Distributed Novelty Detection at the Edge for IoT Network Security *

Luís Puhl[1], Guilherme Weigert Cassales[1[0000-0003-4029-2047]], Helio Crestana Guardia[1[0000-0001-5010-747X]], and Hermes Senger[1[0000-0003-1273-9809]]

Universidade Federal de São Carlos, Brasil
https://www2.ufscar.br/

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 from network administrators and the need for strategies to detect and react to those threats. For a network security system to operate in the context of edge and IoT, it has to comply with processing, storage and energy requirements alongside traditional requirements for stream and network analysis like accuracy and scalability. Using a previously defined architecture (IDSA-IoT), we address the construction of a distributed Novelty Detection System with the Data Stream Novelty Detection (DSND) algorithm MINAS and MPI library and we evaluate our proposal in a Network Introduction Detection (NIDS) role. 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. Results show a negligible loss of accuracy (hits) between original, serial and distributed executions while using less time than previous implementation, however efficient distribution was not achieved as the observed time as we added nodes remained constant.

Keywords: novelty detection · intrusion detection · data streams · distributed system · edge computing · internet of things

1 Introduction

The Internet of Things (IoT) brings together a wide variety of devices, including mobile, wearable, consumer electronics, automotive, and sensors of various types. Such devices can either be accessed by users through the Internet or connect to other devices, servers, and applications with little human intervention or supervision [14,1,8,12]. Security and privacy is a major concern in the IoT, specially regarding devices having access to user personal data like location, health, and many other sensitive data [11]. Furthermore, if compromised such devices can also be used to attack other devices and systems, steal information, cause immediate physical damage or perform various other malicious acts [9]. As an additional concerns, IoT devices likely have long lifespan, less frequent software patches, growing diversity of technologies combined with lack of control over the software and hardware of such devices by the host organization (where the device is deployed) considerably increases the attack surface.

Because most IoT devices have limited resources (i.e., battery, processing, memory, and bandwidth), configurable and expensive algorithm-based security techniques may not fit [16]. Machine Learning (ML) techniques have been studied for years to detect attacks from known patterns or to discover new attacks at an early stage [2,10]. A recent survey [14] shows that ML based methods are a promising alternative which can provide potential security tools for the IoT network making them more reliable and accessible than before.

Despite the promising use of ML to secure IoT systems, studies in the literature [2,10,14] are limited to traditional ML methods that use static models of traffic behavior. Most existing ML solutions for network-based intrusion detection cannot maintain their reliability over time when facing evolving attacks [15]. Unlike traditional methods, stream mining algorithms can be applied to intrusion detection with several advantages, such as: (*i*) processing traffic data with a single read; (*ii*) working with limited memory (allowing the implementation in small devices commonly employed in edge services); (*iii*) producing real-time response; and (*iv*) detecting novelty and changes in concepts already learned.

Adicionar referencias e relacionados. Argumentar que além de NIDS, DSND para IoT tem outras aplicações

Given the recent use of Data Stream Novelty Detection (DSND) in network data streams, this paper shows the effects of adapting these mechanisms to edge services for use in IoT environments. Our proposal, *MFOG*, instantiated and experimentally validated the IDSA-IoT architecture [3] employed the DSND algorithm MINAS [5,7] (as it was already tested in a similar IoT scenario) on a distributed system composed of small devices with limited resources on the edge of the network. We evaluated through experimental methodology, how the distribution of MINAS affects the capability to detect changes (novelty) in traffic patterns, and the impact on the computational efficiency. Finally, some distribution strategies and policies for the data stream novelty detection system are discussed.

This paper is organized as follows: Section 2 reviews the chosen DSND algorithm MINAS. A distributed extension of MINAS, including its implementation and evaluation are presented in Section 3 and in Section 4 we show how we evaluated *MFOG* and the discuss results we found. Finally, Section 5 summarizes the main findings and presents possible future work.

2 MINAS

MINAS [5,7] is an offline-online DSND algorithm, meaning it has two distinct phases. The first phase (offline) creates a initial model set with several clus-

* The authors would like to thank Brazilian funding agencies FAPESP and CNPq for the financial support.

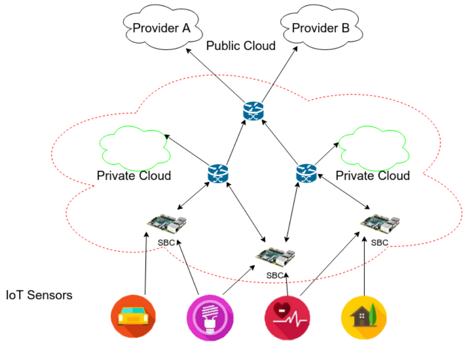


Fig. 1: IDSA-IoT [3] physical architecture and deployment scenario overview.

ters based on a clustering algorithm with a training set. Each cluster can be associated with only one class of the problem, but each class can have many clusters.

The online phase is where the model performs three tasks in (near) real-time. We describe the online phase in more detail since we focus on its tasks. In summary, the online phase of MINAS executes classification, novelty detection, and model update tasks until the possible end of stream, as shown in Algorithm 19. MINAS tries to classify each incoming unlabeled instance according to the current decision model. Instances not explained by the current model receive an *unknown* label and are stored in the unknowns-buffer. When the unknowns-buffer reaches a parametric size, MINAS executes the Novelty Detection function. Beyond the Novelty Detection, MINAS also cleans the unknowns-buffer removing old instances as they represent noise or outliers and, has a mechanism to forget clusters that became obsolete and unrepresentative of the current data stream distribution [7].

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

3 Proposal

In this work we investigate application of IDSA-IoT architecture [3] with DSND techniques by use of MINAS algorithm [7], by implementation and evaluation of a parallel and distributed descendant of those two works we named *MFOG*. However, given the distributed nature and the typical use of small computing devices in IoT scenarios, new constraints apply: (*i*) the classification phase of the algorithm must occur in parallel at different nodes; (*ii*) the novelty detection phase, which provides the model evolution, must also be asynchronous; (*iii*) the

```
Input: ModelSet, inputStream
Output: outputStream
Parameters: cleaningWindow, noveltyDetectionTrigger
Function MinasOnline(ModelSet, inputStream):
  UnkownSet ← ∅, ModelSleepSet ← ∅ ;
  lastCleanup ← 0 , noveltyIndex ← 0;
  foreach sample_i ∈ inputStream do
    nearest ← nearestCluster (sample, ModelSet);
    if nearest.distance < nearest.cluster.radius then
      sample.label ← nearest.cluster.label;
      nearest.cluster.lastUsed ← i ;
    else
      sample.label ← unknown;
      UnkownSet ← UnkownSet ∪ sample;
      if | UnkownSet | ≥ noveltyDetectionTrigger then
        novelties ← NoveltyDetection (ModelSet ∪ ModelSleepSet,
          *UnkownSet);
        ModelSet ← ModelSet ∪ novelties;
      if i > ( lastCleanup + cleaningWindow ) then
        ModelSet ← moveToSleep (ModelSet, *ModelSleepSet, lastCleanup);
        UnkownSet ← removeOldSamples (UnkownSet, lastCleanup);
        lastCleanup ← i;
    outputStream.append(sample);
```

Algorithm 1: Our interpretation of MINAS [7]

algorithm complexity (time and space) must allow it to be processed by modest computing devices.

introdução? NIDS monitor the packet network traffic, aggregate into flow descriptors and analyze to identify any intrusion or misbehavior. However, this problem requires both fast and accurate response [4]: fast response is needed to have a proper reaction before harm can be cast to the network and to cope with the traffic without imposing loss or delay in the NIDS or observed network; accurate response is required as to not misidentify harmless with harmful and vice-versa, especially the case of false positive that leads to false alarms. To achieve those goals we leverage fog computing.

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

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

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

Algorithm 2: MINAS [7] Novelty Detection task.

To have a better overview of our proposal and how it integrate in existing IoT environments, Figure 1 depicts such scenario showing from bottom to top: IoT devices directly connected to a (local) gateway network; this gateway network could be as simple as a single router to the internet or be more complex by connecting to private clouds or containing more devices providing fog computing capabilities; lastly, available over the internet, traditional public cloud provides inexpensive computing and storage on demand. In this scenario the further apart resources are, more network resources need to be dedicated and, as any networked system, more latency between action and reaction is observed.

The overall *MFOG* architecture can be cut down to individual modules: three main functional modules being Classification and Novelty Detection handling MINAS main tasks.

Classification Module houses the homonymous task of MINAS Online phase and is the focal point for parallelism in our proposal, being replicated in the fog on each local network containing a cluster with one or more nodes and each node multiple processes (limited to the individual CPU core count).

Novelty Detection Module can be also replicated, one instance per local network and one global instance, also handling the homonymous task of MINAS Online phase. This modules takes as input all samples labeled with *unknown*, stores them in a internal *unknown* buffer and when this buffer is full, triggers MINAS Novelty Detection task (clustering followed by validation).

3.1 Polices

The distribution of steps and tasks in various modules opens data distribution and its impacts to discussion. The decisions following these discussions can be organized in several policies, some of them were recurring during our implementation discussions and are:

```
Parameters: cleaningWindow, noveltyDetectionTrigger, mpiClusterSize as
    mpiSize, mpiNodeRank as mpiRank
Input: ModelSet, Sample Stream
Output: Classified Stream as out
Function Mfog(ModelStream, InputStream):
    ModelSet = ∅;
    ModelSetLock = new Lock ();
    if mpiRank == 0 then root
        new Thread (Detector, [ModelSet]);
        Sampler (InputStream);
    else leaf
        new Thread (modelReceiver, [ModelSet]);
        Classifier (ModelSet);
```

Algorithm 3: MFOG: main MPI entry-point.

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

```
Function Classifier(mp):
    while True do
        sampe = receive (SampleType, root);
        if sample == EndOfStream then break;
        sample.label = unknown;
        with readLock (ModelSetLock)
            (distance, cluster) = nearestCluster (sample, ModelSet);
        if distance < cluster.radius then
            sample.label = cluster.label;
        send (root, SampleType, sample);
Function modelReceiver(mp, ModelSet):
    while True do
        cl = receive (ClusterType, root);
        if cl == EndOfStream then break;
        with writeLock(ModelSetLock)
            ModelSet = ModelSet ∪ cl;
```

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

3.2 Implementation

The original MINAS algorithm has a companion unpublished implementation (*Ref*) written in Java using MOA library base algorithms such as K-means and CluStream, however our implementation only used K-means. Another difference between *Ref* and *MFOG* is cluster radius calculation from the distances of elements forming the cluster and the cluster's center. *Ref* uses the maximum distance while *MFOG* uses the standard deviation of all distances as described in [7].

The stream format for input and output also of note. Input information needed is the samples value ($\vec{v}$), this $\vec{v}$ is a number sequence of length d (dimension). In addition to $\vec{v}$, for evaluation and training purposes the class identifier as single character, and unique item identifier (*uid*) can be provided or be the sample index in the stream.

For output information and format the decision isn't so clear as we can't predict future system integrations needs like only novelty alarms or every samples original $\vec{v}$ with assigned label so, we have a compromise and put only enough information for the Sink Module (where the full information from the testing file or stream can be accessed) meaning the format can be defined as a tuple containing *uid* and assigned label.

For *MFOG*, the Message Passing Interface (MPI, from *Open MPI 4.0.4*) library was used. In MPI programming, multiple processes of the same program are created by the run-time and each process instance receives a rank parameter, for *MFOG* this parameters indicate if the process is root, rank 0, or leaf otherwise. Beyond this division, each process also operates two threads, for the root there is a sampler and detector threads, for the leafs each has a model receiver thread and multiple classifier threads. The overall sequence of interactions is shown in Figure 2.

```
Function Sampler(mp, p, InputStream):
    dest = 1;
    foreach sample from InputStream do
        if typeOf (sample) is Cluster then
            broadcast (ClusterType, sample, root);
            with writeLock (ModelSetLock)
                ModelSet = ModelSet ∪ sample;
            continue;
        send (dest, SampleType, sample);
        dest = dest +1;
        if dest > mpiSize then dest = 1;
Function Detector(mp, ModelSet):
    lastCleanup = while True do
        sampe = receive (SampleType, any);
        if sample == EndOfStream then break;
        out ← sample;
        if sample.label == unknown then
            UnkownSet = UnkownSet ∪ sample;
            if sizeOf (UnkownSet) ≥ NDT then
                novelties = NoveltyDetection (p, ModelSet, *UnkownSet);
                with writeLock (ModelSetLock)
                    ModelSet = ModelSet ∪ novelties;
                foreach cl in novelties do
                    broadcast (ClusterType, cl, root);
            if now () > lastCleanup + CW then
                ModelSet = handleModelSleep (ModelSet, ModelSleepSet);
                UnkownSet = removeOldSamples (UnkownSet, lastCleanup);
```

Algorithm 5: MFOG Root Tasks: Sampler and Detector.

4 Experiments and Results

For the experimental setup we dedicated three Raspberry Pi 3 model B single board computers connected via Ethernet Switch forming a simple cluster. This cluster stored all source code, binaries (compiled and linked in place) and datasets, being accessed via our laboratory network over Secure Shell (SSH). All experiments were executed in this cluster for isolation of otherwise unforeseen variations.

The dataset used is the December 2015 segment of Kyoto 2006+ Dataset[1] (Traffic Data from Kyoto University's Honeypots) [13]. This segment was filtered (from 7 865 245 instances) to only examples associated to known attack types identified by existing NIDS, and attack types with more than 10 000 instances for significance as previously done by [3]. The remaining instances then were transformed by normalization so each feature value space (e.g. IP Address, Duration, Service) is translated to the Real interval $[0, 1]$.

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

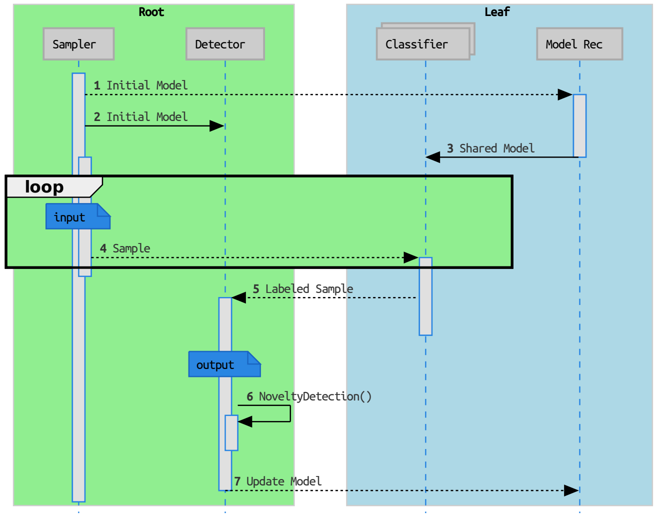


Fig. 2: *MFOG* life line overview.

The resulting derived dataset is then stored in two sets, training set and test set, using the holdout technique. However, for the training set we filter in only normal class resulting in 72 000 instances. For the test set we use 653 457 instances with 206 278 instances with "*N*" (normal) class and 447 179 instances with "*A*" (attack) class. Note that this choice results in a possible over-fitting for the normal class and, under-fitting for attack class as the system first needs to detect a novel class and then add it to the model.

4.1 Mesures and Visualizations

There are two broad evaluation mesures for each experiment: a time mesure extracted by using *GNU Time 1.9* measuring of the full program execution and, a set of qualitative mesures extracted by a python program.

Our evaluation script was build following reference techniques like multi-class confusion matrix with label-class association [7] to extract classification quality mesures. This program takes two inputs, the test dataset and the captured output stream, and outputs the confusion matrix, label-class association, final quality summary with: Hits (true positive), Misses (Err), Unknowns (UnkR); and stream visualization chart with per example instance summary with novelty label markers. For clarity, it is necessary to detail how to interpret and compare each mesure, as for some it is trivial but others are not so much.

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

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

Added to the original confusion matrix C are the rows *Assigned* and *Hits*. *Assigned* row represents which original class c (or if *unknown*, *"-"*) the label l is assigned to, this is computed by using the original class if $c = l$ or by associated novelty label to original class as described in [6] section 4.1 (class from where the most samples came from). *Hits* row shows the true positive count for each label, computed by coping the value of the cell $M_{c,l}$ where the label is the same and the class c is the value in the above *Assigned* row. The *Hits* row is also used to compute the overall true positive. One complete matrix is shown in Tab. 1a.

Table 1: Confusion Matrixs and Qualitative Mesures

(a) Reference implementation

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

(b) Serial implementation

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

(c) Parallel single-node

Labels	-	N	0	1	2	3	4
Classes							
A	12282	433797	147	952	0	0	1
N	3088	203019	40	99	27	5	0
Assigned	-	N	A	A	N	N	A
Hits	0	203019	147	952	27	5	1

(d) Parallel multi-node

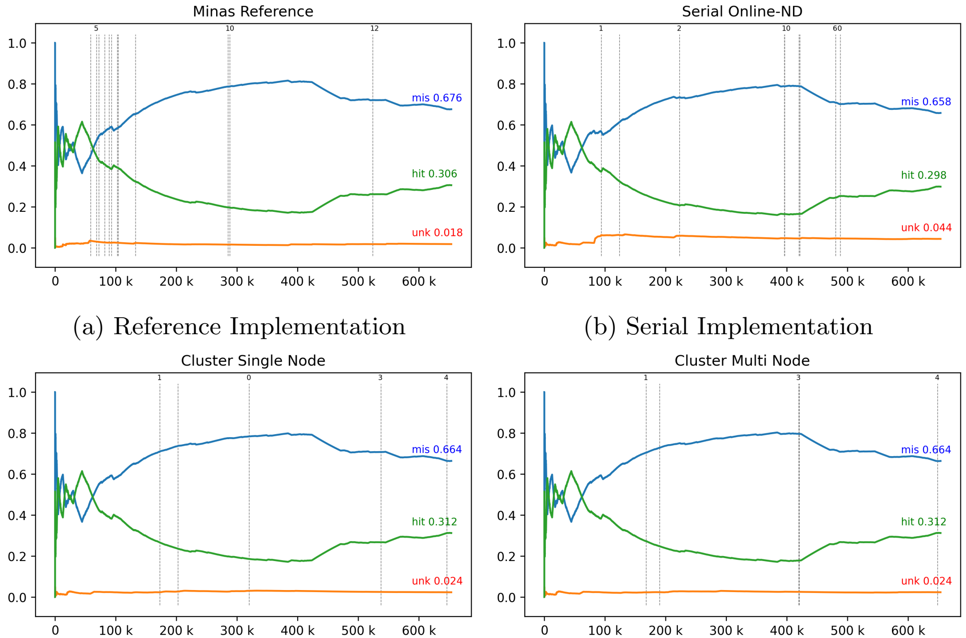
Labels	-	N	0	1	2	3	4
Classes							
A	12378	433631	117	886	0	162	5
N	3121	202916	40	96	105	0	0
Assigned	-	N	A	A	N	A	A
Hits	0	202916	117	886	105	162	5

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

Also in the mesure summary table, *Time*, *System* and *Elapsed* mesures represented in seconds, are extracted from *GNU Time*. *Time* is the amount of CPU seconds expended in user-mode (indicates time used doing CPU intensive computing, e.g. math); *System* is the amount of CPU seconds expended in kernel-mode (for our case it indicates time doing input or output); *Elapsed* is the real-world (wall clock) elapsed time and indicates how long another system or person had to wait for the result. To compare the time mesure is simple, the lower time taken, the better. Our four main experiments are shown in Tab. 2.

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

Adding to the stream visualization chart, novelty label markers are represented as vertical lines indicating *when* in the captured output stream a new label first appeared. Some of the novelty label markers include the label itself ($l \in \mathbb{N}$) for reference (showing every label would turn this feature unreadable due to overlapping). Figure 3 shows complete stream visualization charts.



(a) Reference Implementation

(b) Serial Implementation

(c) Parallel single-node

(d) Parallel multi-node

Fig. 3: Stream hits and novelties visualization

4.2 Results Discussion

Four main experiments need detailed discussion: (a) reference implementation of Minas (*Ref*) [7]; (b) new implementation in serial mode; (c) new implementation in single-node, multi-task mode and (d) new implementation in multi-node, multi-task mode. Each experiment uses the adequate binary executable, initial model (or training set for the reference implementation) and test set to compute a resulting output stream which is stored for qualitative evaluation. The summary of all four experiments is shown in Table 2.

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

As stated, to validate our implementation we compare it to *Ref* (the original MINAS companion implementation), so we extracted the same mesures using same process for both a and b, they can be viewed on Tables 1a, 1b and for ease of comparison on Table 2 the summary can be compared side by side.

In general the observed classification quality mesures are very similar, they diverge slightly where a has more *Hits* and *Misses* whereas b shifted those to *Unknowns*. This phenomenon was watched very closely during development and we found that small changes to MINAS parameters, MINAS internals like K-means ordering, cluster edge inclusion and cluster radius formula as stated in Subsection 3.2.

As for the time mesures on Table 2 our implementation used less time to analyze the test data set. This is mostly due to the stop condition on the internal K-means algorithm, while *Ref* uses a fixed iteration limit of 100, our implementations adds the “no improvement” check and stops earlier on most cases and this in turn reduces time taken on the *NoveltyDetection* function. There are also small optimizations on the *nearestCluster* function (minimal distance from sample to cluster center in the set) affecting the *classifier* task and *NoveltyDetection* function. Also note that *Ref* time in a includes the Offline phase while our implementation runs it once and reuses the initial model for b, c and d, in the table the offline time is on its separated column.

As for the effects of running a MPI cluster with our implementation we observe an increase of time when e go from 2 to 4 instances in a single node (b and c respectively), hinting that our choice of load distribution is not as effective as we expected. Further experiments were conducted with instances varying from 1 (serial) to 12 (3 nodes with 4 CPUs each) and true positive rate (*Hits*) and elapsed time had no major difference. More detailed time measurements can be seen on Figure 4, where we observe near constant time for *elapsed* (near 100s), the *system* increases gradually while *user* decreases at the same rate.

Table 2: Collected Mesures Summary.

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

We interpret this behavior as a display of potential for gains using a better load balancing than our choice of round-robin such as micro-batching for better compute-to-communication ratio (CCR). In general Figure 4 shows no speedup but also no penalty for scaling to more than 4 instances.

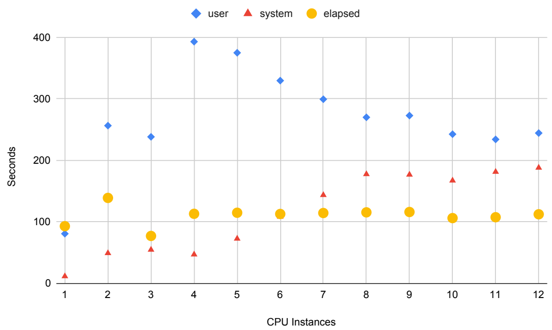


Fig. 4: Time measurements per added instance

Nevertheless, we can also show the effects of delay in the Classify, Novelty Detection, Model Update and Classify feedback loop. Comparing *b* and *c* we observe a reduction in Novelty labels on the Confusion Matrix (tabs. 1b and 1c) from 10 to 4. The same effect is observed on the stream visualization (figs. 3b and 3c) where our serial implementation has less novelty markers and they appear latter, but the mesures keeps the same "shape". Comparing *c* and *d* the difference is even smaller, (figs. 3b and 3c) as they both suffer the expected delay in the feedback loop.

5 Conclusion

Novelty Detection in Data Streams (DSND) is a useful tool for IoT Network Intrusion Detection (NIDS) or other related application of DSND using continuous network or system behavior monitoring and analysis. However, in the IoT context, it is expected that small edge devices perform such maintenance tasks. In that small computing on edge scenario, we propose *MFOG*: a distributed DSND implementation based on the DSND algorithm MINAS, and, evaluated with a NIDS task with appropriate dataset. The main goal this work is to observe the effects of our approach to a previously serial only algorithm, specially in regards to time and quality metrics.

While there is some impact on the predictive metrics this is not reflected on overall classification quality metrics indicating that distribution of MINAS shows a negligible loss of accuracy. In regards of time and scale, our distributed executions used less time than previous implementation but efficient distribution was not achieved as the observed time as we added nodes remained constant.

Our treatment involved reworking the algorithm and implementation to be distributed and to minimize the memory usage as to fit in smaller devices. Other algorithms still need a similar treatment and, more importantly, other distribution strategies should be considered.

Acknowledgment

The authors thank CNPq (Contract 167345/2018-4). Hermes Senger also thanks CNPq (Contract 305032/2015-1) and FAPESP (Contract 2018/00452-2, and Contract 2015/24461-2) for their support.

References

1. Abane, A., Muhlethaler, P., Bouzefrane, S., Battou, A.: Modeling and improving named data networking over ieee 802.15.4. In: 2019 8th International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN). pp. 1–6 (2019). https://doi.org/10.23919/PEMWN47208.2019.8986906
2. Buczak, A.L., Guven, E.: A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Communications Surveys & Tutorials **18**(2), 1153–1176 (2016)
3. Cassales, G.W., Senger, H., DE FARIA, E.R., Bifet, A.: Idsa-iot: An intrusion detection system architecture for iot networks. In: 2019 IEEE Symposium on Computers and Communications (ISCC). pp. 1–7 (June 2019). https://doi.org/10.1109/ISCC47284.2019.8969609, `https://ieeexplore.ieee.org/document/8969609/`
4. da Costa, K.A., Papa, J.P., Lisboa, C.O., Munoz, R., de Albuquerque, V.H.C.: Internet of things: A survey on machine learning-based intrusion detection approaches. Computer Networks **151**, 147–157 (2019). https://doi.org/10.1016/j.comnet.2019.01.023

5. Faria, E.R., Gama, J.a., Carvalho, A.C.P.L.F.: Novelty detection algorithm for data streams multi-class problems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. p. 795–800. SAC '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2480362.2480515, `https://doi.org/10.1145/2480362.2480515`
6. de Faria, E.R., Gonçaalves, I.R., Gama, J., Carvalho, A.C.P.d.L.F.: Evaluation of multiclass novelty detection algorithms for data streams. IEEE Transactions on Knowledge and Data Engineering **27**(11), 2961–2973 (nov 2015). https://doi.org/10.1109/TKDE.2015.2441713, `http://ieeexplore.ieee.org/document/7118190/`
7. Faria, E.R.d., Ponce de Leon Ferreira Carvalho, A.C., Gama, J.: Minas: multiclass learning algorithm for novelty detection in data streams. Data Mining and Knowledge Discovery **30**(3), 640–680 (May 2015). https://doi.org/10.1007/s10618-015-0433-y, `https://doi.org/10.1007/s10618-015-0433-y`
8. HaddadPajouh, H., Dehghantanha, A., Parizi, R.M., Aledhari, M., Karimipour, H.: A survey on internet of things security: Requirements, challenges, and solutions. Internet of Things p. 100129 (2019)
9. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: DDoS in the IoT: Mirai and Other Botnets. Computer **50**(7), 80–84 (2017). https://doi.org/10.1109/MC.2017.201, `http://ieeexplore.ieee.org/document/7971869/`
10. Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Computing Surveys (CSUR) **46**(4), 55 (2014)
11. Sengupta, J., Ruj, S., Bit, S.D.: A comprehensive survey on attacks, security issues and blockchain solutions for iot and iiot. Journal of Network and Computer Applications **149**, 102481 (2020)
12. Shanbhag, R., Shankarmani, R.: Architecture for internet of things to minimize human intervention. 2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015 pp. 2348–2353 (2015). https://doi.org/10.1109/ICACCI.2015.7275969
13. Song, J., Takakura, H., Okabe, Y., Eto, M., Inoue, D., Nakao, K.: Statistical analysis of honeypot data and building of kyoto 2006+ dataset for nids evaluation. Proceedings of the 1st Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS 2011 pp. 29–36 (2011). https://doi.org/10.1145/1978672.1978676
14. Tahsien, S.M., Karimipour, H., Spachos, P.: Machine learning based solutions for security of internet of things (iot): A survey. Journal of Network and Computer Applications **161**(November 2019) (2020). https://doi.org/10.1016/j.jnca.2020.102630
15. Viegas, E., Santin, A., Bessani, A., Neves, N.: Bigflow: Real-time and reliable anomaly-based intrusion detection for high-speed networks. Future Generation Computer Systems **93**, 473 – 485 (2019). https://doi.org/https://doi.org/10.1016/j.future.2018.09.051, `http://www.sciencedirect.com/science/article/pii/S0167739X18307635`
16. Zhou, J., Cao, Z., Dong, X., Vasilakos, A.V.: Security and privacy for cloud-based iot: Challenges. IEEE Communications Magazine **55**(1), 26–33 (2017). https://doi.org/10.1109/MCOM.2017.1600363CM