Universidade Federal de São Carlos – UFSCar
Centro de Ciências Exatas e de Tecnologia – CCET
Departamento de Computação – DC
Programa de Pós-Graduação em Ciência da Computação – PPGCC

Luís Henrique Puhl de Souza

Uma Implementação Distribuída em Névoa do Algoritmo de Detecção de Novidade em Fluxos de Dados MINAS

São Carlos
2021

Luís Henrique Puhl de Souza

Uma Implementação Distribuída em Névoa do Algoritmo de Detecção de Novidade em Fluxos de Dados MINAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Sistemas de Computação

Orientador: Doutor Hermes Senger

São Carlos
2021

Puhl, Luis

Uma Implementação distribuída em Névoa do algoritmo de Detecção de Novidade em Fluxos de Dados MINAS / Luis Puhl -- 2021.
65f.

Dissertação (Mestrado) - Universidade Federal de São Carlos, campus São Carlos, São Carlos
Orientador (a): Hermes Senger
Banca Examinadora: Helio Guardia
Bibliografia

1. Detecção de Intrusão. 2. Internet das Coisas. I. Puhl, Luis. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Ronildo Santos Prado - CRB/8 7325

Universidade Federal de São Carlos – UFSCar
Centro de Ciências Exatas e de Tecnologia – CCET
Departamento de Computação – DC
Programa de Pós-Graduação em Ciência da Computação – PPGCC

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Luís Henrique Puhl de Souza, realizada em 5 de Julho de 2021:

Doutor Hermes Senger
Presidente e Orientador

Doutor Kelton Augusto Pontara da Costa
Convidado

Doutor Paulo Sérgio Lopes de Souza
Convidado

São Carlos
2021

Agradecimentos

Agradeço os ensinamentos dos colegas, especialmente ao Professor Hélio, Professor Hermes e Mestre Guilherme pela experência compartilhada.

Agradeço à companheira Carolina pelo constante alegria e apoio a todos os momentos e ao incentivo e ajuda nas horas difíceis.

Agradeço aos familiares e amigos pelo carinho.

Agradeço também à CNPq pelo suporte financiero (contrato 167345/2018-4).

Resumo

Em um cenário de crescente número de dispositivos na Internet das Coisas (IoT), gerando proporcional crescimento no volume dos fluxos de dados gerados, são constantes e evolutivas as ameaças ativas e passivas aos recursos computacionais e aos conteúdos transmitidos. Métodos para mineração de dados de forma robusta e contínua podem ser um aliado à segurança nesses casos. Particularmente em ambientes distribuídos e nos quais busca-se manter os fluxos de informação próximos a onde eles são gerados, como nas bordas das redes IoT, e na computação em névoa de maneira geral, a detecção de ameaças é essencial e não trivial. Além disso, a evolução constante dos tipos de dispositivos e de tráfegos nessas redes favorece que as ferramentas de detecção de ameaças sejam beneficiadas por algoritmos de detecção de novidades em fluxo de dados. MINAS é um exemplo de algoritmo de detecção de novidades em fluxos de dados com potencial para aplicação na computação em névoa. 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 maneira distribuída num contexto de detecção de intrusão e computação em névoa e também avalia esta implementação. Os resultados obtidos mostram a viabilidade do modelo de detecção de novidades distribuído em ambiente de computação em névoa. Mesmo em um cenário distribuído, o sistema apresentou eficácia equivalente a do algoritmo original, ainda que executando em dispositivos de capacidade computacional limitada, como é previsto em cenários de IoT.

Palavras-chave: Detecção de Novidades. Detecção de Intrusão. Fluxos de Dados. Computação Distribuída. Computação em Névoa. Internet das Coisas.

Abstract

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 fog 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 equivalent metrics in the distributed version but also a 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 same response times.

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

Lista de ilustrações

Lista de tabelas

Lista de siglas

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

A **IoT!** (**IoT!**) conecta globalmente variados dispositivos, incluindo dispositivos móveis, *wearables*, eletrônicos domésticos, automóveis e sensores industriais. Estes dispositivos podem, através da Internet, ser acessados, conectar-se a outros dispositivos, servidores ou aplicações, tudo com mínima intervenção ou supervisão humana (**????????**). Outra característica de dispositivos **IoT!** são os recursos computacionais dimensionados para propósitos específicos, que limitam a capacidade de computar outras funções muito além da função original do dispositivo.

Segurança e privacidade são uma grande preocupação em **IoT!**, especialmente em relação aos dados pessoais como localização e saúde aos quais dispositivos podem ter acesso (**??**). Além dos dados de sensores e atuadores que esses dispositivos gerenciam, se esses dispositivos forem subvertidos podem gerar tráfego maligno, como o produzido pela *mirai botnet* 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, bem como a menor frequência de atualizações deste software. Além disso, estes dispositivos têm longa vida e, após implantação, convivem com ampla diversidade de outros dispositivos, o que torna complexa a manutenção da rede que os hospeda, aumentando sua superfície de ataque.

No contexto de segurança de redes **IoT!**, ferramentas que facilitem a detecção e resposta a ataques são necessárias. Como a maioria dos dispositivos **IoT!** tem recursos limitados (como energia, processamento, memória e comunicação), técnicas de segurança tradicionais baseadas em algoritmos configuráveis não são usuais, restando as técnicas de observação de rede (**??**). Ferramentas como **NIDS!** (**NIDS!**) observam o comportamento da rede e de seus dispositivos e detectam possíveis ataques.

Para implementação de **NIDS!**, técnicas de **ML!** (**ML!**) têm sido empregadas na

detecção de ataques a partir de características de ataques conhecidos ou na descoberta de novos ataques o mais cedo possível (**????**). Apesar do uso promissor de **ML!** para segurança para sistemas **IoT!**, muitos estudos na literatura (**??????**) são limitados a métodos tradicionais de **ML!**. Estes métodos comumente utilizam modelos estáticos para descrever e prever o comportamento da rede que não mantêm a confiabilidade frente à evolução de ataques (**????**).

Além das complicações de confiabilidade, a grande quantidade de dispositivos em redes distantes gerando dados em volumes e velocidades elevadas as técnicas tradicionais que tratam grandes lotes em *datacenters* não são aplicáveis. Para esses fluxos contínuos de dados (*Data Stream*), técnicas de mineração de fluxos de dados (*Data Stream Mining*) são promissoras. 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.

Dentre as técnicas de mineração de fluxo de dados, classificadores podem ser utilizados para detectar padrões conhecidos e, em conjunto com algoritmos de **ND!** (**ND!**) (*Novelty Detection in Data Streams*), detectar novos padrões. Essa capacidade de detectar novos padrões é relevante para **NIDS!**, onde novidades na rede podem representar novas funcionalidades ou ataques por agentes maliciosos, sem assinaturas existentes em bancos de dados de ataques conhecidos. Outras características que fazem **ND!** atraente para **NIDS!** são a produção de respostas imediatas e a detecção de novidades e mudança de conceitos já conhecidos.

Análises como mineração de fluxos de dados e **ND!** têm sido implementadas sobre o paradigma de computação na nuvem (*Cloud Computing*) e, recentemente, também sobre paradigmas como computação em névoa (*fog computing*). Para névoa, além dos recursos em nuvem, são explorados os recursos distribuídos pela rede desde nós remotos até a nuvem. Processos que dependem desses recursos são distribuídos de acordo com características como sensibilidade à latência, privacidade, consumo computacional ou energético.

De maneira geral, a aplicação de **ND!** para detecção de ameaças em fluxos de dados originários de redes **IoT!** dentro de **NIDS!** tem sido um ponto de interesse (**??????**). Este trabalho explora as características de implementação destas técnicas em conjunto, concentrando-se em serviços localizados na borda da rede, de maneira distribuída, para uso em ambientes **IoT!**.

Este trabalho apresenta a construção e avaliação do sistema M-FOG[1], uma implementação paralela e distribuída em névoa de dispositivos **IoT!** do algoritmo MINAS. Esta implementação foi construída com o padrão **MPI!** (**MPI!**) buscando escalabilidade na tarefa de processamento de fluxo de dados e economia dos recursos limitados comumente encontrados em sistemas **IoT!**, seguindo a arquitetura IDSA-IoT (**??**).

[1] Disponível em <https://github.com/luis-puhl/minas-flink>.

A avaliação do sistema M-FOG é constituída de métricas de qualidade de classificação e métricas de escalabilidade, extraídas com auxilio do conjunto de dados *Kyoto 2015* relevante para **NIDS!**. As métricas de qualidade de classificação obtidas dos resultados mostraram valores equivalentes à implementação de referência do algoritmo MINAS e as métricas de escalabilidade mostraram melhora em relação à implementação de referência porém com eficiência de paralelismo abaixo do esperado. Este trabalho contribui com uma análise do algoritmo MINAS com a ótica de distribuição em névoa, mostrando benefícios e desafios deste tipo de aplicação, iluminando os detalhes do problema abordado e apontando algumas soluções para trabalhos futuros.

1.1 Motivação

Um problema recente que une, em um único contexto, os métodos de computação em névoa, processamento de fluxo de dados e detecção de novidades nesses fluxos é a detecção de intrusão em redes de dispositivos **IoT!**. Para tratar esse problema, a arquitetura IDSA-IoT, recentemente proposta por **??**), aplica ao problema algoritmos relevantes do tema 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.

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 em ambientes distribuídos, como em cenários **IoT!** e névoa. O tratamento de distribuição em ambiente névoa é essencial para aplicação deste algoritmo ao problema de detecção de intrusão em redes **IoT!**, pois esta aplicação requer tempo de resposta mínimo e pequena comunicação entre nós distantes, como aquelas comunicações entre borda e a nuvem. Ainda observando o algoritmo MINAS, destaca-se a possível divisão em três partes semi-independentes, sendo elas treinamento, classificação e detecção de novidades. A classificação é o elemento central cujos resultados são utilizados para a identificação de intrusões, enquanto a detecção de novidades fornece atualização automática do modelo de classificação.

Ainda no contexto de **ND!** 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. 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!**, porém com perspectivas diferentes. O primeiro investiga **IoT!** e processamento em névoa e baseia-se em um algoritmo genérico de detecção de novidade. O segundo trabalho trata de *backbones* e processamento em nuvem 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 de névoa com um algoritmo estado da arte de detecção de novidades, porém sem paralelismo. Do outro lado da lacuna, tem-se um sistema escalável de alto desempenho porém limitado ao ambiente nuvem e com um algoritmo que não foi projetado para os desafios de detecção de novidades.

A proposta deste trabalho, aqui chamada sistema M-FOG, adapta a arquitetura IDSA-IoT (**??**) empregando o algoritmo de **ND!** MINAS (**??**), tornando-o capaz de ser executado em um sistema distribuído composto de pequenos computadores com recursos limitados, alocados na borda da rede próximos dos dispositivos **IoT!**. Utilizando a nova implementação do algoritmo MINAS, avalia-se experimentalmente como a distribuição afeta a capacidade do sistema de detectar mudanças (novidades) nos padrões de tráfego e o impacto na eficiência computacional. Por fim, algumas estratégias e políticas para configuração do sistema de detecção de novidades em fluxo de dados são discutidas.

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 no contexto de computação distribuída em névoa. Neste sentido, um algoritmo 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.

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, seguindo a arquitetura IDSA-IoT (**??**).

Além disso, propõem-se também 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; arquitetar e implementar um mecanismo de execução e avaliação de qualidade e desempenho para os ambientes escolhidos; avaliar e comparar a qualidade de detecção de intrusão da nova implementação; avaliar a qualidade de detecção de intrusão em ambiente distribuído conforme a arquitetura IDSA-IoT 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 foi o estabelecimento do estado da arte desses assuntos, de forma que alguns desses trabalhos servissem para comparações e relacionamentos. Além disso, desses trabalhos buscou-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 foi focada em plataformas, ferramentas e técnicas para realizar a implementação proposta. Portanto, foram selecionadas plataformas de processamento distribuído de fluxos contínuos de dados e técnicas de aprendizado de máquina associadas a elas. Dessa revisão também foram identificadas técnicas ou ferramentas necessárias para extração das métricas de avaliação, bem como conjunto de dados públicos relevantes para **NIDS!**.

Uma vez definidos o estado da arte, as ferramentas técnicas e os conjunto de dados, o passo seguinte foi a experimentação. Nesse passo, foi desenvolvida uma aplicação na plataforma escolhida que, com base no algoritmo MINAS (**??**), classifica e detecta novidades em fluxos contínuos de dados. Também nesse passo, a implementação foi validada comparando os resultados de classificação obtidos com os resultados de classificação do algoritmo original MINAS. Posteriormente, foram realizados experimentos com a implementação e variações em cenários de distribuição em névoa, coletando as métricas de classificação e escalabilidade.

Ao final, a aplicação, resultados, comparações e discussões foram organizados para publicação em 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 resultados preliminares durante a escolha da plataforma e apresenta os desafios encontrados durante o desenvolvimento do trabalho; Capítulo **??** aborda os ambientes de teste, descreve experimentos realizados e discute os resultados obtidos. Capítulo **??** adiciona considerações gerais sobre o trabalho, seus resultados e comentários para trabalhos futuros.

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 **IoT!**: computação em nuvem, computação de borda, e computação em névoa. A computação em nuvem (*Cloud Computing*) é aplicada a vários problemas, e para sistemas **IoT!** fornece vastos recursos como computação e armazenamento para onde geralmente os dispositivos enviam todos dados relevantes ao sistema. O segundo e terceiro ambientes são computação de borda (*edge computing*) e a computação em névoa (*fog computing*) que utilizam 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, 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 de nuvem, 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. 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 como 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 de computação de borda foi proposto para trazer o processamento e armazenamento para os dispositivos de borda tentando solucionar esses problemas. Entretanto, 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 capaci-

dades 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 computação de borda tem camadas menores, pois atua só nos dispositivos de borda, computação em névoa 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 para computação em névoa 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:** computação em névoa 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 ao longo de diferentes domínios;
- **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 Arquiteturas e Plataformas de Processamento de Fluxos de Dados

Tradicionalmente, aplicações foram construídas com um sistema gerenciador de banco de dados 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 fluxo de dados leva à necessidade de distribuir o processamento, acrescentando poder computacional a cada nó adicionado. Entretanto, 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 de fluxos de dados, 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 o *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 linhagem de RDD (*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 facilita o processamento de fluxo de dados (**????**). Ao invés de executar trabalhos (*jobs*) como algumas ferramentas citadas anteriormente, *Apache Storm* executa topologias. Os *jobs* eventualmente finalizam, e as topologias executam continuamente até serem finalizadas manualmente. 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 stand-alone*. Já o acesso programático a essa plataforma pode ser feito através das linguagens Java, Scala ou Python.

«««< HEAD

2.3 MPI - Interface de Troca de Mensagens

=======

2.4 Interface de Troca de Mensagens

MPI

»»»»> origin/overleaf-2021-06-15-2250 Em um sistema distribuído multiprocessado, a memória é distribuída entre múltiplos nós e processos, tendo cada processo acesso direto somente a sua memória local. Para esse tipo de sistema, o paradigma de programação paralela **SPMD!** (**SPMD!**) onde múltiplas instâncias de um único programa tratam parcelas de dados, pode ser implementado utilizando o conceito de memória distribuída compartilhada através de troca de mensagens. Nesse modelo, cada processo tem sua memória local e se comunica com outros processos através da troca de mensagens (**??**). Observando boas práticas e melhores funcionalidades ao longo dos anos, a comunidade e a indústria padronizaram esse modelo de troca de mensagens no padrão **MPI!** (**MPI!**).

O **MPI!**[1] é um padrão que estabelece um protocolo de comunicação e define sintaxe e semântica para bibliotecas de troca de mensagens em ambientes de memória distribuída compartilhada. Esse modelo pode ser implementado desde computadores multiprocessados de memória compartilhada até supercomputadores com centenas de nós e milhares de processadores. O **MPI!**, por meio de alguma implementação como OpenMPI e MPICH, permite a construção de um sistema distribuído com um executável único (monólito) e sua execução em um ambiente gerenciado.

O **MPI!** tem duas primitivas básicas para comunicação dos processos que são o enviar (*send*) e o receber (*receive*) (**??**). A primitiva de envio é composta pelos argumentos: conteúdo da mensagem, tamanho da mensagem, processo destino, e *tag* para diferenciar mensagens (ex: ordem e conteúdo). Associado à cada primitiva enviar, a primitiva receber possui os argumentos: conteúdo da mensagem, tamanho máximo da mensagem, remetente da mensagem, tamanho real da mensagem, e *tag*. Além de operações de envio e recebimento, pode-se fazer envio de mensagens de uma fonte para múltiplos destinos, e um destino receber de múltiplas fontes numa única operação.

Um dos detalhes importantes quando se utiliza o **MPI!** com o TCP/IP, é que as garantias de envio são fornecidas pelo protocolo TCP, ou seja, feito um envio com a operação *send* o processo que enviou é desbloqueado logo após que a mensagem é escrita no buffer de saída do TCP do sistema operacional. Portanto, existe tempo de voo (*time of flight*) entre o envio e recebimento.

O **MPI!** possui um ambiente de execução e gerenciamento chamado *mpirun* que recebe como entrada a configuração do cluster e o programa a ser executado, e se encarrega de iniciar os processos em todos os nós, geralmente por *ssh*, repassando os parâmetros que recebeu a cada processo e o identificador do processo para que ele saiba qual dado ler e qual função executar sob o paradigma **SPMD!**.

Este trabalho utiliza o OpenMPI[2], que é uma implementação livre de **MPI!**, para a comunicação entre os processos de cada nó do cluster. Cada nó do cluster possui um conjunto de processos, e cada processo possui seu espaço de endereçamento e é *multithread*,

[1] Disponível em <https://www.mpi-forum.org/>.
[2] Disponível em <https://www.open-mpi.org/>.

onde as *threads* se comunicam através da memória compartilhada. O OpenMPI permite a comunicação entre os processos independente se estão ou não no mesmo nó.

O amplo uso de **MPI!** para programação paralela e distribuída está principalmente pautado no desempenho. Processadores modernos possuem memórias hierárquicas que beneficiam programas que fazem o melhor uso de cache. O modelo de passagem de mensagens permite o particionamento dos dados em fatias menores otimizando o uso de cache nesses processadores. Portanto, para aplicações limitados por memória pode haver acelerações super-lineares (*superlinear speedup*) (**??**).

Apesar do **MPI!** não ter sido idealizado para o processamento de fluxo de dados, ele permite o controle sobre a distribuição de tarefas de maneira semelhante ao Apache Flink, porém dispensando o uso de um gerenciador. Essa dispensa de gerenciador em experimentos preliminares mostrou crucial para implementação do sistema M-FOG em dispositivos IoT, pois não excede o limite de memória do nó.

Em resumo, o **MPI!** é utilizado em um programa monólito escrito em C, executado por múltiplos processos em múltiplos nós, cada processo recebe um conjunto de dados diferente e, de acordo com o tipo de dado recebido, o processo executa funções diferentes. Em adição, os processos podem trocar informações entre si com mensagens, efetivamente compartilhando segmentos discretos de memória. Uma das estratégias é a leitura e distribuição de dados por um processo enquanto os outros processam, dividindo o trabalho, sendo possível desta maneira tratar de fluxos contínuos de dados de maneira escalável.

2.5 Fluxo de Dados e Mineração de 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 (**??**). A mineração de *Big Data* trata de enormes conjuntos de dados que podem não ser processados em tempo viável devido a limitações como memória ou armazenamento principal. Uma abordagem para minerar esse tipo dado é a mineração de fluxo de dados. Os fluxos de dados são definidos na Definição **??**.

Definição 1. *Um Fluxo de Dados S é uma sequência massiva, potencialmente ilimitada de exemplos $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n, \ldots$, recebida em instantes $T_1, T_2, \ldots, T_n, \ldots$, onde cada exemplo $\mathbf{X}$ tem dimensão d (**??**).*

O volume dos dados em *Big Data* e a velocidade com qual eles são produzidos afeta como os dados de um problema são modelados e manipulados. Técnicas e algoritmos de mineração de fluxos de dados atendem esses desafios utilizando restrições como apenas uma leitura do conjunto de dados e baixo complexidade na construção de seus algoritmos (**????**). O processamento requerido para a mineração de um enorme fluxos de dados pode não ser atendido por recursos computacionais de um único computador (nó), de forma que pode ser necessária a distribuição dos dados e do processamento em múltiplos nós computacionais em um sistema distribuído (**??**).

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 **??**.

2.5.1 Detecção de Novidade em Fluxo de Dados

No âmbito de classificação de dados, parte da área de aprendizado de máquina, os métodos de **ND!** (**ND!**) lidam com o reconhecimento e a classificação de exemplos que diferem de exemplos anteriores (**??????**). Esses métodos tratam da classificação em fluxos de dados que evoluem com o tempo.

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

Os métodos de **ND!** são aplicados a diversos problemas como detecção de intrusos (**????????**), detecção de falhas (**??**), diagnósticos médicos (**??**), detecção de regiões de interesse em imagens (**??**), detecção de fraudes (**????**), filtros de spam (**??**) e detecção de variações comportamentais em um jogador (**??**).

Alguns métodos de **ND!** tratam de novidades como uma classificação de uma ou duas classes (binariamente) onde um conceito representa a classe normal e as anomalias são

representadas pela falta de conceito no modelo ou como um segundo conceito no modelo. Além da abordagem de classificação binária, podem existir múltiplos conceitos em um mesmo conjunto de dados, para isso é necessário abordar **ND!** como classificação multi-classe. Alguns métodos que abordam **ND!** como classificação multi-classe não atendem completamente características de conjuntos com evolução temporal, como *Concept Evolution* e *Concept Drift*, deixando de detectar múltiplos padrões que surgem simultaneamente num intervalo de avaliação (**????**).

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

Dentre os métodos de **ND!** que baseiam-se em aprendizado *Offline-Online*, muitos são baseados em algoritmos de agrupamento não supervisionados, tanto para construção do modelo inicial como na extração de novos conceitos dos exemplos não explicados pelo modelo marcados como desconhecidos (**??????**).

2.5.2 O algoritmo MINAS

Um algoritmo de **ND!** que tem recebido atenção nos últimos anos é o algoritmo MINAS, originalmente proposto por **??**), refinado por **??**) e recentemente aprimorado por **??**), com o uso de conceitos *Fuzzy*, e expandido por **??**), para tratar problemas multi-rótulo além dos problemas multi-classe já tratados na versão original. Esse algoritmo segue a abordagem de duas fases no modelo *Offline-Online* e emprega algoritmos de agrupamento (*clustering*) não supervisionados como *K-means* e *CluStream*.

O algoritmo MINAS em sua fase *Offline* consome um conjunto de treinamento contendo exemplos rotulados. Esse conjunto de treinamento é dividido em grupos usando como chave o rótulo, e para cada grupo de exemplos o método de agrupamento é executado. O método de agrupamento objetiva resumir um conjunto maior de exemplos em um conjunto menor de *micro-clusters* (Definição **??**). A cada *micro-cluster* é adicionado o rótulo do grupo original e todos *micro-clusters* são arranjados em um único conjunto formando o modelo de decisão.

Definição 2. *Para um grupo de exemplos* $\mathbf{X} \in \mathcal{M}$*, um* micro-cluster $\mathcal{M}$ *é uma tupla de quatro componentes* $(n, \mathbf{LS}, \mathbf{SS}, t)$ *derivados dos exemplos representados por este* micro-cluster *onde (****????****):*

- ❑ n *é a contagem de exemplos,*
- ❑ **LS** *é o vetor soma linear dos exemplos sendo* $\mathbf{LS}_d = \sum_{i=0}^{n} \mathbf{X}_{i,d}$,
- ❑ **SS** *é o vetor da soma dos quadrados dos exemplos sendo* $\mathbf{SS}_d = \sum_{i=0}^{n} (\mathbf{X}_{i,d})^2$,
- ❑ t *é o instante de chegada do último exemplo adicionado ao* micro-cluster.

Deste sumário extrai-se o centro e o raio que são utilizados na operação de classificação da fase *Online* (**??**). O vetor centro é definido pelo algoritmo de agrupamento e corresponde ao ponto médio dos exemplos do *micro-cluster*, visto na Equação **??**. O raio pode ser definido de duas maneiras e ambas dependem no conjunto das distâncias $D_{\mathcal{M}}$ entre centro e cada elemento do *micro-cluster*. A primeira definição de raio é o desvio padrão das distâncias em $D_{\mathcal{M}}$ multiplicado por um fator parametrizado (**??**), visto na Equação **??**, a segunda definição é o valor máximo das distâncias em $D_{\mathcal{M}}$ (**??**), visto na Equação **??**.

Definição 3. *O vetor centro é o ponto médio dos exemplos do* micro-cluster*:*

$$centro = 1/n \cdot \mathbf{LS} \tag{1}$$

O conjunto das distâncias $D_{\mathcal{M}}$ *entre centro e cada elemento do* micro-cluster*:*

$$D_{\mathcal{M}} = \{d_i : d_i = dist(\mathbf{X}_i, centro)\} \qquad \mathbf{X}_i \in \mathcal{M}$$

O escalar raio é o desvio padrão ou o máximo do conjunto $D_{\mathcal{M}}$*:*

$$\mu = 1/n \cdot \sum_{i=1}^{n} d_i \qquad d_i \in D_{\mathcal{M}}$$

$$\sigma^2 = 1/n \cdot \sum_{i=1}^{n} (d_i - \mu)^2 \qquad d_i \in D_{\mathcal{M}}$$

$$\sigma = \sqrt{\sigma^2}$$

$$raio_1 = f_{raio} \cdot \sigma \tag{2}$$

$$raio_2 = max\{d_i \in D_{\mathcal{M}}\} \tag{3}$$

A distinção entre as duas definições de raio é importante, pois ela impacta imensamente nos resultados. Para cada *data set* ou problema abordado uma configuração de parâmetros afinada é essencial, pois as características da distribuição dos valores e sua associação às classes são geralmente únicas para cada problema. Em testes preliminares, a implementação do algoritmo MINAS com raio definido pela distância máxima (Eq. **??**) mostrou facilidade na configuração quando comparado com a mesma implementação utilizando a definição que utiliza desvio padrão (Eq. **??**), pois pequenas variações no parâmetro f_{raio} resultavam em alta taxa de desconhecidos ou alta taxa de erro.

Entrada: Modelo, fluxoEntrada
Saída : fluxoSaída
Parâmetros: janelaLimpeza, gatilhoDetecçãoNovidade

```
1  Função MinasOnline(Modelo, fluxoEntrada):
2  | Desconhecidos ← ∅, ModeloAntigo ← ∅ ;
3  | últimaLimpeza ← 0 , proximaNovidade ← 0;
4  | para cada exemplo_i ∈ fluxoEntrada faça
5  | | maisPróximo ← clusterMaisPróximo (exemplo, Modelo);
6  | | se maisPróximo.distância < maisPróximo.cluster.raio então
7  | | | exemplo.rótulo ← maisPróximo.cluster.rótulo;
8  | | | maisPróximo.cluster.últimoUso ← i ;
9  | | senão
10 | | | exemplo.rótulo ← "desconhecido";
11 | | | Desconhecidos ← Desconhecidos ∪ exemplo;
12 | | | se | Desconhecidos | ≥ gatilhoDetecçãoNovidade então
13 | | | | novidades ← DetecçãoNovidade (Modelo ∪ ModeloAntigo, *Desconhecidos);
14 | | | | Modelo ← Modelo ∪ novidades;
15 | | | se i > ( últimaLimpeza + janelaLimpeza) então
16 | | | | Modelo ← moveModeloAntigo (Modelo, *ModeloAntigo, últimaLimpeza);
17 | | | | Desconhecidos ← removeExemplosAntigos (Desconhecidos, últimaLimpeza);
18 | | | | últimaLimpeza ← i;
19 | | fluxoSaída.adicione(exemplo);
```

Algoritmo 1: Fase *online* interpretada do algoritmo MINAS (**??**).

Na fase *Online*, listada no Algoritmo **??**, o algoritmo MINAS executa três operações: classificação de novos exemplos, detecção de padrões-novidade e atualização do modelo de decisão (**??**).

A primeira operação é a classificação, onde exemplos do fluxo de dados são consumidos e avaliados pelo modelo de decisão. O modelo de decisão avalia cada exemplo calculando a distância euclidiana entre o exemplo e todos *micro-clusters* do modelo, selecionando o *micro-cluster* de menor distância. Se a distância entre o exemplo e o centro do *micro-cluster* for menor que o raio do *micro-cluster*, o exemplo é classificado com o rótulo do *micro-cluster* e o sumário estatístico do *micro-cluster* é atualizado. Caso a distância (mínima no modelo) seja maior que o raio, o exemplo é marcado como desconhecido e armazenado no conjunto de desconhecidos (**??**).

A segunda operação da fase *Online* é a detecção de padrões novidade listada no Algoritmo **??** que é executada quando o tamanho do conjunto de desconhecidos é maior que um parâmetro predefinido. Esta operação utiliza o agrupamento (*clustering* descrito na fase *Offline*) e valida os *micro-clusters* gerados verificando sua representatividade e coesão. A representatividade é avaliada com o número de exemplos do conjunto de desconhecidos que é englobado pelo novo *micro-cluster*, onde o valor mínimo de exemplos é parametrizado.

Um padrão é considerado coeso se o valor *silhueta*, Equação **??**, é maior que 0. O valor

silhueta por sua vez é calculado com os valores: b a distância do novo *micro-cluster* até o mais próximo contido no modelo atual; a o desvio padrão das distâncias entre o centro do novo *micro-cluster* e cada exemplo representado no novo *micro-cluster*.

$$\begin{aligned} a = \sigma &= SD\{dist(\mathbf{X}_i, centro) : \mathbf{X}_i \in \mathcal{M}\} \\ b &= min\{dist(centro_{novo}, centro(\mathcal{M}_i)) : \mathcal{M}_i \in Modelo\} \\ silhueta &= \frac{b-a}{max(b,a)} \end{aligned} \tag{4}$$

Parâmetros: minExemplos, fatorNovidade
Função `DetecçãoNovidade(`*Modelo, Desconhecidos*`):`
  novoModelo ← ∅;
  para cada *novoCluster* ∈ `agrupamento` *(Desconhecidos)* **faça**
    se (| *novoCluster.exemplos* | ≥ minExemplos) ∧ (*novoCluster.silhueta* > 0) **então**
      maisPróximo ← `clusterMaisPróximo` (novoCluster, Modelo);
      se *maisPróximo.distância* < *(maisPróximo.cluster.raio* × fatorNovidade*)* **então**
        novoCluster.rótulo ← maisPróximo.cluster.rótulo;
        novoCluster.tipo ← "extensão";
      senão
        novoCluster.rótulo ← proximaNovidade;
        proximaNovidade ← proximaNovidade +1;
        novoCluster.tipo ← "novidade";
      Desconhecidos ← Desconhecidos − novoCluster.exemplos;
      novoModelo ← novoModelo ∪ novoCluster;
  retorna *novoModelo*;

Algoritmo 2: Detecção de Novidades interpretada do algoritmo MINAS (**??**).

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

O algoritmo MINAS original (**??**) tem uma implementação de referência[3] (**??**), aqui referida como *Ref*, escrita em Java utilizando os algoritmos base como *K-means* e *CluStream* da biblioteca MOA (**??**).

Além do algoritmo MINAS, existem derivados que estendem a implementação original seguindo sua estrutura básica, duas notáveis são FuzzyND e MINAS-LC. O algoritmo FuzzyND foi proposto em **????**) e incrementa o algoritmo original aplicando teoria de

[3] Disponível em <http://www.facom.ufu.br/~elaine/MINAS>

conjuntos *fuzzy* modificando a representação de *cluster*, o algoritmo de agrupamento, a manutenção do modelo para exemplos conhecidos e para novidades. O algoritmo MINAS-LC foi proposto por **??**) e trata a classificação multi-rótulo, porém não trata evoluções de conceito (*Concept Evolution*). Estes trabalhos foram importantes para entender MINAS e ajudaram a construir uma ideia geral do algoritmo e como proceder para criação de uma versão distribuída.

Algumas propostas de modificação no algoritmo MINAS foram recorrentes durante as discussões no desenvolvimento deste trabalho porém, como estão fora do escopo deste trabalho, ficam somente aqui registradas para algum trabalho futuro. As modificações são:

a) Na fase *offline* a verificação de sobreposição de *clusters* pertencentes a classes distintas e tratamento adequado;

b) Diferentes métodos de cálculo de distância entre pontos além da distância euclidiana;

c) Mudança de representação de *clusters*, atualmente esferas no espaço de dimensão d, para paralelogramos no espaço de dimensão d tratando conjuntos de dados onde as características representadas por cada dimensão são completamente independentes;

d) Em uma implementação de avaliação, ajuste automático de parâmetros otimizando para o máximo de classificações corretas e mínimo tempo de execução para um conjunto de dados fornecido.

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 aplicados à detecção de intrusão em redes **IoT!**.

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. Como mecanismo de detecção de intrusão na construção de um **NIDS!**, duas abordagens são comuns, 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 comparada a detecção por assinatura que tem tempo de resposta 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 pacotes da rede monitorada, extrair as características desses pacotes formando descritores de fluxo 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: classificador, responsável por classificar exemplos; de verificação, responsável por verificar o resultado de classificação; de exemplos rejeitados, responsável por requisitar a um especialista etiquetas para exemplos rejeitados e; de atualização incremental, que atualiza e distribui o modelo aos classificadores.

??) destaca que conjuntos de dados adequados para **NIDS!** são poucos devido ao conjunto de qualidades que eles devem atender, como realismo, validade, etiquetamento,

grande variabilidade e reprodutibilidade (disponibilidade pública).

Para avaliar o desempenho de **NIDS!**, o conjunto de dados MAWIFlow é proposto por **??**). Este conjunto de dados é derivado dos traços de pacotes no *backbone* WIDE, ponto de amostras F (*Packet traces from WIDE backbone, samplepoint-F*), composto por seções diárias de captura de pacotes de 15 minutos de um link de 1 Gbps entre Japão e EUA, com início em 2006 continuamente até hoje, anonimizados e etiquetados por MAWILab (**????**). Desse conjunto de dados original, o conjunto de dados MAWIFlow utiliza apenas os eventos de 2016, dos quais 158 atributos são extraídos resultando em 7.9 TB de captura de pacotes. Por fim, os dados são estratificados para redução de seu tamanho a um centésimo, mantendo as proporções de etiquetas (Ataque e Normal), facilitando o compartilhamento e avaliação de NIDS, além de atender às qualidades anteriormente mencionadas.

Com o conjunto de dados 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 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. Em todos os casos a métrica observada foi a acurácia. Além do conjunto de treinamento, o modelo é atualizado semanalmente com base nas instâncias rejeitadas pelo submódulo de verificação.

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

Em conclusão, a ferramenta BigFlow demonstra capacidade de classificação e detecção de anomalias em fluxos de dados de alta velocidade no contexto de detecção de intrusão. No entanto a atualização semanal e a dependência da avaliação de um especialista não são ideais para detecção de ameaças e respectiva ação sobre a descoberta de novos padrões.

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 gerados descritores de fluxos por uma aplicação *Python* utilizando a biblioteca *flowtbag*[2]. Esses descritores de fluxo 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 virtuais de rede.

A camada de processamento consome o tópico de mensagens que contém os descritores de fluxos da camada de captura e extrai características dos descritores de 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. A última camada da ferramenta fornece uma interface gráfica que apresentada a visualização dos descritores de fluxos processados bem como os conhecimentos extraídos e armazenados no banco de dados. 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 ameaças 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 95 GB de pacotes captura-

[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.

dos, este conjunto de dados é composto de 24 atributos e 16 classes. O outro conjunto, referido como NetOp, foi coletado de um operador de rede que atendia 373 residências na cidade do Rio de Janeiro em 2017. O conjunto NetOp é formado por 5 TB de pacotes capturados e etiquetados por um detector de intrusão comercial.

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

Em conclusão, a ferramenta CATRACA apresenta uma arquitetura dividida em camadas alocadas em ambientes de névoa 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.

3.3 Arquitetura IDSA-IoT

A arquitetura IDSA-IoT, proposta por **??**), tem por objetivo monitorar uma rede local com dispositivos **IoT!** e detectar tentativas de intrusão e alguma subversão do comportamento das transmissões destes dispositivos. O principal destaque da arquitetura é a distribuição de tarefas do sistema de detecção de intrusão entre nós de névoa 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 de nuvem, 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: ECS-Miner (**??**), AnyNovel (**??**) e MINAS (**??**). A avaliação foi feita com o conjunto de dados *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 conjunto de dados tem as características desejáveis de um conjunto para detecção de novidades como: realismo, validade, etiquetas previamente definidas, alta variabilidade, reprodutibilidade e disponibilidade pública. O conjunto de dados *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 proposta de distribuição das tarefas em serviços abre oportunidades para a discussão de diferentes métodos de distribuição dessas tarefas em diferentes ambientes computacionais. Contudo, o algoritmo MINAS que apresentou resultados promissores para este tipo de cenário ainda não foi implementado e avaliado com paralelismo e distribuição que são necessários para tratar fluxos de dados com grandes volumes de dados e velocidades.

3.4 Síntese dos trabalhos relacionados

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.

Um sumário dos trabalhos abordados pode ser visto na Tabela **??**.

Tabela 1 – Sumário dos trabalhos relacionados

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

Capítulo 4

Proposta e Metodologia

Neste trabalho investiga-se uma arquitetura e implementação adequada para executar **ND!** (**ND!**) em ambiente de névoa para detecção de intrusão em redes de dispositivos **IoT!** (**IoT!**). Esta abordagem é baseada na arquitetura IDSA-IoT (**??**) e no algoritmo de **ND!** MINAS (**??**). Nomeada sistema M-FOG, esta implementação do algoritmo MINAS explora computação distribuída habilitando o uso de computadores e dispositivos com recursos limitados presentes no ambiente de névoa a realizar, além de tarefas habituais de um *gateway* **IoT!**, a classificação e detecção de tráfego indesejável.

A arquitetura física do sistema M-FOG pode ser vista na Figura **??**, onde um dispositivo *gateway* observa o tráfego da rede local (sensores IoT) e entre a rede local e a nuvem. A detecção de intrusão é executada de forma paralela e distribuída numa névoa de dispositivos SBC (*Single Board Computer*). Esta arquitetura física foi escolhida por aproximar os recursos comuns em redes deste tipo mas o sistema M-FOG não é limitado a ela, podendo utilizar dispositivos de borda ou névoa com recursos disponíveis para a tarefa de detecção de intrusão ou novidades.

Um **NIDS!** (**NIDS!**) monitora o tráfego em redes e analisa as características de cada fluxo de rede com objetivo de identificar intrusos e tráfegos indesejados. No entanto, este problema requer respostas rápidas e acuradas (**??**): rapidez é necessária para executar uma ação adequada antes que maior dano seja feito à rede, e para lidar com a velocidade do tráfego sem impor atraso ou perda de pacotes na rede observada; acurácia é necessária para não identificar padrões incorretamente, gerando alarmes falsos ou ignorando ataques.

Em cenários **IoT!** comuns, dados podem ser capturados por sensores e pequenos dispositivos computacionais e enviados para a nuvem se mais recursos computacionais ou de armazenamento são necessários. Contudo, para **NIDS!** com requisito de resposta rápida, tal abordagem pode não ser viável. Para atender a essa necessidade a Computação em

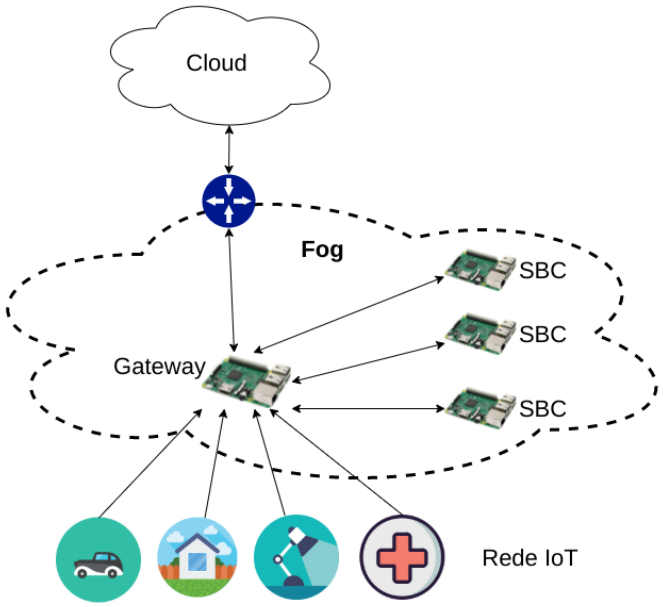


Figura 1 – Arquitetura física do sistema M-FOG.

Névoa é uma alternativa promissora.

A infraestrutura de Computação em Névoa oferece capacidades de realocar parte do processamento e armazenamento dos provedores de nuvem, posicionando dispositivos de capacidade intermediária próximos aos usuários e às fontes de dados. Dada a natureza distribuída e o típico uso de pequenos computadores em cenários de **IoT!** e névoa, alguns desafios para implementação **NIDS!** são notáveis:

(*i*) O tráfego nas bordas da rede é inerentemente distribuído, sem uma entidade centralizadora que tenha acesso a todas as transmissões;

(*ii*) A tarefa de classificação do algoritmo deve ocorrer em paralelo em diferentes nós;

(*iii*) A tarefa de detecção de novidade, que provê a atualização do modelo, deve ser assíncrona;

(*iv*) A complexidade do algoritmo (tempo e espaço) deve permitir sua execução em computadores modestos, isto é, pouca memória ($< 1GB$) e processadores de pouco desempenho.

Nesta proposta, recursos em névoa são combinados para minimizar a latência entre o recebimento (ingestão) do descritor de fluxo e a identificação daquele fluxo. Executa-se múltiplas instâncias da tarefa de classificação do algoritmo MINAS. Após identificado o rótulo do descritor de fluxo (exemplo), este par (rótulo, exemplo) pode ser utilizado imediatamente para tomada de decisões e, se o exemplo é rotulado como "desconhecido", ele é armazenado para posterior execução assíncrona da tarefa de detecção de novidade, portanto sem interromper o processo de identificação.

Além do multiprocessamento da tarefa de classificação e da execução assíncrona da tarefa de detecção de novidade na fase *online*, outra mudança em relação ao MINAS original é a remoção do mecanismo de esquecimento de padrões antigos. Esta remoção tem por finalidade simplificar a atualização e o compartilhamento do modelo entre as múltiplas instâncias e *threads* concorrentes. O impacto desta alteração não é grande, pois o algoritmo MINAS não estabelece a remoção permanente dos *micro-clusters* com pouco uso. Essa alteração aumenta marginalmente a complexidade computacional da classificação, pois inclui na busca os padrões menos utilizados.

Já a fase *offline* e a função de detecção de novidades do algoritmo MINAS permanecem as mesmas.

A arquitetura geral do sistema M-FOG é dividida em dois módulos, Classificação e Detecção de Novidade, que equivalem à fase *online* do algoritmo MINAS. O módulo Classificação executa a tarefa de mesmo nome do algoritmo MINAS e é ponto focal dos esforços de paralelismo desta proposta. Este módulo é replicado em todos os nós do cluster em névoa.

O módulo Detecção de Novidade tem uma instância por cluster, equivalendo a uma instância por rede local. O compartilhamento de modelo entre redes distintas é possível, mas não foi implementado e testado neste trabalho. O módulo Detecção de Novidade é responsável pela tarefa homônima do algoritmo MINAS, recebendo exemplos com rótulo "desconhecido" e os armazenando no conjunto de desconhecidos. Quando o tamanho do conjunto de desconhecidos alcança um valor parametrizado, este módulo executa a função de detecção de novidade do algoritmo MINAS e distribui os novos *micro-clusters* para as instâncias do módulo de Classificação.

4.1 Políticas

O projeto desta arquitetura de **ND!** distribuída inclui o particionamento das funcionalidades de MINAS e o estabelecimento dos fluxos de dados apropriados entre os diferentes atores. Mudanças no posicionamento de cada ator e no comportamento tem impactos diferentes no desempenho e na acurácia do modelo e devem ser escolhidas com cuidado. Alguns dos aspectos com impactos potenciais na implementação incluem:

1. Com relação à alocação do módulo Detecção de Novidade e a sincronização de modelo entre redes distintas para compartilhamento de padrões de novidade local com outras redes que não receberam aquele padrão de ataque:

    a) Em névoa (há um módulo em cada cluster alocado a uma rede local): padrões serão somente detectados se um número suficiente de exemplos deste padrão ocorrer na rede observada; considerando o uso do poder computacional da

névoa, qualquer sincronização de modelo deve ter a propriedade de aditividade; também é preciso reconhecer padrões duplicados;

b) Em nuvem: capacidade de detecção de padrões dispersos em cada rede local onde se analisado não formaria um cluster válido, neste caso todos exemplos com rótulo "desconhecido" são enviados da rede local para a nuvem implicando aumento do uso do *link* de internet e aumento do atraso entre aparição de um padrão, sua detecção e a propagação para os classificadores em névoa;

c) Em ambos (nuvem e névoa): cada conjunto de desconhecidos em cada rede local é mantido bem como a detecção de novidade para padrões locais, quando um exemplo é considerado ruído e removido do conjunto de desconhecidos, o exemplo é enviado à nuvem para detecção de novidade global. Esta opção requer um mecanismo complexo de sincronização de modelo.

2. Com relação ao mecanismo de esquecimento: mesmo quando a detecção de novidade global é usada, modelos locais podem ser otimizados para classificação rápida usando estatísticas locais para ordenar (e remover, usando menos memória) clusters menos utilizados;

3. Por último, na reclassificação do conjunto de desconhecidos, feita pela tarefa de detecção de novidade do algoritmo MINAS, o subconjunto de desconhecidos que pertencem a um novo cluster válido é, efetivamente, classificado com o rótulo deste novo cluster. No Algoritmo **??**, linha **??**, o novo cluster válido inclui o subconjunto de exemplos que o compõem. Portanto, se este conjunto de exemplos fosse classificado com o rótulo atribuído ao novo cluster e adicionado ao fluxo de saída, esta saída teria exemplos atrasados e duplicados em relação ao fluxo de entrada e o resultado obtido poderia ser mais acurado. Além disso, esta escolha modificaria o comportamento do operador de fluxo de dados de um *map*, onde para cada exemplo do fluxo de entrada existe um exemplo no fluxo de saída, para *flatMap*, onde cada exemplo entrada pode ter mais de um exemplo no fluxo de saída.

4.2 Implementações Preliminares

No desenvolvimento desta pesquisa, uma vez determinado o modelo de operação distribuída, com processamento nas bordas da rede, algumas experimentações e algumas ferramentas de teste foram desenvolvidas. Aspectos desses desenvolvimentos são descritos a seguir.

4.2.1 Implementação com *Python* e *Apache Kafka*

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

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

4.2.2 Implementação com *Apache Flink*

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

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

Após desenvolvimento e testes em um computador pessoal, o sistema foi testado no ambiente de testes como descrito na Seção **??**, onde observou-se uma redução enorme no desempenho que concluiu-se ser causada pelo uso excessivo de memória pela plataforma *Apache Flink*. Com a configuração dos parâmetros de memória do cluster, resultados compatíveis com o esperado foram obtidos; no entanto, apesar de exemplos de sucesso na literatura (**??????**), o ambiente de testes não permaneceu estável para execução de repe-

tições do experimento, necessitando reinicializações para que o controlador não ocupasse mais de $1GB$ na segunda execução, o que degradava imensamente o desempenho.

Em conclusão, apesar de promissora, a plataforma *Apache Flink* ainda não suportava a execução em dispositivos computacionais restritos de maneira confiável, sendo a principal barreira o uso excessivo de memória, comum em plataformas do gênero *Big Data*.

4.3 Implementação com MPI

Com os desafios de distribuição e memória encontrados nas implementações preliminares, optou-se por tomar mais controle sobre estes aspectos para maximizar o aproveitamento do *hardware* escolhido, visto que seus recursos são limitados. Para isso escolheu-se implementar a proposta[1] utilizando o padrão **MPI!** e linguagem C, onde tem-se controle fino sobre o uso de memória e distribuição. No entanto, ao escolher-se **MPI!** perde-se facilidades e garantias inclusas nas plataformas tradicionais de fluxos de dados como *Apache Flink* que, entre outras, oferecem o funcionamento confiável e contínuo com mecânicas de recuperação de erros.

A implementação de referência do algoritmo MINAS (**??**) (*Ref*) serve de referência para a construção do modelo distribuído previsto, provendo método de validação dos resultados e servindo também de comparação básica de desempenho.

Os primeiros pontos de divergência do sistema M-FOG e *Ref* são o algoritmo de agrupamento e cálculo de raio. Enquanto *Ref* permite a escolha entre *K-means* e *CluStream* para a fase *offline* e *online*, sistema M-FOG implementa apenas *K-means*. O cálculo de raio em *Ref* é definido com o máximo do conjunto de distância dos exemplos de um *micro-cluster* ao seu centro, seguindo a Equação **??**; enquanto o sistema M-FOG segue a definição em **??**) utilizando o desvio padrão dos valores do mesmo conjunto de distâncias multiplicado pelo parâmetro f_{raio} seguindo a Equação **??**.

Os formatos dos fluxos de dados de entrada e de saída também são notáveis. O formato do fluxo de saída é definido como a tupla contendo o número de sequência do exemplo no fluxo de entrada (*uid*) e a etiqueta de um caractere atribuída ao exemplo. Como entrada, o algoritmo da proposta recebe dois fluxos, o principal é o que contém os exemplos, sendo cada exemplo um vetor de números de dimensão d. O segundo fluxo de entrada consiste de *micro-clusters* representando o modelo inicial criado e capturado da fase de treinamento *offline*.

A fase de treinamento *offline* foi implementada em programa separado da fase *online* e segue o algoritmo MINAS (**??**). Este programa é executado uma vez com o conjunto de treinamento (rotulado com as classes iniciais) como fluxo de entrada e sua saída, que é um fluxo finito de *micro-clusters*, forma o modelo inicial e é capturada em arquivo.

[1] Esta implementação está disponível em <https://github.com/luis-puhl/minas-flink>.

Para avaliação desta proposta, ela foi construída em linguagem C com *OpenMPI 4.0.4*[2], seguindo o paradigma de programação paralela **SPMD!** (**SPMD!**). No paradigma **SPMD!**, uma única versão do programa é inicializada em todos os nós, sendo que para cada instância são passados os parâmetros `mpiSize` e `mpiRank`, que representam o número de nós e o índice de cada nó no cluster. Neste caso, o parâmetro `mpiRank` é o componente de múltiplos dados em **SPMD!** e, como cada nó recebe com um valor diferente, cada nó pode ter um comportamento diferente. Neste quesito, o Algoritmo **??**, linha **??** mostra exatamente esse comportamento no ponto de entrada do sistema M-FOG, dividindo o comportamento de cada nó de acordo com seu `mpiRank` em dois tipos: raiz e folha.

As interações entre os diferentes módulos e ciclo de vida do sistema é ilustrado no diagrama de sequencia na Figura **??**.

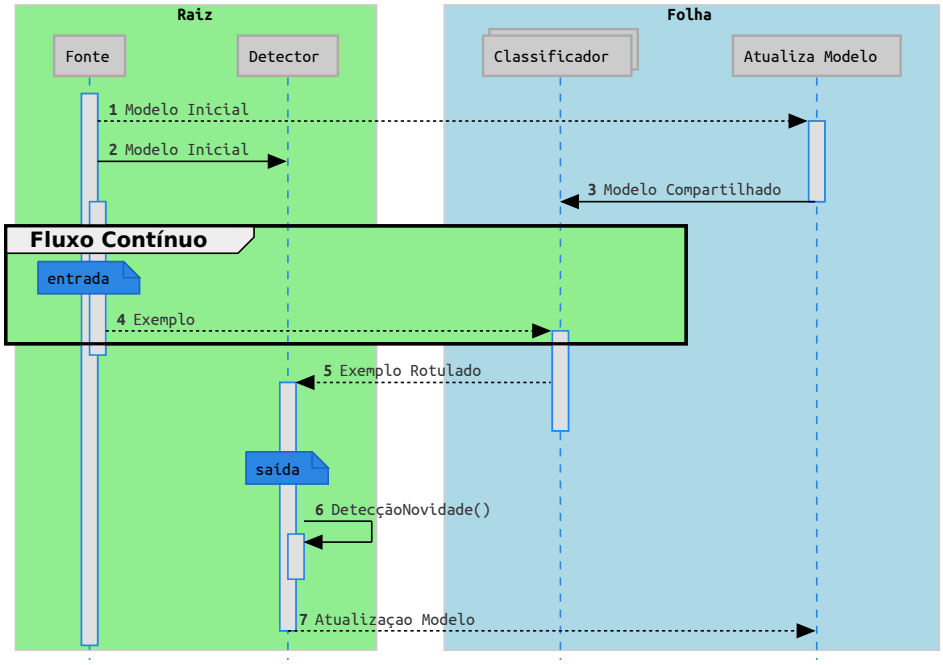


Figura 2 – Diagrama UML de Sequência do sistema M-FOG: visão geral das linhas de vida.

O Algoritmo **??** mostra os primeiros passos do sistema sistema M-FOG, a linha **??** cria Modelo inicial como um conjunto vazio, e a trava (*lock*) utilizada para controle de acesso concorrente das *threads* `Detector` e `Fonte` no nó raiz e `AtualizaModelo` e `Classificador` nos nós folha.

No processo raiz, de `mpiRank` $= 0$ e com acesso aos fluxos de entrada e de saída, uma *thread* com a função `Detector` é iniciada e a função `Fonte` chamada. Nos processos folha, de `mpiRank` > 0, uma *thread* com a função `AtualizaModelo` é iniciada e a função `Classificador` é chamada pela *thread* associada à função principal.

[2] Documentação em <https://www.open-mpi.org/doc/v4.0/>.

```
Parâmetros: mpiRank, mpiSize
Entrada   : fluxoEntrada
Saída     : fluxoSaída
1 Função Mfog(fluxoEntrada, fluxoSaída):
2 |  Modelo ← ∅; trava ← new Trava ();
3 |  se mpiRank == 0 então raiz
4 |  |  new Thread (Detector, [fluxoSaída, Modelo, trava]);
5 |  |  Fonte (fluxoEntrada, Modelo, trava);
6 |  senão folha
7 |  |  new Thread (AtualizaModelo, [Modelo, trava]);
8 |  |  Classificador (Modelo, trava);
```

Algoritmo 3: Sistema M-FOG, ponto de entrada.

Destaca-se que a possibilidade de, ao invés de um `Classificador` por processo e múltiplos processos por nó, pode-se utilizar um processo por nó e múltiplas *threads* executando `Classificador` por nó. Esta estratégia economiza a transmissão de *micro-clusters* para cada nó folha na função `AtualizaModelo`, porém aumenta a concorrência de acesso ao modelo entre as *threads* executando `Classificador`. Em avaliações simples comparando estas estratégias nenhum resultado conclusivo foi observado e há espaço em trabalhos futuros para avaliações mais profundas destas estratégias e outras de estratégias de otimização.

A função `Classificador`, listada no Algoritmo **??**, é o centro do sistema e opera com o modelo atualizado e um exemplo recebido como mensagem do nó raiz. Esta função calcula as distâncias entre o exemplo e todos *micro-clusters* do modelo, encontrando o mais próximo e verificando se o modelo explica aquele exemplo; se explica, o rótulo do *micro-cluster* mais próximo é atribuído como rótulo do exemplo. A função `Classificador`, como as outras, só chega ao seu final se a mensagem recebida (ou instância lida no caso da função `Fonte`) é um marcador de fim de fluxo (`FimDeFluxo`, *end of stream, eos*).

```
1 Função Classificador(Modelo, trava):
2 |  enquanto Verdade faça
3 |  |  exemplo ← recebe (TipoExemplo, raiz);
4 |  |  se exemplo == FimDeFluxo então pare;
5 |  |  exemplo.rótulo ← "desconhecido";
6 |  |  com travaLeitura (trava)
7 |  |  |  (distância, cluster) ← clusterMaisPróximo (exemplo, Modelo);
8 |  |  se distância < cluster.raio então
9 |  |  |  exemplo.rótulo ← cluster.rótulo;
10 |  |  envia (raiz, TipoExemplo, exemplo);
```

Algoritmo 4: Função `Classificador` do nó folha do sistema M-FOG.

A função `AtualizaModelo`, listada no Algoritmo **??**, recebe novos *micro-clusters* como mensagens do nó raiz, independente deste *micro-cluster* ser do modelo inicial, novidade ou

extensão, adequadamente travando o acesso ao modelo compartilhado com a *thread* executando a função `Classificador`. Além da trava de acesso à leitura e escrita na implementação, a função `Classificador` espera por um sinal emitido pela função `AtualizaModelo`, indicando que o modelo está completo antes de começar a busca pelo *micro-clusters* mais próximo.

```
1 Função AtualizaModelo(Modelo, trava):
2 |  enquanto Verdade faça
3 |  |  cluster ← recebe (TipoCluster, raiz);
4 |  |  se cluster == FimDeFluxo então pare;
5 |  |  com travaEscrita (trava)
6 |  |  |  Modelo ← Modelo ∪ cluster;
```

Algoritmo 5: Função `AtualizaModelo` do nó folha do sistema M-FOG.

A função `Fonte`, listada no Algoritmo **??**, lê uma instância do fluxo de entrada que pode ser do tipo exemplo ou do tipo cluster. Se a instância for um exemplo, ele é enviado para um dos nós folha, o nó é escolhido via balanceamento de carga *round-robin*. Se a instância for do tipo cluster, o modelo compartilhado com a *thread* executando a função `Detector` é travado para leitura e escrita e atualizado, e o novo cluster é enviado para todas os nós folhas.

```
1  Função Fonte(fluxoEntrada, Modelo, trava):
2  |  dest ← 1;
3  |  para cada exemplo_i ∈ fluxoEntrada faça
4  |  |  se tipoDe (exemplo) é TipoCluster então
5  |  |  |  broadcast (TipoCluster, exemplo, raiz);
6  |  |  |  com travaEscrita (trava)
7  |  |  |  |  Modelo ← Modelo ∪ exemplo;
8  |  |  |  continue;
9  |  |  envia (dest, TipoExemplo, exemplo);
10 |  |  dest ← dest +1;
11 |  |  se dest > mpiSize então dest ← 1;
```

Algoritmo 6: Função `Fonte` do nó raiz do sistema M-FOG.

Enquanto a função `Fonte` gerencia a entrada de dados, a função `Detector`, listada no Algoritmo **??**, gerencia a saída de dados e, como tem acesso aos exemplos já classificados, também gerencia o conjunto de desconhecidos. Se o tamanho do conjunto de desconhecidos atinge um valor mínimo, a função de detecção de novidade é chamada, os *micro-clusters* representando padrões novidades ou extensões são adicionados ao modelo e enviados para todos os nós folha. Além da remoção dos exemplos utilizados para formar *micro-clusters* de novidades e extensões, exemplos que participaram duas vezes do processo de detecção de novidade são considerados *outliers* ou ruído e são removidos.

```
1  Função Detector(fluxoSaída, Modelo, trava):
2  | Desconhecidos ← ∅; últimaLimpeza ← 0;
3  | enquanto Verdade faça
4  | | exemplo ← recebe (TipoExemplo, qualquer);
5  | | se exemplo == FimDeFluxo então pare;
6  | | fluxoSaída.adicione(exemplo);
7  | | se exemplo.label == unknown então
8  | | | Desconhecidos ← Desconhecidos ∪ exemplo;
9  | | | se | Desconhecidos | ≥ gatilhoDetecçãoNovidade então
10 | | | | novidades ← DetecçãoNovidade (Modelo, *Desconhecidos);
11 | | | | com travaEscrita (trava)
12 | | | | | Modelo ← Modelo ∪ novidades;
13 | | | | para cada cluster ∈ novidades faça
14 | | | | | broadcast (TipoCluster, cluster, raiz);
15 | | | se exemplo.uid > ( últimaLimpeza + janelaLimpeza) então
16 | | | | Desconhecidos ← removeExemplosAntigos (Desconhecidos, últimaLimpeza);
17 | | | | últimaLimpeza ← exemplo.uid;
```

Algoritmo 7: Função `Detector` do nó raiz do sistema M-FOG.

Em suma, a arquitetura do sistema M-FOG é composta de múltiplos nós numa névoa, que detectam intrusão numa rede IoT. Para isso, os nós da rede processam de forma paralela e distribuída o cálculo das distâncias e detecção de novidades, sendo assim um sistema multiprocessado e distribuído. Utiliza-se um único programa com vários processos por nó, e cada processo recebe um conjunto de dados diferentes seguindo o paradigma **SPMD!**.

4.4 Metodologia de Avaliação

A avaliação da proposta aqui apresentada é feita por meio de métricas da literatura, divididas em duas partes: métricas de qualidade de classificação e métricas de escalabilidade.

As métricas de qualidade de classificação escolhidas devem ser adequadas para avaliar detecção de novidades em fluxos de dados, felizmente o tratamento necessário é estabelecido por **??**) e expandido por **????????**). Além do tratamento estabelecido, as métricas adaptadas não são calculadas somente para o conjunto completo capturado no final do fluxo, e sim para cada exemplo $\mathbf{X}$ classificado. Portanto, as métricas têm como índice o contador de exemplos x, informando a posição do exemplo em relação ao fluxo, ou seja, as métricas são de todos os exemplos até a instância x.

Definição 4. *O problema de Detecção de Novidades em fluxo de dados pode ser modelado como uma função f onde, dado o um exemplo $\mathbf{X}$ de um fluxo de dados S, pertencente a uma classe real $c \in \mathbf{C}$, é associado a um rótulo $l \in \mathbf{L}$ podendo ser uma classe conhecida*

$c' \in \mathbf{C}'$, *"desconhecido" ou um rótulo novidade* $y \in \mathbf{Y}$*:*

$$f : \mathbf{X} \mapsto l \quad \mathbf{X} \in S, \quad l \in \mathbf{L} \tag{5}$$

$$\mathbf{C} = \{c_1, c_2, \cdots, c_m\} \tag{6}$$

$$\mathbf{Y} = \{y_1, y_2, \cdots, y_k\} \tag{7}$$

$$\mathbf{L} = \{l_1, l_2, \cdots, l_n\} = \mathbf{C}' \cup \{\textit{"desconhecido"}\} \cup \mathbf{Y} \tag{8}$$

O tratamento estabelecido das métricas de qualidade para mineração de fluxos de dados define que as métricas sejam extraídas de uma matriz de erro de classificação multi-classe $\mathbf{E}_x$ na Equação **??**, adaptada para detecção de novidade.

Definição 5. *Uma matriz de confusão* $\mathbf{E}$ *para a instância* x*, de dimensão* $m \times n$ *para* m *classes e* n *rótulos, é preenchida com o número de instâncias da classe* c_i *classificados com o rótulo* l_j*.*

$$\mathbf{E}_x = (e_{ij}) \in \mathbb{N}^{m \times n} = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m,1} & e_{m,2} & \cdots & e_{m,n} \end{pmatrix} \tag{9}$$

Note que a soma de todos os elementos da matriz de confusão é igual a x, ou seja, a contagem de todos exemplos.

Um dos possíveis rótulos $l_j \in \mathbf{L}$ é o rótulo "desconhecido", que indica um erro de classificação diferente dos demais e é tratado separadamente. A taxa de desconhecidos *UnkR* na Equação **??** é uma das métricas escolhidas pois trata deste tipo específico de erro (**??**).

Definição 6. *A taxa de desconhecidos* $UnkR_x$ *para um instante* x *é definida como a média da taxa de desconhecidos de cada classe (**??**). A taxa de desconhecidos de cada classe* c_i *é definida como o elemento* $e_{ij} \in \mathbf{E}_x$ *onde* j *é a coluna do rótulo "desconhecido" na matriz de rótulos* $\mathbf{L}$ *dividido pelo número de exemplos da classe.*

$$UnkR_x = \frac{1}{m} \sum_{i=1}^{m} UnkR_{x,i} \tag{10}$$

$$UnkR_{x,i} = \frac{e_{ij} : l_j = \textit{"desconhecido"}}{\sum_{j=1}^{n} e_{ij}} \tag{11}$$

Para todos os rótulos l_j diferentes de "desconhecido", uma classe c_i é associada se o rótulo l_j é procedente do modelo inicial, ou seja $l_j = c_i : c_i \in \mathbf{C}$ ou, em último caso, o rótulo l_j é associado à classe com o maior número de exemplos com o rótulo l_j, ou seja $e_{ij} = max\{e_{aj} : a \in [0, m]\}$ para $c_i \in \mathbf{C}$ e $l_j \in \mathbf{L}$ (**??**).

Definição 7. *A função de associações $A(l)$ é definida como:*

$$A(l_j) : l_j \in \mathbf{L} \mapsto c_i \in \mathbf{C} = \begin{cases} \textit{indefinido} & \textit{se } l_j = \textit{“desconhecido”} \\ c_i & \textit{se } \exists c_i = l_j \quad : c_i \in \mathbf{C}' \\ c_i & \textit{se } e_{ij} = max\{e_{aj} :\in [0, m]\} \end{cases} \tag{12}$$

No contexto de classificação multi-classe, a acurácia acc_x para um instante x é definida como a média da acurácia de cada classe (Eq. **??**). A acurácia de cada classe c_i no instante x, de forma semelhante à definição da matriz de confusão, é definida como extensão da acurácia da classificação binária (Eq. **??**). Para classificação binária a acurácia calculada com os valores verdadeiro-positivo (tp), verdadeiro-negativo (tn), falso-positivo (fp) e falso-negativo (fn). Para cada classe c_i, o valor verdadeiro-positivo (tp_i) é definido como número de exemplos onde o rótulo do exemplo l é associado à classe c_i (Eq. **??**). O valor falso-negativo (fn_i) são os exemplos da classe onde o rótulo do exemplo não é associado à classe c_i e o rótulo não é “desconhecido” (Eq. **??**). Os valores verdadeiro-negativo (tn) e falso-positivo (fp) são zero.

$$tp_i = \sum_{j=1}^{n} e_{ij} \quad \text{se } l_j \neq \text{“desconhecido” e } A(l_j) = c_i \tag{13}$$

$$fn_i = \sum_{j=1}^{n} e_{ij} \quad \text{se } l_j \neq \text{“desconhecido” e } A(l_j) \neq c_i \tag{14}$$

$$acc_i = \frac{tp + tn}{tp + fn + fp + tn} = \frac{tp_i}{fn_i + tp_i} \tag{15}$$

$$acc_x = \frac{1}{m} \sum_{i=1}^{m} acc_i \tag{16}$$

Concluindo as métricas de qualidade de classificação, a métrica de erro combinado (err) é a média do erro de cada classe, sendo o erro de cada classe o valor falso-negativo (fn_i) dividido pelo número de exemplos da classe.

$$err = \frac{1}{m} \sum_{i=1}^{m} \frac{fn_i}{fn_i + tp_i} \tag{17}$$

Para a validação da corretude da implementação do sistema M-FOG com relação ao algoritmo MINAS original, ambos são executados com o mesmo *data set* de avaliação e as métricas de qualidade de classificação são comparadas.

Um programa de avaliação foi construído seguindo as técnicas de referência como matriz de confusão multi-classe com associação de classe de rótulo (**??**) para extrair medidas de qualidade de classificação. Este programa recebe duas entradas, o conjunto de dados de teste e o fluxo de saída capturado. Com estas informações o programa gera a matriz de confusão do instante final, associação de classes e rótulos, resumo de qualidade

final com *Hits* (acurácia), *Misses* (erro), *Unknowns* (taxa de desconhecidos) e, um gráfico de visualização do fluxo com as métricas e marcadores de novidades para cada instante do fluxo de saída capturado.

As métricas de escalabilidade extraídas são o número de processadores, tempo de processamento e latência de eventos e, estas métricas permitem o cálculo de *speedup* prático.

Capítulo 5

Experimentos e Resultados

Este capítulo apresenta o ambiente experimental e os resultados obtidos dos experimentos, discutindo as métricas de qualidade da classificação e escalabilidade do sistema M-FOG.

5.1 Ambiente de Teste

Com o objetivo de avaliar esta proposta e averiguar os efeitos da distribuição da detecção de novidades em um cenário **IoT!**, construiu-se um ambiente experimental de névoa.

Este ambiente é composto por três computadores de única placa (*Single Board Computer*) modelo Raspberry Pi 3 model B, equipados com o SoC (*system-on-chip* - sistema em um chip) de arquitetura ARM *BCM2837* com 4 núcleos de processamento à frequência de $1.2\,GHz$, $32\,kB$ e $512\,kB$ de memória *cache* nível 1 e 2 respectivamente, $1\,GB$ de memória RAM, armazenamento em cartão SD e conectados por rede cabeada *Ethernet*.

A ideia central é criar um cluster simples simulando *gateway* de uma rede **IoT!** com recursos limitados. Este cluster armazenou todo o código-fonte, binários (compilados no mesmo cluster) e *data set*. Nesta configuração, o *data set* é armazenado no cartão SD do nó raiz e é lido para cada execução do experimento. Todos os experimentos foram executados neste cluster para isolamento de outras variações imprevistas e para garantir que as comparações seriam justas com *software* e *hardware* constante.

O *data set Kyoto 2006+*[1], tráfego das *Honeypots* da Universidade de Kyoto, é a referência de um cenário real para este trabalho. Este *data set* contém dados ainda representativos (até 2015) e as característica desejáveis de um conjunto de dados (realismo,

[1] Disponível em <http://www.takakura.com/Kyoto_data/>.

validade, etiquetas previamente definidas, alta variabilidade, reprodutibilidade e disponibilidade pública) são atendidas (**????**).

O segmento utilizado deste *data set* é o de Dezembro de 2015, contendo 7 865 245 instâncias. Deste segmento, são mantidas apenas instâncias associadas a tráfego normal ou a ataques conhecidos, identificados por **NIDS!** que tenham mais de 10 000 instâncias para significância, como feito previamente por **??**). As instâncias mantidas são normalizadas para que o valor de cada característica original (como endereço IP, duração do fluxo, serviço) seja transposta para o intervalo Real $[0, 1]$.

O conjunto resultante da operação recém descrita é então dividido em dois conjuntos, treinamento e teste. Para avaliar a detecção de ataques, o conjunto treinamento é composto apenas de tráfego normal, contendo 72 000 instâncias. O conjunto de teste possui 653 457 instâncias, sendo elas 206 278 instâncias com classe "N" (normal) e 447 179 instâncias com classe "A" (ataque).

Destaca-se que essa manipulação do *data set* causa *Overfitting* para a classe normal e *under-fitting* para a classe ataque, pois o sistema primeiro precisa detectar um padrão novidade para então adicionar ao modelo. Como o foco deste trabalho é na arquitetura e metodologia de paralelização e distribuição em névoa, a otimização do processo de detecção de intrusão não foi abordada, deixando a questão de escolha do *data set* e do processo de sua manipulação em aberto.

Quanto aos parâmetros do algoritmo MINAS, ilustrados na Figura **??**, os valores foram escolhidos por serem valores comuns para o algoritmo presentes na literatura (**????**) e na implementação de referência *Ref* (**??**). Os parâmetros que não tem valores comuns na literatura foram escolhidos e ajustados até os os resultados obtidos se aproximarem aos resultados da implementação de referência *Ref*.

```
MinasParams minasParams = {
  .k=100, .dim=22, .precision=1.0e-08,
  .radiusF=0.25, .minExamplesPerCluster=20, .noveltyF=1.4,
  .thresholdForgettingPast = 10000,
};
```

Figura 3 – Parâmetros do algoritmo MINAS.

Os parâmetros utilizados da literatura são `k`, que é o número de *micro-clusters* gerados pelo algoritmo de agrupamento, `minExamplesPerCluster`, que indica o número mínimo de exemplos para um *micro-cluster* válido (representatividade) e, `thresholdForgettingPast`, que estabelece o limite para remoção de exemplos do conjunto de desconhecidos.

Os parâmetros escolhidos por aproximação de resultados são: `precision`, que é o valor limite para melhora na distância global reduzindo as iterações no algoritmo de agrupamento (otimização), `radiusF`, que corresponde ao fator que multiplica o desvio

padrão das distâncias entre o centro e cada exemplo formador do novo *micro-cluster* definindo o raio do *micro-cluster* e, `noveltyF`, que é o fator que multiplica o desvio padrão das distâncias do *micro-cluster* mais próximo distinguindo um novo padrão entre extensão e novidade.

5.2 Experimentos, Métricas e Visualizações

Seguindo as especificações de ambiente da Seção **??**, cada experimento consiste na execução de um dos programas, com variação do parâmetro de paralelismo, como mostra a Tabela **??**, fornecendo um modelo inicial e *data set* de teste como fluxo de entrada e capturando o fluxo de saída para e extração das métricas estabelecidas na Seção **??**. Destes experimentos os seguintes resultados são apresentados.

Tabela 2 – Listagem dos principais experimentos.

Experimento	**Programa**	**Características**
a-Referência	MINAS referência 2013	Raio é a distância máxima.
b-Sequencial	MINAS sequencial para validação	Raio é o desvio padrão das distâncias; Modelo único; Remoção de desconhecidos mais agressivo.
c-Paralelo	sistema M-FOG 1 nó, 4 processadores	Classificadores paralelos; Detecção de novidade assíncrona.
d-Distribuído	sistema M-FOG 3 nós, 12 processadores	Mais processadores; Comunicação em rede.

5.2.1 Validação do algoritmo

Para validar a biblioteca de funções do sistema M-FOG que implementa o algoritmo MINAS, uma versão do Algoritmo **??** foi construída, sem classificadores paralelos ou detecção de novidade assíncrona, aspectos que caracterizam o sistema M-FOG. Esta implementação utiliza a definição de raio da Equação **??** (desvio padrão das distâncias). Feita esta implementação foram realizados os experimentos *b-Sequencial* com esta implementação e *a-Referência* com a implementação MINAS referência. Estes dois experimentos utilizam apenas um nó e um processador do ambiente experimental, sendo que as métricas são extraídas conforme a Seção **??** e são comparadas nesta Subseção, mostrando a equivalência entre as implementações.

A Tabela **??** mostra a matriz de confusão no instante final do fluxo de saída do experimento *a-Referência*. Nesta matriz, o rótulo "desconhecido" é o caractere "−" e os valores de associação e verdadeiro-positivo estão atrelados à coluna de cada rótulo; no demais a matriz segue os atributos definidos na Seção **??**. Esta matriz é comparada com a matriz de confusão do mesmo instante do fluxo de saída do experimento *b-Sequencial*, vista na Tabela **??**.

Tabela 3 – Experimento *a-Referência*, Matriz de confusão do *data set Kyoto* Dez. 2015.

Rótulos	-	N	1	2	3	4	5	6	7	8	9	10	11	12
Classes														
A	3 774	438 750	123	145	368	8	52	165	1	1 046	161	2 489	71	26
N	8 206	193 030	0	79	44	0	0	0	229	181	154	4 066	289	0
Associação	-	N	A	A	A	A	A	A	N	A	A	N	N	A
Hits (tp)	0	193 030	123	145	368	8	52	165	229	1 046	161	4 066	289	26

Comparando as duas matrizes, a primeira diferença aparente é o índice do primeiro rótulo novidade (0 ao invés de 1), mas isso se deve apenas à decisão arbitrária de íncide de novidades. A primeira diferença notável que influi nas métricas de qualidade é a falta de 3 rótulos novidade, 12 rótulos no experimento *a-Referência* ($[1, 12]$) e 9 no experimento *b-Sequencial* ($[0, 10]$ exceto $\{3, 9\}$). Também nota-se o número de exemplos desconhecidos, 28 567 no experimento *b-Sequencial* e 11 980 no experimento *a-Referência* e, o montante de exemplos incluídos nos rótulos novidade, muito menor no experimento *b-Sequencial*.

Tabela 4 – Experimento *b-Sequencial*, Matriz de confusão do *data set Kyoto* Dez. 2015.

Rótulos	-	N	0	1	2	4	5	6	7	8	10
Classes											
A	16 086	429 765	94	995	104	0	23	3	29	46	34
N	12 481	193 642	3	94	0	47	0	0	0	11	0
Associação	-	N	A	A	A	N	A	A	A	A	A
Hits (tp)	0	193 642	94	995	104	47	23	3	29	46	34

Estas três diferenças são atribuídas à divergência entre as definições de raio discutidas na Seção **??**, Equações **??** e **??**. Esta divergência, apesar de inicialmente pequena, muda muito o comportamento do passo de classificação, resultando em um conjunto diferente de desconhecidos e, por fim, mudando muito o resultado da detecção de novidades. Adicionalmente, como o parâmetro f_{raio} (`radiusF`) não existe na versão com raio definido como distância máxima (Eq. **??**), este parâmetro foi escolhido experimentalmente observando os resultados durante a implementação.

Além das matrizes de confusão que dão uma visão detalhada, porém somente sobre o último instante do fluxo, pode-se comparar as visualizações de fluxo onde as métricas de qualidade são apresentadas para todo o fluxo. As Figuras **??** e **??** ilustram os fluxos de saída dos experimentos *a-Referência* e *b-Sequencial* respectivamente.

As Figuras **??** e **??** reforçam a observação anterior de que mais rótulos novidade (linhas tracejadas verticais com rótulo no topo) foram encontrados no experimento *a-Referência* do que as encontradas no experimento *b-Sequencial*. Outro aspecto com relação a rótulos novidade, no experimento *a-Referência* eles surgem mais "cedo" no fluxo, sendo a primeira marcação em $x = 58\,967$ contra $x = 94\,155$ no experimento *b-Sequencial*.

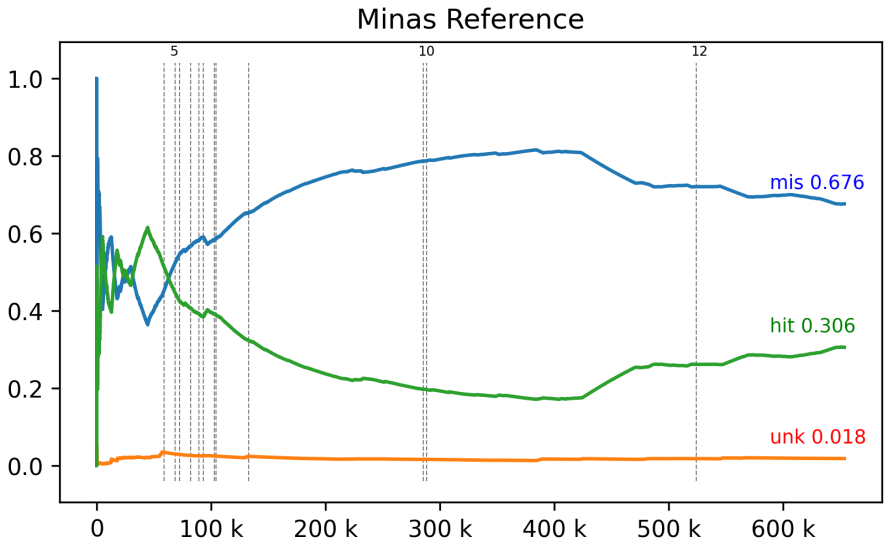


Figura 4 – Experimento *a-Referência*, visualização de fluxo do *data set Kyoto* Dez. 2015.

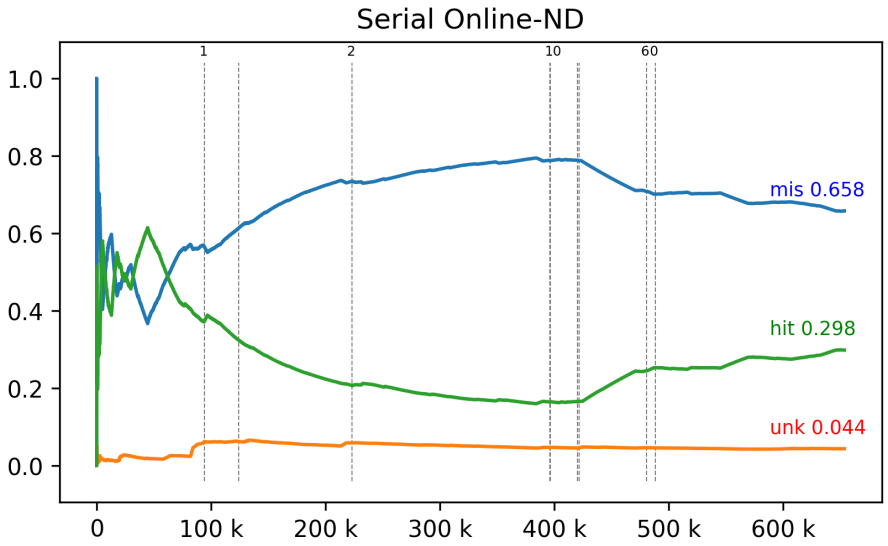


Figura 5 – Experimento *b-Sequencial*, visualização de fluxo do *data set Kyoto* Dez. 2015.

O gráfico de visualização do fluxo tem no eixo horizontal (x, domínio) o índice do exemplo x e o eixo vertical (y, imagem) mostra o valor das métricas de qualidade calculadas até aquele índice de exemplo x no fluxo de saída capturado. As linhas horizontais na visualização de fluxo mostram o progresso das métricas de qualidade de classificação durante o fluxo e, ao final na esquerda, o valor impresso junto e de mesma cor da linha é o valor final de cada métrica. As métricas são: em laranja a taxa de desconhecidos (`unk`), em verde a acurácia (`hit`) e em azul o erro (`err`). Estas métricas podem ser arranjadas em uma tripla-sumário (`unk`, `hit`, `err`) com os valores em pontos percentuais para fácil comparação.

Em conclusão das métricas de qualidade de classificação, os experimentos *a-Referência* e *b-Sequencial* não são idênticos, pois não implementam a mesma definição para o raio,

porém são equivalentes em seus resultados. Enquanto o experimento *a-Referência* tem tripla-sumário (1.8%, 30.6%, 67.6%), o experimento *b-Sequencial* piora marginalmente nas métricas com (4.4%, 29.8%, 65.8%), tendo mais desconhecidos, menor acurácia e menor erro.

Para as métricas de escalabilidade, os experimentos *a-Referência* e *b-Sequencial* servem somente como base para os próximos experimentos onde o paralelismo passa a ser utilizado. Uma última nota quanto ao tempo de execução, mesmo não sendo paralelo, as otimizações de memória, redução do uso de *strings* e condição de parada do algoritmo *K-Means*, reduziram o tempo utilizado pela implementação de referência de 2 772s (917s *offline* e 1 845s *online*) para 287s (194s *offline* e 93s *online* no experimento *b-Sequencial*) na nova implementação.

5.2.2 Paralelismo e Distribuição

Seguindo as comparações, para avaliar a escalabilidade dos sistemas, foram realizados dois experimentos que seguem o mesmo roteiro, o experimento *c-Paralelo* (1 nó e 4 núcleos) e o experimento *d-Distribuído* (3 nós e 12 núcleos). As Tabelas **??** e **??** mostram as matrizes de confusão dos experimentos *c-Paralelo* e *d-Distribuído* respectivamente.

Tabela 5 – Experimento *c-Paralelo*, sistema M-FOG com 1 nó e 4 núcleos, Matriz de confusão do *data set Kyoto* Dez. 2015.

Rótulos	-	N	0	1	2	3	4
Classes							
A	12 282	433 797	147	952	0	0	1
N	3 088	203 019	40	99	27	5	0
Associação	-	N	A	A	N	N	A
Hits (*tp*)	0	203 019	147	952	27	5	1

A tripla-sumário do experimento *c-Paralelo* é (2.4%, 31.2%, 66.4%), com menos desconhecidos em relação ao experimento *b-Sequencial* (4.4%, 29.8%, 65.8%). A redução de desconhecidos é de 2%, sendo distribuída 1.4% em acurácia e 0.6% em erro, representando melhora marginal em todas as métricas.

Comparando as matrizes dos experimentos *b-Sequencial* e *c-Paralelo*, observa-se os efeitos do paralelismo de classificadores e, mais importante, a execução assíncrona da tarefa de detecção de novidade. O impacto principal é a redução no número de rótulos novidade, de 9 para 5. Comparando a visualização de fluxo, Figuras **??** e **??**, observa-se também que a aparição do primeiro padrão novidade acontece mais "tarde"; enquanto em *b-Sequencial* o rótulo novidade "1" é observado em $x = 94\,155$, em *c-Paralelo* o mesmo rótulo é observado em $x = 172\,917$.

Esse comportamento é devido à detecção de novidade assíncrona, pois enquanto essa tarefa é executada os classificadores continuam utilizando o modelo antigo. Além disso,

como já foi elencado na Seção **??** item **??**, os exemplos utilizados para formar o novo *micro-cluster* com rótulo novidade não são recolocados no fluxo de saída para nova análise. Estas duas características do sistema M-FOG acarretam em um atraso significativo entre a aparição de um padrão no fluxo de entrada e a aparição do rótulo correspondente no fluxo de saída. Em outras palavras, o fluxo avança com o modelo antigo e, mesmo após atualizado, só gera uma saída com o rótulo novo quando (e se) o padrão aparece novamente. Este é um resultado muito importante para compreender os problemas relacionados à distribuição de um algoritmo deste gênero.

Tabela 6 – Experimento *d-Distribuído*, sistema M-FOG com 3 nós de 4 núcleos cada, Matriz de confusão do *data set Kyoto* Dez. 2015.

Rótulos	-	N	0	1	2	3	4
Classes							
A	12 378	433 631	117	886	0	162	5
N	3 121	202 916	40	96	105	0	0
Associação	-	N	A	A	N	A	A
Hits (*tp*)	0	202 916	117	886	105	162	5

Avançando para o experimento *d-Distribuído*, a matriz de confusão na Tabela **??**, mostra pouca diferença quando comparado ao experimento *c-Paralelo*, com o número de desconhecidos sofrendo a maior variação.

O mesmo é observado quando compara-se a visualização de fluxo, Figuras **??** e **??**, há pouca diferença e a tripla-sumário (2.4%, 31.2%, 66.4%) é idêntica.

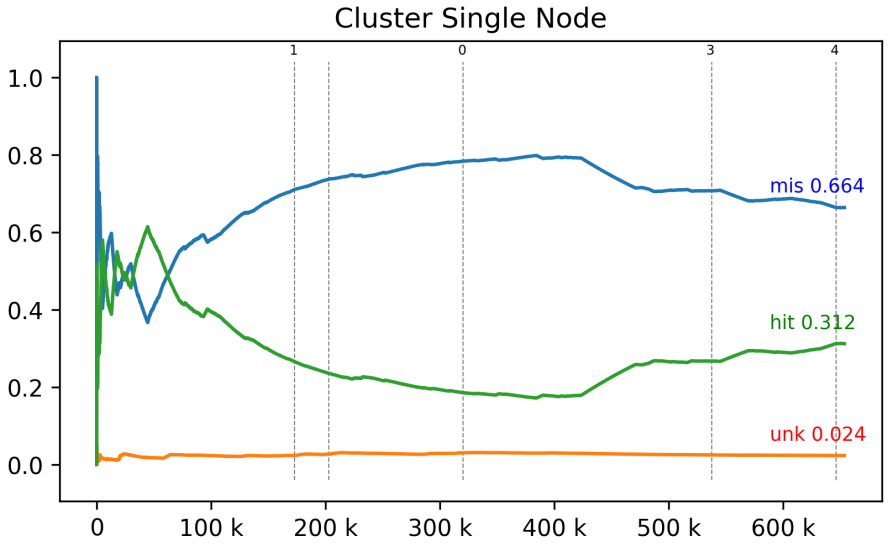


Figura 6 – Experimento *c-Paralelo*, sistema M-FOG com 1 nó e 4 núcleos, visualização de fluxo do *data set Kyoto* Dez. 2015.

Isto reforça a ideia de que o maior impacto nas métricas de qualidade de classificação é a mudança de detecção de novidade assíncrona, pois mesmo com o triplo de instâncias classificadoras, as métricas continuam idênticas.

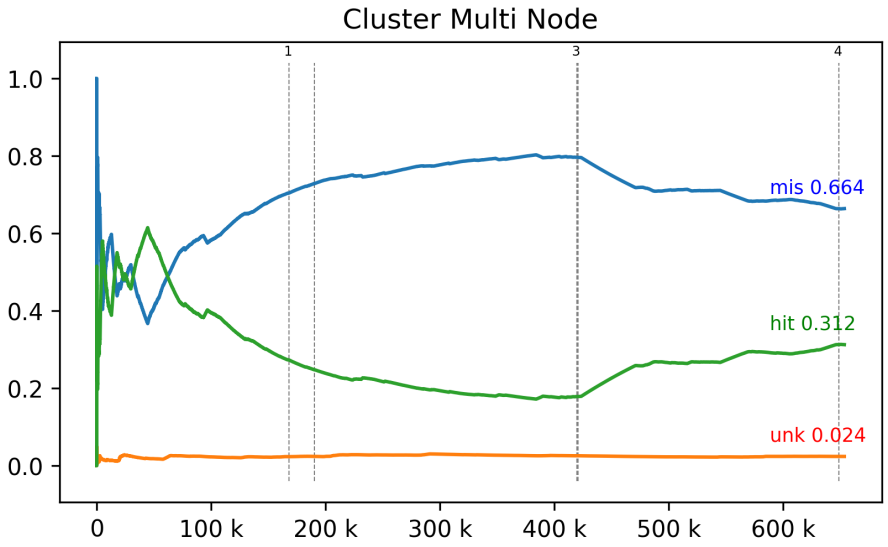


Figura 7 – Experimento *d-Distribuído*, sistema M-FOG com 3 nós de 4 núcleos cada, visualização de fluxo do *data set Kyoto* Dez. 2015.

Com a discussão das métricas de qualidade de classificação concluída, resta abordar as métricas de escalabilidade que são muito mais simples em sua definição e extração porém de muito interesse a este trabalho.

Essas métricas de escalabilidade extraídas dos experimentos são número de núcleos de processamento utilizados, *Tempo*, *Sistema* e *Decorrido*, representados em segundos, e obtidas via *GNU Time 1.9* durante a execução do experimento em questão. *Tempo* é a quantidade de segundos de CPU gastos no modo de usuário (indica o tempo usado em computação intensiva, por exemplo cálculos matemáticos). *Sistema* é a quantidade de segundos de CPU gastos no modo *kernel* (para nosso caso, indica o tempo de entrada ou saída). *Decorrido* é o tempo decorrido do mundo real (relógio de parede) e indica quanto tempo o programa levou para ser concluído. Quanto mais baixos os tempos, melhor.

Os quatro experimentos principais são mostrados na Tabela **??**, relacionando as métricas de qualidade de classificação da tripla-sumário (`unk`, `hit`, `err`) e as métricas de escalabilidade.

Na Tabela **??**, a coluna *Offline* representa os valores de métricas de tempo de execução para a fase *Offline* (treinamento) nos experimentos utilizando o sistema M-FOG (*b-Sequencial*, *c-Paralelo* e *d-Distribuído*). Esta distinção é necessária pois a implementação de referência do algoritmo MINAS executa as duas fases sem separação. Já o tempo de treinamento no sistema M-FOG é mostrado na coluna *Offline*. Uma modificação da implementação de referência mostrou que dos 2 772.07 segundos listados na tabela, 917s são da fase *offline* e 1 845s da fase *online*.

Além das métricas extraídas dos experimentos, as métricas de escalabilidade calculadas a partir dos valores extraídos são a latência de eventos e *speedup*. A latência média é calculada com o tempo decorrido dividido pelo o tamanho do *data set* e pode ser vista na

Tabela 7 – Sumário das métricas extraídas dos experimentos principais.

Experimento Métrica	*a-Referência*	*Offline*	*b-Sequencial*	*c-Paralelo*	*d-Distribuído*
`unk`	11980 0.018333		28567 0.043717	15370 0.023521	15499 0.023718
`hit`	199708 0.305618		195017 0.298438	204151 0.312416	204191 0.312478
`err`	441769 0.676049		429873 0.657843	433936 0.664061	433767 0.663802
Tempo (s)	2761.83	194.12	80.79000	522.1000	207.1400
Sistema (s)	7.15	0.075	11.51000	47.7700	157.6100
Decorrido (s)	2772.07	194.27	93.03000	145.0400	95.3800
Latência (s)	$4.24 \cdot 10^{-3}$		$1.42 \cdot 10^{-4}$	$2.22 \cdot 10^{-4}$	$1.46 \cdot 10^{-4}$
Processadores	1	1	1	4	12
Speedup				0.6414092	0.9753617
Eficiência				0.1603523	0.0812801

Tabela **??**. O *speedup* é calculado com a divisão do tempo decorrido na versão sequencial divido pelo tempo utilizado pela versão paralela. A eficiência é calculada com a divisão do tempo decorrido na versão sequencial divido pelo produto do número de processadores utilizado e o tempo utilizado pela versão paralela.

Observando as métricas de escalabilidade nota-se que o sistema M-FOG não fornece *speedup* e eficiência em relação à versão sequencial. Isto indica que, apesar da premissa de independência de classificadores paralelos, a comunicação entre os processos utilizando dois conjuntos de instruções **MPI!** *send* e *receive* para cada exemplo não é eficiente para o processamento de fluxo de dados.

5.2.3 Experimentos Adicionais

Experimentos adicionais foram realizados para melhor compreender o comportamento do sistema M-FOG para número variado de processadores entre 1 e 12 utilizando de 1 a 3 nós. Os resultados obtidos, ilustrados na Figura **??**, são limitados ao nó raiz, e mostram um aumento do tempo *Sistema* (operações em modo *kernel*). Este aumento indica maior carga associada às operações de entrada e saída das instruções *send* e *receive* reforçando a ideia apresentada na Subseção **??**.

Outros experimentos adicionais foram realizados para averiguar a latência para cada exemplo do fluxo, mostrando o comportamento de cada implementação, o que é ilustrado na Figura **??**. No entanto, é importante destacar que esta métrica não foi extraída dos experimentos principais, sendo apenas ilustrativa dos diferentes comportamentos.

O primeiro comportamento é ilustrado na Figura **??**, onde a execução da tarefa de

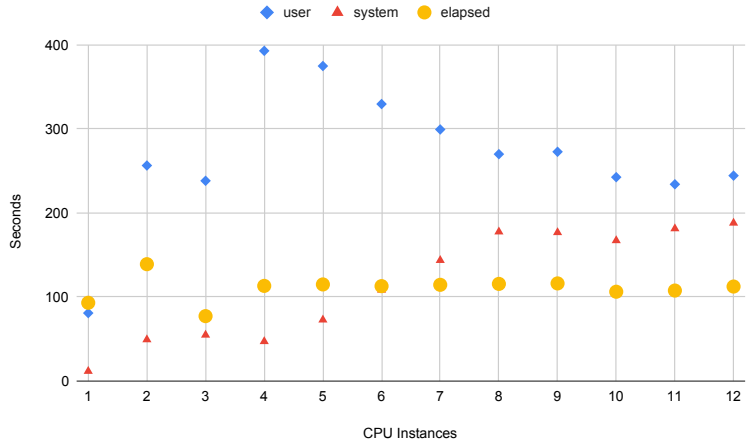
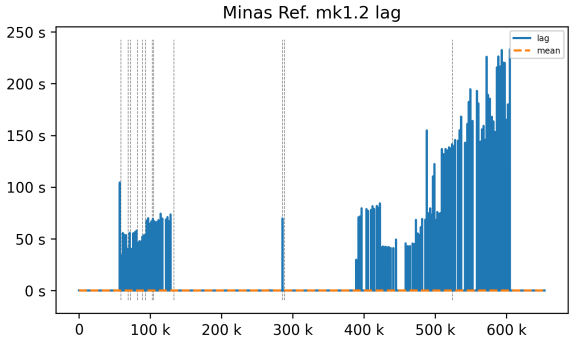
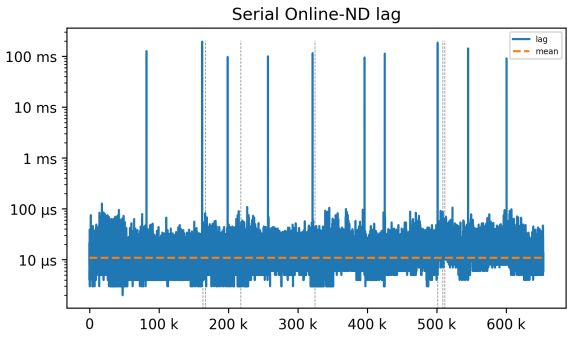


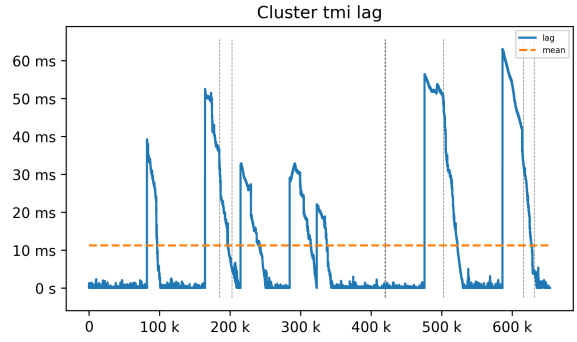
Figura 8 – Métricas de tempo para execuções do sistema M-FOG com variação no número de processadores.



(a) Implementação de referência.



(b) Implementação sequencial.



(c) Implementação paralela.

Figura 9 – Visualização de Latência das implementações de referência, sequencial e paralela do algoritmo MINAS.

detecção de novidade gera picos de 50 a 200 segundos na implementação de referência do algoritmo MINAS. Um comportamento semelhante é visto na Figura **??** onde ainda há picos de latência, porém não tão intensos e frequentes na implementação sequencial do algoritmo MINAS utilizado a biblioteca do sistema M-FOG. Esta diferença está relacionada com mecanismo de remoção de desconhecidos e ausência do mecanismo de esquecimento de *micro-clusters* menos utilizados no Modelo. Já o comportamento visto na Figura **??**,

extraído do sistema M-FOG, mostra o efeito de *buffers* de comunicação entre diferentes processos onde o pico de latência da detecção de novidade cai progressivamente conforme os buffers são esvaziados já que o processo de detecção de novidade é assíncrono, porém ocupa a *thread* responsável pelo fluxo de saída do sistema M-FOG.

5.3 Conclusão

Os experimentos executados e os resultados obtidos demonstram que a **ND!** (**ND!**) executada de maneira distribuída em ambiente de névoa é uma estratégia válida para detecção de mudanças nos padrões de tráfego, sendo viável como parte de um **NIDS!** (**NIDS!**) para redes **IoT!**.

No entanto para aplicação em uma situação real, são necessárias otimizações desde o *data set* de treinamento e avaliação, passando pelo algoritmo de detecção de novidade, parâmetros do algoritmo e estratégias de paralelismo e distribuição.

Capítulo 6

Conclusão

Segurança e privacidade são uma grande preocupação em **IoT!** (**IoT!**), seja para a preservação do sigilo de dados pessoais ou para evitar a subversão do dispositivo para uso em uma *botnet*. Nesses cenários, o monitoramento da rede para detecção de possíveis intrusões é uma tarefa necessária. Atendendo a essa necessidade, um **NIDS!** (**NIDS!**) é uma ferramenta que observa o tráfego da rede e alerta para possíveis ataques. Este alerta é gerado através de detecção de anomalias, e uma técnica para detectar anomalias são algoritmos de **ND!** (**ND!**). O algoritmo MINAS é um algoritmo de detecção de novidade que já foi avaliado e considerado promissor para aplicação em **NIDS!**.

Para um bom funcionamento de um **NIDS!**, é necessário que este responda rapidamente aos padrões do fluxo da rede. Portanto, para aplicações em redes **IoT!** o envio das observações de uma rede remota para processamento em nuvem pode resultar em latências proibitivas. Uma alternativa viável à computação em nuvem neste caso é a computação em névoa que pode disponibilizar recursos computacionais mais próximos à fonte de dados, mesmo que de menor capacidade, oferece menor latência em seus processamentos. Porém, a necessidade de resposta rápida de um **NIDS!** e a natureza distribuída da computação em névoa requer que as técnicas utilizadas estejam preparadas para um cenário paralelo e distribuído, o que não acontece com o algoritmo MINAS.

Este trabalho apresenta a construção e avaliação do sistema M-FOG, uma implementação paralela e distribuída com foco na execução em névoa de dispositivos **IoT!** do algoritmo MINAS, pautado em um caso de uso para **NIDS!**. Esta implementação foi construída com **MPI!** buscando a escalabilidade na tarefa de processamento de fluxo de dados e economia dos recursos limitados comumente encontrados em sistemas **IoT!**, seguindo a arquitetura IDSA-IoT (**??**).

A avaliação do sistema construído utilizou o conjunto de dados (*data set*) *Kyoto 2015*,

relevante para **NIDS!** por conter descritores de fluxo de uma rede de *Honeypots*. A manipulação do *data set* utilizado, apesar de não focada em **NIDS!**, foi suficiente para avaliar a implementação e comparar resultados.

As métricas de qualidade de classificação escolhidas mostraram resultado equivalente entre a nova implementação e a implementação de referência do algoritmo MINAS, seja na versão sequencial, paralela com 4 processadores ou distribuída com 3 nós. As métricas de escalabilidade do sistema M-FOG em ambiente de névoa **IoT!** mostraram melhora significante comparada à implementação de referência no mesmo ambiente. No entanto não foi observada melhora compatível com o aumento no número de nós no cluster, ou seja, a escalabilidade almejada não foi alcançada.

As principais contribuições deste trabalho são a análise, implementação revisada e melhora do desempenho do algoritmo MINAS guiado pelo contexto de computação distribuída em névoa de dispositivos **IoT!** e, adição de uma técnica palpável para **NIDS!**, contribuindo para redução de danos a quem depende do bom funcionamento dos sistemas em redes **IoT!**.

Para o problema **NIDS!** em redes **IoT!** (ou detecção de comportamentos anômalos em fluxos de dados de sensores **IoT!** em geral) com base em **ND!** paralela e distribuída em névoa, o equilíbrio dos atributos de escalabilidade e qualidade de classificação ainda não foi alcançado.

Como trabalhos futuros, espera-se que esta implementação e análise sirva de inspiração para busca de algoritmos e sistemas que maximizam o aproveitamento do *hardware* e energia dedicados a esta tarefa. Em termos mais práticos, sugere-se a utilização do algoritmo *CluStream* (**??**) ou seus derivados e a ideia de um Modelo de tamanho fixo, maximizando o aproveitamento de memória e a acurácia com um grande conjunto de *micro-clusters*. Seguindo esta sugestão, estes *micro-clusters* podem ser definidos como polígonos para melhor cobertura do espaço, o Modelo pode ser checado quanto à sobreposição evitando ambiguidade e o Modelo organizado em índices para que a busca espacial seja rápida e eficiente. Acredita-se que estas características junto com as propriedades de paralelismo e distribuição em névoa podem levar a resultados brilhantes.