misses* (Err), *Unknowns* (UnkR); and stream visualization chart with per example instance summary with novelty label markers.

In the confusion matrix $M = m_{ij} \in \mathbb{N}^{c \times l}$, computed by our evaluation script, each row denotes the actual class c and each column denotes the predicted label l present in the captured output stream. Thus, each cell $M_{c,l}$ contains the count of examples from the test data set of class c found in the output stream with the label l assigned by the under evaluation experiment.

For the data set under use, original classes are $c \in \{N, A\}$, and for the labels we have the training class *"N"*, *unknown* label *"-"* and the novelties $i \in \mathbb{N}$ so $l \in \{N, -\} \cup \mathbb{N}$.

Added to the original confusion matrix M are the rows *Assigned* and *Hits*. *Assigned* row represents which original class c (or if *unknown*, *"-"*) the label l is assigned to, this is computed by using the original class if $c = l$ or by associated novelty label to original class as described in [dFGGC15] section 4.1 (class from where the most samples came from). *Hits* row shows the true positive count for each label l with assigned class c, being the same value as cell $M_{c,l}$. The *Hits* row is also used to compute the overall true positive in the summary table and stream visualization chart. One complete matrix is shown in Tab. 6.2.

Tabela 6.2: Reference implementation

Labels	-	N	1	2	3	4	5	6	7	8	9	10	11	12
Classes														
A	3774	438750	123	145	368	8	52	165	1	1046	161	2489	71	26
N	8206	193030	0	79	44	0	0	0	229	181	154	4066	289	0
Assigned	-	N	A	A	A	A	A	A	N	A	A	N	N	A
Hits	0	193030	123	145	368	8	52	165	229	1046	161	4066	289	26

Tabela 6.3: Serial implementation

Labels	-	N	0	1	2	4	5	6	7	8	10
Classes											
A	16086	429765	94	995	104	0	23	3	29	46	34
N	12481	193642	3	94	0	47	0	0	0	11	0
Assigned	-	N	A	A	A	N	A	A	A	A	A
Hits	0	193642	94	995	104	47	23	3	29	46	34

Tabela 6.4: Parallel single-node

Lab.	-	N	0	1	2	3	4
Cla.							
A	12282	433797	147	952	0	0	1
N	3088	203019	40	99	27	5	0
Ass.	-	N	A	A	N	N	A
Hits	0	203019	147	952	27	5	1

For the measurements summary table, six measurements from two sources are displayed. Three measures *Hits*, *Unknowns* and *Misses* represented as ratio of the captured output stream, extracted from the evaluation python program, computed as follows: *Hits* (true positive rate) is the sum of the *Hits* row in the extended confusion matrix; *Unknowns* is the count of examples in the captured output stream marked with the *unknown* label (*"-"*); *Misses* is the count of all examples in the captured output stream marked with a label distinct from the *Assigned* original class and are not marked as unknown.

Furthermore in the measurement summary table, *Time*, *System* and *Elapsed* represented in seconds, are extracted from *GNU Time 1.9*. *Time* is the amount of CPU seconds expended in user-mode (indicates time used doing CPU intensive computing, e.g., math); *System* is the amount of CPU seconds expended in kernel-mode (for our case, it indicates time doing input or output); *Elapsed* is the real-world (wall clock) elapsed time and indicates how long the program

Tabela 6.5: Parallel multi-node

Lab.	-	N	0	1	2	3	4
Cla.							
A	12378	433631	117	886	0	162	5
N	3121	202916	40	96	105	0	0
Ass.	-	N	A	A	N	A	A
Hits	0	202916	117	886	105	162	5

took to complete. The lower the times, the better. Our four main experiments are shown in Tab. **??**.

Lastly, the stream visualization chart shows the summary quality measurement (*Hits*, *Unknowns*, *Misses*) computed for each example in the captured output stream. This summary is computed for each example, but it uses the *Assigned* row computed previously to evaluate *Hits*; the other measurements are derived as described before. The Horizontal axis (x, domain) plots the index of the example and the vertical axis (y, image) shows the measurement computed until that example index on the captured output stream.

Adding to the stream visualization chart, novelty label markers are represented as vertical lines indicating *when* in the captured output stream a new label first appeared. Some of the novelty label markers include the label itself ($l \in \mathbb{N}$) for reference (showing every label would turn this feature unreadable due to overlapping). Figure 6.4 shows complete stream visualization charts.

Reference Implementation

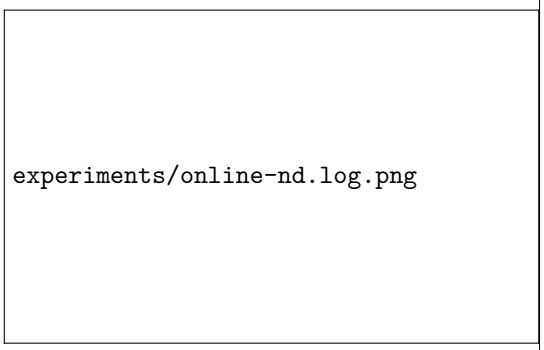

Figura 6.1: Serial Implementation

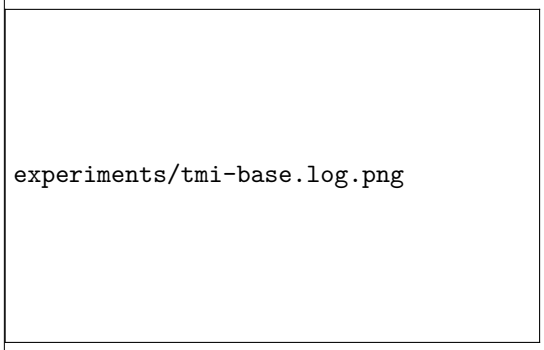

Figura 6.2: Parallel single-node

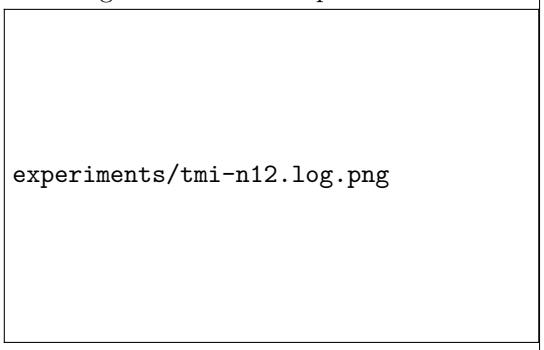

Figura 6.3: Parallel multi-node

Figura 6.4: Stream hits and novelties visualization