Uma Implementação distribuída em Névoa do algoritmo de Detecção de Novidade em Fluxos de Dados MINAS

Luís Henrique Puhl de Souza

23 de abril de 2021

ficha-catalografica.pdf

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato , realizada em 23 de abril de 2021:

Orientador **Professor**
Convidado 1 **Professor**
Convidado 2

Agradeço à constante compania, incentivo e ensinamentos dos colegas e agradeço especialmente à CNPq pelo suporte financiero (contrato 167345/2018-4).

Detecção de Novidades. Detecção de Intrusão. Fluxos de Dados. Computação Distribuída. Computação em Névoa. Internet das Coisas

Em um cenário de crescente número de dispositivos na Internet das Coisas (IoT), gerando proporcional crescimento no volume dos fluxos de dados gerados, são necessários métodos robustos para a mineração de fluxos contínuos de dados. Uma das áreas afetadas pelo crescimento vertiginoso do número de dispositivos e os fluxos associados a eles é a área de segurança da informação, onde são necessárias ferramentas de detecção de intrusão em redes que operem em ambientes de computação em névoa, devido aos custos de comunicação associados a operar estas ferramentas somente em ambiente de nuvem. As ferramentas de detecção de intrusão utilizam extensivamente algoritmos de detecção de novidade em fluxos de dados para identificar padrões no tráfego da rede. Porém, os algoritmos que tratam adequadamente dos desafios de detecção de novidade em fluxos de dados, como mudança e evolução de conceito e atualização contínua do modelo de classificação sem interferência de especialistas, ainda são pouco utilizados. O algoritmo de detecção de novidade em fluxo de dados MINAS tem recebido atenção de pesquisas recentes por tratar desses desafios de detecção de novidade em fluxos de dados. No entanto, apesar de sua divisão em três partes semi-independentes, este algoritmo ainda não foi adaptado para processar grandes volumes de fluxos reais em ambiente de computação em névoa. O presente trabalho aborda essa lacuna, propondo um sistema que implementa o algoritmo MINAS de maneiraiot distribuída num contexto de detecção de intrusão e computação em névoa. Experimentos mostram que o algoritmo MINAS pode ser paralelizado e distribuído utilizando plataformas de processamento de fluxos como *Apache Flink*.

Resumo

Novelty Detection. Intrusion Detection. Data Streams. Distributed Computing. Fog Computing. IoT devices

The ongoing implementation of the Internet of Things (IoT) is sharply increasing the number and variety of small devices on edge networks. Likewise, the attack opportunities for hostile agents also increases, requiring more effort from network administrators and strategies to detect and react to those threats. For a network security system to operate in the context of edge and IoT, it has to comply with processing, storage, and energy requirements alongside traditional requirements for stream and network analysis like accuracy and scalability. Using a previously defined architecture (IDSA-IoT), we address the construction and evaluation of a support mechanism for distributed Network Intrusion Detection Systems (NIDS) based on the MINAS Data Stream Novelty Detection algorithm. We discuss the algorithm steps, how it can be deployed in a distributed environment, the impacts on the accuracy and evaluate performance and scalability using a cluster of constrained devices commonly found in IoT scenarios. The obtained results show a negligible accuracy loss in the distributed version but also a small reduction in the execution time using low profile devices. Although not efficient, the parallel version showed to be viable as the proposed granularity provides equivalent accuracy and viable response times.

Capítulo 1

Introdução

The Internet of Things () 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 [**?**, **?**, **?**, **?**]. Security and privacy is a major concern in the , especially regarding devices having access to user personal data like location, health and many other sensitive data [**?**]. 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 [**?**]. As an additional concern, 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 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 [**?**]. 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 [**?**, **?**]. A recent survey [**?**] shows that ML based methods are a promising alternative which can provide potential security tools for the network making them more reliable and accessible than before.

Despite the promising use of ML to secure systems, studies found in the literature [**?**, **?**, **?**] 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 [**?**, **?**]. 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 [?, ?, ?] use of Data Stream Novelty Detection () in network data streams, this paper shows the effects of adapting these mechanisms to edge services for use in environments. Our proposal, called , adapted the architecture [?] using the algorithm [?, ?], 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 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 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 [?]. 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*,). 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 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 . Para tratar esse problema, a arquitetura IDSA-IoT, recentemente proposta por **?**, 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, **?** avaliou os algoritmos ECSMiner [**?**], AnyNovel [**?**] e MINAS [**?**], 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 [**?**].

Contudo, o algoritmo MINAS ainda não foi implementado e avaliado com paralelismo, multi-processamento 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 , 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 como método de detecção de intrusão, outras propostas tratam do caso de fluxos com grandes volumes e velocidades, como é o caso de **?**, 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 **?** e **?** 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 arquitetura 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 **??**, 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 **?**, esta pesquisa escolheu investigar o algoritmo MINAS [**?**] 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 **??**, 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 [**?**], 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 **??** aborda conceitos teóricos e técnicos que embasam esse trabalho; Capítulo **??** enumera e discute trabalhos relacionados e estabelece o estado da arte do tema detecção de novidade em fluxos de dados e seu processamento; Capítulo **??** descreve a proposta de implementação, discute as escolhas de plataformas e resultados esperados. Também são discutidos no Capítulo **??** os desafios e resultados preliminares encontrados durante o desenvolvimento do trabalho. Capítulo **??** Capítulo **??** adiciona considerações gerais e apresenta o plano de trabalho e cronograma até a defesa do mestrado.

Capítulo 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 , entre outras aplicações. A computação em nuvem () é aplicada a vários problemas e neste trabalho seu papel em sistemas 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 (), 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 [**?**].

As principais características do ambiente , segundo **?** são: Serviço sob Demanda, Amplo acesso à rede, Agrupamento de recursos, Elasticidade e Serviço mensurado. Segundo, **?**, 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 [**?**]. Na borda, é possível fazer

armazenamento, processamento e descarregamento de dados, assim como distribuir as requisições e entregar os serviços das nuvens aos usuários. **?** 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 [**?**].

? e **?** 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). **?** 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 [**?**]. 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 [**?**, **?**, **?**]. Esse tipo de computação evita a sobrecarga dos dispositivos de borda. **?** e **?** consideram computação em névoa como complementar da computação em borda, podendo a computação em névoa aproveitar os recursos da nuvem e da borda. **?** 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 **?** e **?**, 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 [**?**]. 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.

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 $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n, \ldots$ [**?**].

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 [**?**, **?**].

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 [**?**].

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 [**?**].

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 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 **?**, 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* [**?**] e *Kappa* [**?**], 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 [**?**].

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* [**?**].

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* [**?**].

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 [**?**].

Uma das extensões de *Apache Spark* é *Spark Streaming*, que é um sistema de processamento de fluxo de dados escalável e tolerante a falhas [**?**, **?**]. *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 [**?**]. 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 [**?**, **?**].

Apache Storm é um sistema de computação tolerante a falhas em tempo real que de fluxo de dados [**?**, **?**]. 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) [**?**]. Essa plataforma segue um paradigma que abrange o processamento de fluxos de dados contínuos e o processamento em lote [**?**, **?**]. 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 standalone*. Já o acesso programático a essa plataforma pode ser feito através das linguagens Java, Scala ou Python.

2.3.1 *OpenMPI*

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.

Capítulo 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 **??**.

Tabela 3.1: Sumário dos trabalhos relacionados

Trabalho	Plataforma	Técnica	Conjunto de dados	Métricas
Ferramenta BigFlow [**?**]	*Python*, *flowtbag*, *Apache Kafka* e	*Hoeffding Tree*, *OzaBoosting*, *Leveraging Bag* e comitê	*MAWILab*	Acurácia (geral e por classe), Taxa de bytes
Ferramenta CATRACA [**?**]	*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 [**?**]	Java, *Apache Kafka* e *Python*	ECSMiner, AnyNovel e MINAS	*Kyoto 2006+*	Fnew, Mnew e erro

3.1 Ferramenta BigFlow

Proposta por **?**, 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.

? destaca que 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 MAWIFlow é proposto por **?**. Este é derivado do *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 [**?**, **?**]. Desse original, o 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 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* [**?**, **?**]. 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 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 **?** 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 ().

Para o desenvolvimento da ferramenta CATRACA, **?** 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 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 [**?**]. 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 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

[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/`.

[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.

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 (). 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 **??** (, , 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 **?**, tem por objetivo monitorar uma rede local com dispositivos 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 (). O objetivo dessa distribuição é a redução de latência, que torna inviável a hospedagem de um sistema detector de intrusão somente em ambiente , 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 [**?**], AnyNovel [**?**] e MINAS [**?**]. A avaliação foi feita com o *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 tem as características desejáveis de um conjunto para detectção de novidades como: realismo, validade, etiquetas previamente definidas, alta variabilidade, reprodutibilidade e disponibilidade pública. O *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.

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

Capítulo 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 at the edge, as a means of allowing small IoT devices to filter and detect undesirable network behavior. Our approach is based on the architecture [**?**] and techniques provide by the algorithm [**?**]. Named , 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 , a distributed data stream novelty detection system based on the algorithm for securing networks. implements a distributed version of according to the architecture proposed in a previous work [**?**], 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).

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 [**?**]: 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 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 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 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 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 architecture has two main modules, Classification and Novelty Detection, which implement the main tasks. The Classification Module performs the same task of the 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 Online phase, receiving all the samples labeled with *unknown*, storing them in an internal *unknown-buffer*, and, when this buffer is full, performing the Novelty Detection task (clustering followed by validation).

4.1 Polices

The design of our distributed architecture includes partitioning the functionalities of 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 , the *unknown* sample buffer is effectively classified using the new set of clusters. In Algorithm **??**, at the line **??**, 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 () 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 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. Porém, como *data streams* variam temporalmente e são ilimitados, 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 (,) 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 para *data streams* gerados por dispositivos devem utilizar algoritmos que considerem os desafios inerentes a fluxos de dados (e) 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 onde um ambiente de *fog computing* é atrativo.

Para preencher a lacuna de algoritmo de em ambiente *fog computing*, propõem-se então o , uma implementação do algoritmo MINAS sobre a plataforma , que considera distribuição em um ambiente de *fog computing*. O descrito neste documento foi refinado com os resultados dos experimentos descritos na Seção **??** 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 , objeto proposta deste trabalho. O 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 : módulo auxiliar *source* (fonte) e módulo auxiliar *sink* (sorvedouro, consumidor final). Os módulos e as interações entre eles são ilustradas na Figura **??**.

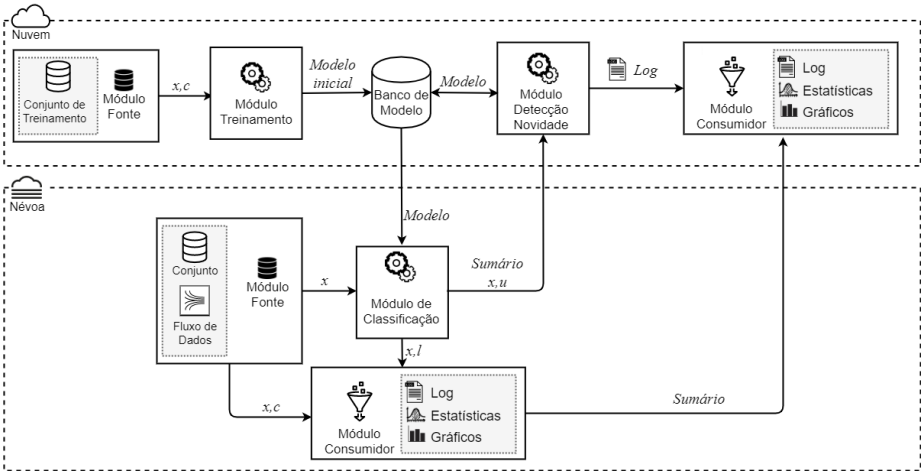


Figura 4.1: Arquitetura e fluxos de dados do .

A implementação do segue a arquitetura IDSA-IoT formalizada por **?**, discutida na Seção **??**. A arquitetura IDSA-IoT estabelece que um serviço de captura e tratamento de dados é instalado na borda de uma rede local com dispositivos . 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 para um fluxo de exemplos (representado por x na Figura **??**) 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 **??**) .

O módulo auxiliar *sink* é responsável por agregar todos resultados do e, juntamente com os valores do 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 pelo módulo auxiliar *source*) são ingeridos pela aplicação no módulo classificador. A ingestão é feita por meio de um operador fonte, fornecida pela plataforma . Na plataforma, com o modelo de classificação disponível, os exemplos são classificados seguindo o algoritmo MINAS original discutido na Seção **??**. A etiqueta atribuída pela classificação, ou meta-etiqueta de desconhecido, juntamente com o exemplo original (representado por x, l na Figura **??**) 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 **??**) 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 **??**) 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 **??**). O módulo detector de novidades também envia meta-informações sobre o processo de detecção de novidade (representado por *Log* na Figura **??**) para o módulo auxiliar *sink*.

O utiliza em seus módulos a distribuição oferecida pela plataforma 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 e, por ser hospedado em ambiente de , herda as qualidades desse ambiente incluindo escalabilidade. destaque sentença 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 **?** e expandido por **?**, **?**, **?**, **?**. 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 **??**), 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 **??** representa o conjunto de classes presentes nos eventos do fluxo até o instante n e a Equação **??** 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 serão taxa de desconhecidos ($UnkR$ na Equação **??**) [**?**], acurácia média (acc na Equação **??**) e Macro F-score ($Fscore$ na Equação **??**, também referido na literatura por F1M) [**?**, **?**]. 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 é prevista a execução de experimentos com diversos, em especial os 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 com relação ao algoritmo MINAS original, as métricas de qualidade de classificação serão extraídas de ambas as Implementação e comparadas.

4.4 Article

4.4.1 Introduction

The Internet of Things () 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 [?, ?, ?, ?]. Security and privacy is a major concern in the , especially regarding devices having access to user personal data like location, health and many other sensitive data [?]. 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 [?]. As an additional concern, 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 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 [?]. 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 [?, ?]. A recent survey [?] shows that ML based methods are a promising alternative which can provide potential security tools for the network making them more reliable and accessible than before.

Despite the promising use of ML to secure systems, studies found in the literature [?, ?, ?] 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 [?, ?]. 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 [?, ?, ?] use of Data Stream Novelty Detection () in network data streams, this paper shows the effects of adapting these mechanisms to edge services for use in environments. Our proposal, called , adapted the architecture [?] using the algorithm [?, ?], 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 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.

This paper is organized as follows: Section **??** reviews the chosen algorithm . A distributed extension of , including its implementation and evaluation are presented in Section **??** and in Section **??** we show how we evaluated and the discuss results we found. Finally, Section **??** summarizes the main findings and presents possible future work.

4.4.2 Proposal

In this work, we investigate an appropriate architecture for performing at the edge, as a means of allowing small IoT devices to filter and detect undesirable network behavior. Our approach is based on the architecture [**?**] and techniques provide by the algorithm [**?**]. Named , 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 , a distributed data stream novelty detection system based on the algorithm for securing networks. implements a distributed version of according to the architecture proposed in a previous work [**?**], 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).

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 [**?**]: 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 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 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 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 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 architecture has two main modules, Classification and Novelty Detection, which implement the main tasks. The Classification Module performs

the same task of the 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 Online phase, receiving all the samples labeled with *unknown*, storing them in an internal *unknown-buffer*, and, when this buffer is full, performing the Novelty Detection task (clustering followed by validation).

Polices

The design of our distributed architecture includes partitioning the functionalities of 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 , the *unknown* sample buffer is effectively classified using the new set of clusters. In Algorithm **??**, at the line **??**, 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).

Implementation

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

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.

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 1: 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 2: MFOG Leaf Tasks: Model Receiver and Classifier.

For evaluation purposes, an 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 program was initiated on all

```
   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 3: MFOG Root Tasks: Sampler and Detector.

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 **??**. On the root process, a sampler thread is responsible for distributing 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 **??**. 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 **??**. The overall sequence of interactions is shown in Figure **??**.

4.5 Experiments and Results

Aiming to evaluate our proposal for the effects of distributed novelty detection in a 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 network with constrained resources at the edge of the network. This cluster stored all source code, binaries (compiled

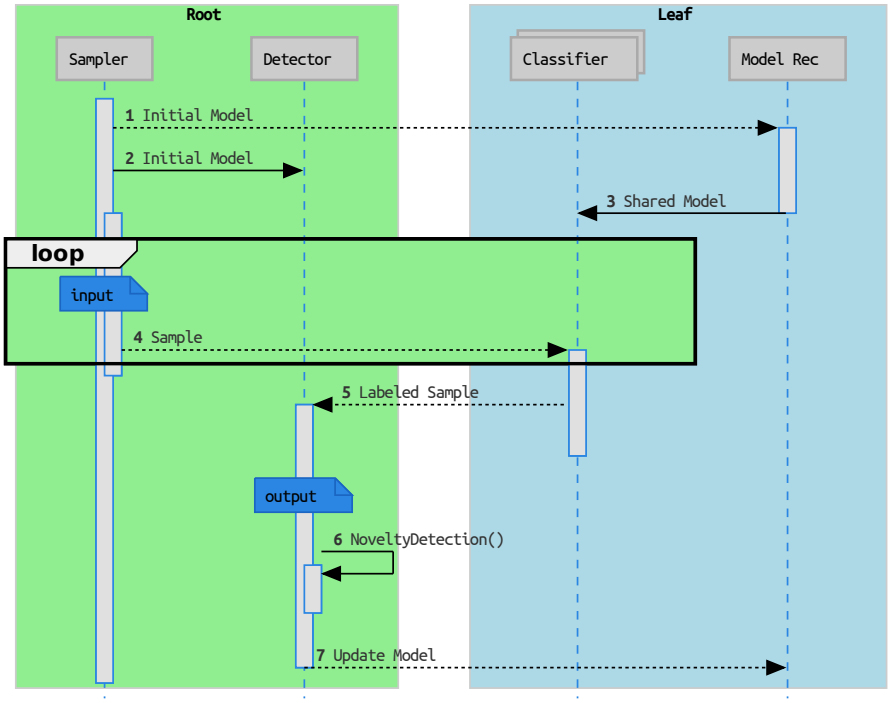


Figura 4.2: life line overview.

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) [**?**] 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 , and attack types with more than 10 000 samples for significance, as previously done by [**?**]. 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 holdout technique. However, for the training set we filter in only normal class resulting in 72 000 instances. For the test set we use 653 457 instances with 206 278 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.

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

4.5.1 Measurements and Visualizations

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 [**?**] 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 [**?**] 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. **??**.

Tabela 4.1: 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 4.2: 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

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,

Tabela 4.3: 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

Tabela 4.4: 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

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.

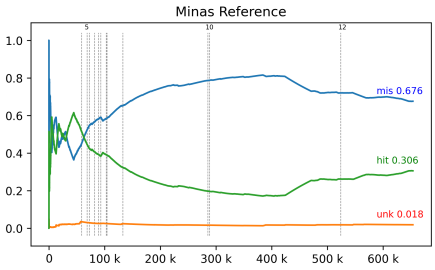
Furthermore in the measurement summary table, *Time*, *System* and *Elapsed* represented in seconds, are extracted from *GNU Time 1.9*. *Time* is the amount of CPU seconds expended in user-mode (indicates time used doing CPU intensive computing, e.g., math); *System* is the amount of CPU seconds expended 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 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 **??** shows complete stream visualization charts.

Discussion

Four main experiments are presented for discussion: (a) reference implementation of Minas () [**?**]; (b) new implementation in serial mode; (c) new implementation in single-node, multi-task mode and (d) new implementation in multi-node, multi-task mode. Each experiment uses the adequate binary executable, initial model (or training set for the reference implementation) and test set to com-



Reference Implementation

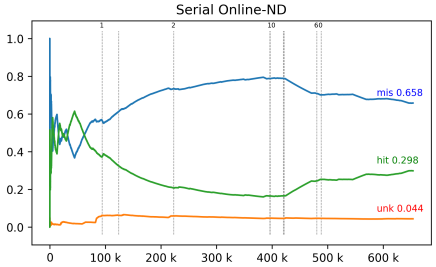


Figura 4.3: Serial Implementation

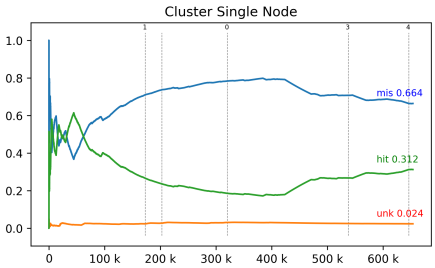


Figura 4.4: Parallel single-node

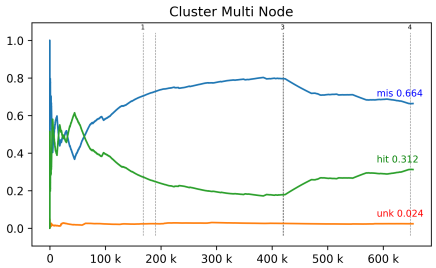


Figura 4.5: Parallel multi-node

Figura 4.6: Stream hits and novelties visualization

pute a resulting output stream which is stored for qualitative evaluation. The summary of all four experiments is shown in Table **??**.

Tabela 4.5: Collected Measures Summary.

	(a)	Offline	Serial (b)	Single Node (c)	Multi Node (d)
Hits	199708 0.305618		195017 0.298438	204151 0.312416	204191 0.312478
Misses	441769 0.676049		429873 0.657843	433936 0.664061	433767 0.663802
Unknowns	11980 0.018333		28567 0.043717	15370 0.023521	15499 0.023718
Time	2761.83	194.12	80.79000	522.1000	207.1400
System	7.15	0.075	11.51000	47.7700	157.6100
Elapsed	2772.07	194.27	93.03000	145.0400	95.3800

The comparison of the first two experiments (a and b) provides a validation for our implementation, while the latter three (b, c and d) serve as showcase for the effects of distribution.

As stated, to validate our implementation we have compared it to (the original companion implementation), so we extracted the same measurements using same process for both a and b, which can be viewed in Tables **??**, **??** and for ease of comparison in Table **??** the summary can be compared side by side.

In general, the observed classification quality measurements are very similar, and only diverge slightly where *a* has more *Hits* and *Misses* whereas *b* shifted those to *Unknowns*. This phenomenon was watched very closely during development and we found that it was due to small changes to parameters, internals like K-means ordering, cluster edge inclusion and cluster radius formula as stated in Subsection **??**.

As for the time measurements in Table **??** our implementation used less time to analyze the test data set. This is mostly due to the stop condition on the internal K-means algorithm; while uses a fixed iteration limit of 100, our implementations adds the "no improvement" check and stops earlier in most cases, which in turn reduces the time taken on the *NoveltyDetection* function. There are also small optimizations on the *nearestCluster* function (minimal distance from sample to cluster center in the set) affecting the *classifier* task and *NoveltyDetection* function. One can also note that time in *a* includes the Offline phase while our implementation runs it once and reuses the initial model for *b*, *c* and *d*. In the table the offline time this is shown as a separate column.

As for the effects of running the classification processes on the small devices as MPI nodes with our implementation, we observe an increase of time when we go from 2 to 4 instances in a single node (*b* and *c* respectively), hinting that our choice of load distribution is not as effective as we expected. Further experiments were conducted with the number of instances varying from 1 (serial) to 12 (3 nodes with 4 CPUs each), but that caused no impact on the true positive rate (*Hits*) and elapsed time. More detailed time measurements can be seen in Figure **??**, where we observe near constant time for *elapsed* (near 100*s*), the *system* increases gradually while *user* decreases at the same rate. We interpret this behavior as a display of potential for gains using a better load balancing than our choice of round-robin such as micro-batching for better compute-to-communication ratio (CCR). In general, Figure **??** shows no speedup but also no penalty for scaling to more than 4 instances.

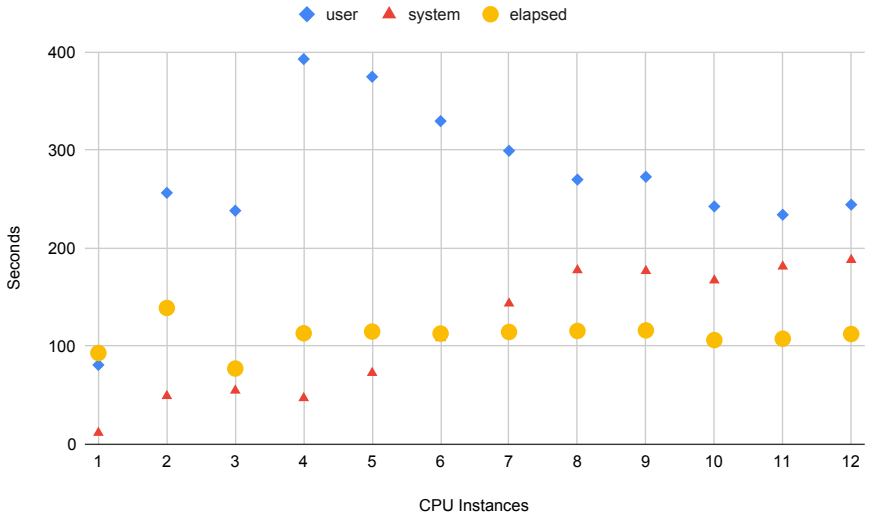


Figura 4.7: Time measurements per added instance

Nevertheless, we can also show the effects of delay in the Classify, Novelty Detection, Model Update and Classify feedback loop. Comparing b and c we observe a reduction in Novelty labels on the Confusion Matrix (tabs. **??** and **??**) from 10 to 4. The same effect is observed on the stream visualization (figs. **??** and **??**) where our serial implementation has fewer novelty markers, and they appear later, but the measures keep the same "shape". Comparing c and d the difference is even smaller, (figs. **??** and **??**) as they both suffer the expected delay in the feedback loop.

4.5.2 Conclusion

Data Stream Novelty Detection () can be a useful mechanism for Network Intrusion Detection () in IoT environments. It can also serve other related applications of using continuous network or system behavior monitoring and analysis. Regarding the tremendous amount of data that must be processed in the flow analysis for , it is relevant that this processing takes place at the edge of the network. However, one relevant shortcoming of the IoT, in this case, is the reduced processing capacity of such edge devices.

In this sense, we have put together and evaluated a distributed architecture for performing in network flows at the edge. Our proposal, is a distributed implementation based on the algorithm .

The main goal of this work is to observe the effects of our approach to a previously serial only algorithm, especially in regards to time and quality metrics.

While there is some impact on the predictive metrics, this is not reflected on overall classification quality metrics indicating that distribution of shows a negligible loss of accuracy. In regards to time and scale, our distributed executions was faster than the previous sequential implementation of , but efficient data distribution was not achieved as the observed time with each added node remained constant.

Overall, and the idea of using distributed flow classification and novelty detection while minimizing memory usage to fit in smaller devices at the edge of the network is a viable and promising solution. Further work include the investigation of other algorithms, other clustering algorithms in and analysis of varying load balancing strategies.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Programa Institucional de Internacionalização – CAPES-PrInt UFSCar (Contract 88887.373234/2019-00). Authors also thank Stic AMSUD (project 20-STIC-09), FAPESP (contract numbers 2018/22979-2, and 2015/24461-2) and CNPq (Contract 167345/2018-4) for their support.

Capítulo 5

Considerações Finais

Este trabalho reúne conceitos de aprendizado de máquina com ênfase em detecção de novidades em fluxos contínuos de dados e conceitos de processamento distribuído de fluxos contínuos, com o objetivo de unir a lacuna no estado da arte desses conceitos à luz de uma implementação e avaliação no cenário de detecção de intrusão em redes de dispositivos da Internet das Coisas () em ambiente de computação em névoa (*fog computing*).

O objeto central desse trabalho () trata da implementação do algoritmo MINAS na plataforma de processamento de fluxos , em três módulos que podem ser distribuídos em um ambiente de *fog computing*. Sua distribuição permite selecionar o nó que tem os recursos computacionais mais adequados para cada tarefa. A avaliação do será feita por meio de métricas de qualidade de classificação e métricas de escalabilidade.

Dando continuidade a este trabalho, segue-se com o desenvolvimento da implementação objeto () bem como a contínua avaliação comparativa dos resultados produzidos pelo com seu algoritmo base, MINAS. Também será dada continuidade nos experimentos com os conjuntos de dados () diversos e configurações variadas de distribuição de processamento em *fog computing* extraindo desses experimentos as métricas previamente discutidas.

Dessa forma, o pode contribuir com adição de uma ferramenta para os interessados em sistemas de detecção de intrusão de redes de dispositivos ou outros sistemas que tratam de fluxos contínuos que tradicionalmente sofrem com os ônus de latência e largura de banda na comunicação entre borda e nuvem. Além disso, o objetiva contribuir com a adição de uma implementação distribuída de um algoritmo cujo modelo é estado da arte em detecção de novidades em fluxos contínuos de dados.

Data Stream Novelty Detection () can be a useful mechanism for Network Intrusion Detection () in IoT environments. It can also serve other related applications of using continuous network or system behavior monitoring and analysis. Regarding the tremendous amount of data that must be processed in the flow analysis for , it is relevant that this processing takes place at the edge of the network. However, one relevant shortcoming of the IoT, in this case, is the reduced processing capacity of such edge devices.

In this sense, we have put together and evaluated a distributed architecture for performing in network flows at the edge. Our proposal, is a distributed implementation based on the algorithm .

The main goal of this work is to observe the effects of our approach to a previously serial only algorithm, especially in regards to time and quality metrics.

While there is some impact on the predictive metrics, this is not reflected on overall classification quality metrics indicating that distribution of shows a negligible loss of accuracy. In regards to time and scale, our distributed executions was faster than the previous sequential implementation of , but efficient data distribution was not achieved as the observed time with each added node remained constant.

Overall, and the idea of using distributed flow classification and novelty detection while minimizing memory usage to fit in smaller devices at the edge of the network is a viable and promising solution. Further work include the investigation of other algorithms, other clustering algorithms in and analysis of varying load balancing strategies.