FixMe: A Self-organizing Isolated Anomaly Detection Architecture for Large Scale Distributed Systems

Emmanuelle Anceaume[1], Erwan Le Merrer[2], Romaric Ludinard[3], Bruno Sericola[3], and Gilles Straub[2]

[1] IRISA / CNRS (France), firstname.name@irisa.fr
[2] Technicolor Rennes (France), firstname.name@technicolor.com
[3] Inria Rennes - Bretagne Atlantique (France), firstname.name@inria.fr

Abstract. Monitoring a system is the ability of collecting and analyzing relevant information provided by the monitored devices so as to be continuously aware of the system state. However, the ever growing complexity and scale of systems makes both real time monitoring and fault detection a quite tedious task. Thus the usually adopted option is to focus solely on a subset of information states, so as to provide coarse-grained indicators. As a consequence, detecting isolated failures or anomalies is a quite challenging issue. In this work, we propose to address this issue by pushing the monitoring task at the edge of the network. We present a peer-to-peer based architecture, which enables nodes to adaptively and efficiently self-organize according to their "health" indicators. By exploiting both temporal and spatial correlations that exist between a device and its vicinity, our approach guarantees that only isolated anomalies (an anomaly is isolated if it impacts solely a monitored device) are reported on the fly to the network operator. We show that the end-to-end detection process, *i.e.*, from the local detection to the management operator reporting, requires a logarithmic number of messages in the size of the network.

1 Introduction

The number of IP-enabled devices keeps on growing in a steady manner, often reaching millions of units managed by a single operator. If those devices are able to provide a service to the user in their intended running state, deviations in behavior or hardware/software problems are generally detected offline by human intervention. The technical barrier for efficient online monitoring and analysis is the size of the devices set to operate, together with the huge amount of parameters and states to consider. Network operators deploy helpdesk in order to support their customers when they are facing problems. In the last years the cable and telecom industry have developed different remote management standards [1] to better support the helpdesk operator via dedicated protocols and tools. As a consequence, the helpdesk operation represents an important part of the overall operating cost of a network provider. Reducing

the number of calls as well as their duration is an important key for every network operator to sustain profitability and reduce the total cost of ownership. Nevertheless both telecom and cable industries came up with client-server architectures where a single server (or a farm of servers) is in charge of managing a set of devices. Such architectures are typically used for management tasks (*e.g.*, service provisioning, device firmware upgrading) rather than for real time monitoring activities, essentially because of scaling issues. Indeed, the massive scale we are considering calls for efficient monitoring algorithms. A first option is to gather all the devices logs in a single place, and to analyze collected data using for instance the MapReduce paradigm [2] to detect the causes of the anomalies. This nevertheless implies a significant detection latency and processing cost at the cloud architecture level.

The second option is to push monitoring procedures on devices. Actually, standardized procedures exist at devices level to autonomously trigger asynchronous alarms in presence of anomalies. However, these procedures are never used for practical reasons. Indeed if the cause of the anomaly lies in the network itself (*e.g.*, at routers, links or data center outages) this may impact a very large number of devices, and thus letting thousands of impacted devices reporting the problem to the helpdesk operator may quickly become a disaster due to the volume of generated messages. On the other hand, it is of utmost importance to minimize the overall network footprint by giving each device the capability to self distinguish network-based anomalies from *isolated* ones – anomalies that only impact the device itself – so that only isolated anomalies are reported on the fly to the helpdesk. This is the problem that we address in this paper. Specifically, we propose a novel distributed monitoring tool, called FixMe, that enjoys the following properties.

- FixMe is self-managing: all the monitored devices self-organize according to their "health" indicators so that they can detect any correlation between their state and the one of their neighbors,
- FixMe is dynamic: *(i)* monitored devices may join the system or may be removed from it at any time, and *(ii)* there is no assumption regarding the QoS repartition of the monitored nodes (*i.e.*, we do not assume that the repartition is uniform),
- FixMe does not rely on any complex bootstrap procedure. In contrast to most of the monitoring tools, devices do not need to be prearranged into a predefined number of clusters (as required for in instance in k-means based solutions),
- FixMe is scalable: the end-to-end detection process, *i.e.* from the local detection to the management operator reporting, requires a logarithmic number of messages.

The remaining of the paper is organized as follows. Section 2 provides an overview of existing monitoring approaches. Section 3 presents the model of the system, and defines the addressed problem. Section 4 describes the FixMe overlay and its associated operations, while in Section 5 its efficiency is analyzed. Section 6 describes the algorithm that solves the addressed problem. Section 7 concludes and presents future works.

2 Related work

This Section provides an overview of the existing techniques used in large scale systems to continuously and automatically monitor time-varying metrics. The authors in [3] exploit temporal and spatial correlations [4–6] among groups of monitored nodes to decrease monitoring communication costs, *i.e.*, the cost incurred by the periodic reporting of the updated metrics values from the monitored nodes to the management node. The idea is to prevent any reporting message from occurring when such a reporting would contain metrics values that could be directly inferred by the management node. This is achieved by giving each monitored node the capability to locally detect whether the current values of its monitored metrics are in accordance with predicted ones (through Kalman filters tools [7] installed at both monitored nodes and the management node), and by gathering nodes into clusters (such that, for each monitored metric, a set of clusters group together nodes that share correlated values of the considered metric according to the Pearson correlation coefficient). At clusters level, an elected leader is in charge of communicating with the management system when the current metric values of its group members differ from each others. Although close to our objectives, the main drawback of this solution lies on the centralized clustering process. All the nodes of the system are continuously organized into clusters computed through the k-means algorithm exclusively run by the management node, which is a clear impediment to the scalability of their approach. Other works aim at minimizing the processing cost for continuous monitoring [8–10] in the light of the theoretical results of [11], however similarly to [3], all these approaches suffer from a centralized handling of the clustering process.
In contrast, our objective is a fine-grain detection tool capable of accurately and efficiently detecting isolated events. As will be described in the remaining of the paper, we combine clustering and structured peer-to-peer architectures to tend toward this objective.

3 Model of the System

We consider a set of N nodes that communicate among each other through the standard synchronous message-passing model. Each node in the system is assigned a unique random identifier derived from a standard hash function (*e.g.* MD5, SHA-1). Each node has access to D services numbered $1, \ldots, D$. At any time t, the QoS of each service is locally measured with an end-to-end performance measurement function

$$\begin{aligned} Q_i : \{1, \ldots, N\} \times \mathbb{N} &\longrightarrow [a_i, b_i] \\ (p, t) &\longmapsto \text{Quality of service } i \text{ at node } p \text{ at time } t \end{aligned}$$

Without loss of generality we suppose that the QoS range $[a_i, b_i]$ of service i is equal to $[0, 1]$. We define the *position* of a node p at time t by the vector $Q(p, t)$ defined as

$$Q(p, t) = (Q_1(p, t), \ldots, Q_D(p, t)). \tag{1}$$

For each monitored service $i = 1, \ldots, D$, we split interval $[0, 1]$ into n_i disjoint intervals $[x_i^{(j-1)}, x_i^{(j)})$, $1 \leq j \leq n_i$, with $x_i^{(0)} = 0$ and $x_i^{(n_i)} = 1$, the last interval being closed. Integer n_i is a parameter of the system. These n_i intervals can be thought as n_i QoS classes of service i. For instance, one can consider a division of $[0, 1]$ for service i such that $|x_i^{(j)} - x_i^{(j-1)}| \geq |x_i^{(j+1)} - x_i^{(j)}|$. Such a division could be used to reflect the increasing sensitivity of users regarding QoS variations. A user is more sensitive to a very small variation of a high QoS than to a large variation of a low QoS. Without loss of generality, we suppose a regular division into identical length intervals and we define $\rho_i = |x_i^{(j)} - x_i^{(j-1)}| = 1/n_i$. In the following these intervals are named *buckets* (a more precise definition is given in Section 4).

In addition to the functions $Q_1, \ldots, Q_D$, each node has access to D anomaly detection functions $A_1, \ldots, A_D$. At each time t, each function A_i is fed with the sequence of the $\ell_i \geq 1$ last QoS values $Q_i(p, t - \ell_i + 1), \ldots, Q_i(p, t)$ and provides some meaningful prediction of what should be the next QoS value. Note that ℓ_i is a parameter of A_i. These functions are implemented to cope with the specific variations of their input values, and thus different kinds of anomaly detection functions exist, ranging from a simple threshold based functions, to more sophisticated ones like the Holt-Winters forecasting or Cusum method. In this paper, we suppose that the output of these anomaly detections are boolean. At time t, $A_i(p, t) =$ `true` if the sequence $Q_i(p, t - \ell_i + 1), \ldots, Q_i(p, t)$ is considered as an anomaly, it is `false` otherwise. Implementation of both Q_i and A_i functions are out of the scope of the paper.

Finally, suppose that a node locally detects an anomaly whose origin comes from a network/service dysfunction or failure. Then this anomaly will have an impact on the QoS of other nodes, and thus these nodes will locally detect it. On the other hand, we suppose that if a node locally detects an anomaly whose origin is local (hardware or software), then this anomaly will only impact its QoS, and thus no other nodes will be impacted by this specific anomaly.

Prior to defining the addressed problem, let us consider the following simple scenario presented in Fig. 1. The QoS of a single service monitored by two nodes a and b is represented by interval $[0, 1]$. At time t the quality positions $Q(a, t)$ and $Q(b, t)$ of both nodes lie in bucket j, while at time $t + 1$, at least one of the two nodes experience a QoS change. Five situations can be observed. In situations (1), (2) and (4) node a is the only node that observes a QoS change. In situation (1), this change does not push a position outside bucket j, while in situation (2) and (4) it does. However in both situations (1) and (2), the anomaly detection function $A_1(a, t + 1) =$ `false`, thus a does not consider this move as an anomaly, therefore does not do any more investigation. In the other hand, in situation (4), $A_1(a, t + 1) =$ `true`, and thus node a triggers a FixMe message. Now observe the two last situations (3) and (5). Both nodes observe a QoS change considered as an anomaly by their function A (*i.e.*, $A_1(a, t + 1) = A_1(b, t + 1) =$ `true`). However in situation (3) the QoS degradation is the same for both nodes ($Q(a, t+1)$ and $Q(b, t+1)$ lie in bucket k) and thus neither a nor b consider this anomaly as isolated, while in situation (5) $Q(a, t+1)$ and $Q(b, t+1)$ respectively lie in buckets

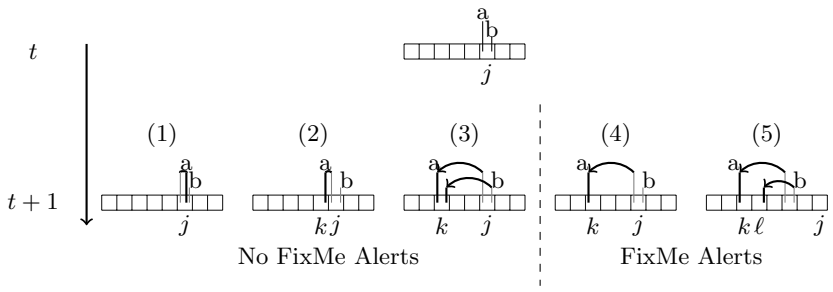

Fig. 1. Isolated anomaly detection of one monitored service. Node a triggers FixMe message in both cases (4) and (5), while node b triggers it only in case (5).

k and ℓ. Thus both nodes trigger a FixMe message. We now formally define the problem we address in this work.

Definition 1 (The Isolated Anomaly Detection Problem). *Let $\mathcal{S} = \{1, \ldots, N\}$ be the set of monitoring nodes, and an additional node named the management operator with which any of the N nodes communicate. Let $\mathcal{S}^t_{j,k} \subseteq \mathcal{S}$ be such that $\forall p \in \mathcal{S}^t_{j,k}$, p has moved from bucket j to bucket k from time $t-1$ to time t and there exists a service i such that $A_i(p,t) =$ `true`. Then at time $t+1$, an alert is raised at the management operator if and only if $|\mathcal{S}^t_{j,k}| \leq \tau$, with τ a parameter of the system. In Fig. 1, $\tau = 1$.*

4 FixMe Framework

4.1 Rationale

In this Section, we describe how we address the Isolated Anomaly Detection problem in a distributed system composed of N monitored nodes. FixMe framework orchestrates the monitored nodes into an overlay network, named in the following FixMe overlay. An overlay network is actually a virtual network built on top of the physical network within which nodes communicate among each other along the edges of the overlay by using the communication primitives provided by the underlying network (*e.g.* IP network service). The algorithms nodes use to choose their neighbors and to route their messages define the overlay topology. The topology of unstructured overlays conforms with random graphs (*i.e.*, relationship among nodes are mostly set according to a random process which reveals to be inefficient to find a particular node or set of nodes in the overlay). On the other hand, structured overlays build their topology according to structured graphs (*e.g.*, tree, torus, hypercube). Most of the structured overlays are based on Distributed Hash Tables (*e.g.*, [12, 13]).

The efficiency and scalability of all these proposed DHTs rely on the uniform distribution of the nodes in the identifiers space at the expense of breaking the application logic. This is why, for specific applications such as streaming applications, broadcast spanning trees structures, that support the application-level broadcast, have been proposed [14]. Our concern is to exploit the QoS relationship among monitored nodes, which make all the aforementioned solutions non adapted. As a consequence, we propose to organize nodes so that at any time t the neighbors of any node p are the nodes q whose QoS (*i.e.* $Q(q,t)$) are closer to the QoS of p (*i.e.* $Q(p,t)$). The description of such an organization is done in Section 4.2. From the application point of view, three operations are provided by the system: the `lookup`, the `join`, and `leave` operations that allow nodes to respectively find a position in the overlay, join the overlay or `leave` it. From the topological structure point of view, two operations are provided: the `split` and `merge` operations that guarantee the scalability of FixMe overlay when some regions of the overlay become too dense or too sparse. All these operations are described in Section 4.3. Finally, when too many monitored nodes share exactly the same QoS (or equivalently sit at the same position in the overlay), nodes within the bucket self-organize into an hypercube as described in Section 4.4.

4.2 Overview of FixMe Overlay

The FixMe overlay is a virtual multi-dimensional cartesian coordinate space on a multi-torus. The entire coordinate space is tessellated into a collection of *buckets*. A bucket is the cartesian product of D intervals of respective length $\rho_1, \ldots, \rho_D$ (*cf.* Fig. 2, where FixMe overlay is made of 16^2 buckets). When a node p `joins` FixMe at time t, p joins the bucket which corresponds to its quality position (or simply its position) $Q(p,t)$. When a bucket is populated by more than $S_{\min}$ nodes this bucket is called a *seed*. The entire coordinate space is dynamically partitioned into distinct zones, named *cells*, such that a cell contains at most one seed (*cf.* Fig. 2, where FixMe overlay on the left is made of four cells, and the one on the right is made of five cells). More formally,

Definition 2 (Cell). *A cell is defined as an hyper-rectangle of buckets, among which at most one is a seed. A cell is fully and uniquely characterized by a set of 2^D buckets called the corners of the cell, sorted using the lexicographic order.*

Figure 2 shows these different elements for $D = 2$ and $\rho_1 = \rho_2 = 1/16$. The buckets are elementary squares, the seeds are represented by the black squares, and the cells are depicted by the coloured rectangles. Note that neither cell 4 (on the figure on the left) nor cell 5 (on the figure on the right) have a seed. The reasons will be detailed in the following.

4.3 FixMe Operations

`Lookup` operation. We describe how a node locates the seed that is in charge of a given bucket b through the `lookup` operation. In FixMe,

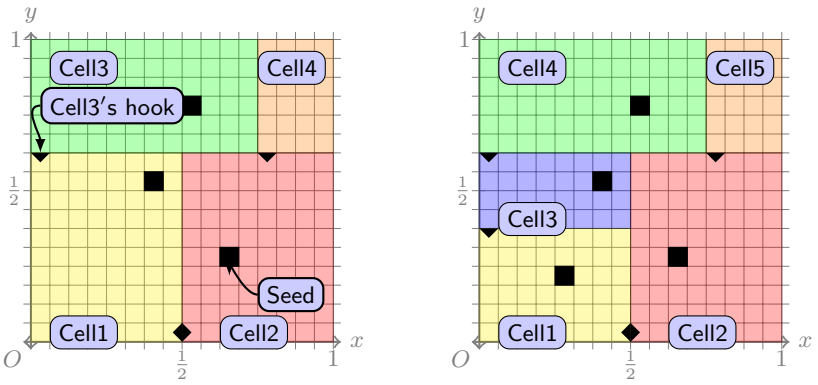

Fig. 2. FixMe overlay before (on the left) and after (on the right) a `split` operation

routing is exclusively handled by seeds. Each seed maintains a *routing table* that contains an entry for each of its $2D$ neighboring seeds in the coordinate space. An entry contains the IP address and the virtual coordinate of the seed. A `lookup` message contains the destination coordinates. Using the neighbor coordinate, a seed routes a `lookup` message toward its destination using a simple greedy forwarding to the neighbor seed that is closest to the destination address. CAN [12] uses this routing to cross its zones. However, as such the lookup operation needs to cross in average $\mathcal{O}(DN^{1/D})$ zones. We combine the multidimensional routing of CAN with Chord-fashioned long-range neighbors [13, 15] to improve the lookup operation cost. Specifically, in addition to its $2D$ neighboring seeds, each seed associates a location key to each neighbor seed of its routing table. Hence, if the seed coordinates are $(x_1, \ldots, x_d, \ldots, x_D)$, then the $+i$th (respectively the $-i$th) key location for the dth axis is defined by $(x_1, \ldots, x_{(d,+i)}, \ldots, x_D)$, where $x_{(d,+i)} = x_d + 2^i \rho_d$ (respectively $x_{(d,-i)} = x_d - 2^i \rho_d$). In addition, the distance between the seed and the location key is bounded by R_d where R_d is a system parameter corresponding to the absolute farthest location to be accessed in one hop in the dth axis. Each seed s also maintains a predecessors table that contains couples (s', l), where s' is a seed pointing on location l in s cell. The predecessors table is used when a `split` or `merge` operation are triggered to update the predecessors routing table.

`Join` *operation.* When some new node p wants to `join` the system at time t, it contacts some node q already in the system. This bootstrap node q sends a lookup request for the incoming node position $Q(p, t)$ to find the seed s responsible for the cell in which p must be inserted. Once p gets s address, it asks s to `join` the bucket that matches its position $Q(p, t)$. If that bucket is the seed s itself, then the procedure described in Section 4.4 is run. Otherwise, s updates its *cell routing table* by inserting p address and its position $Q(p, t)$. Similarly, p keeps a pointer to s (as described above, routing is handled by seeds, thus p only needs to point to s). Now, if the number of nodes that sit in p bucket exceeds $S_{\min}$

Algorithm 1: p.join(t,q=None)

```
1  begin
2      if q = None then
3          q ← getBootstrapNode() ;
4      end
5      seed ← q.lookup(Q(p,t));
6      if p ∈ seed then
7          seed.insert(p);
8      else
9          bucket ← seed.findBucket(p);
10         bucket.insert(p);
11         if | bucket |≥ S_min then
12             cells ← seed.split(bucket);
13         end
14     end
15 end
```

Algorithm 2: cell.merge(bucket)

```
   Data: bucket such that| bucket |< S_min
1  begin
2      seed ← cell.seed;
3      seed.addOrphanCell(bucket.cell);
4      seed.mergeSiblingsCells();
5      seed.notifyPredecessors(bucket.cell);
6  end
```

Algorithm 3: cell.split(bucket)

```
   Requires: | bucket |≥
             S_min ∧ ¬bucket.isSeed()
   Ensures : bucket.isSeed()
1  begin
2      matchingCell ← findCell(bucket);
3      if matchingCell ∈ orphanCells then
4          bucket.cell ← matchingCell;
5          orphanCells.remove(matchingCell);
6      else
7          matchingCell.split(bucket);
8      end
9      bucket.notifyPredecessors(matchingCell);
10     bucket.updateRoutingTable();
11     bucket.setSeed(True);
12 end
```

Algorithm 4: p.leave()

```
1  begin
2      bucket ← p.bucket;
3      isSeed ← bucket.isSeed();
4      bucket.removePeer(p);
5      if isSeed ∧ | bucket |< S_min then
6          bucket.setSeed(False);
7          cell ← bucket.cell.getHook();
8          cell.merge(bucket.cell);
9      end
10 end
```

then this bucket becomes a seed, and a `split` operation is triggered by s (see below). The pseudo-code of the `join` operation is presented in Algorithm 1.

`Split` operation. A cell `splits` into two smaller cells when the population of one of its buckets exceeds $S_{\min}$ nodes and the cell has already one seed. The cell `splits` along the dimension that corresponds to the largest distance between the two seeds. More precisely, let s_1 and s_2 be the two seeds whose coordinates are $s_1 = (x_1^{(1)}, \ldots, x_D^{(1)})$ and $s_2 = (x_1^{(2)}, \ldots, x_D^{(2)})$. Let $i_0 = \text{argmax}_{1 \le i \le D} |x_i^{(1)} - x_i^{(2)}|$. Then the cell is split along the hyperplane orthogonal to i_0 axis and passing through the point $\lfloor (x_{i_0}^{(1)} + x_{i_0}^{(2)})/2\rho_{i_0} \rfloor \rho_{i_0} e_{i_0}$ where e_{i_0} is the D dimensional vector with $e_{i_0}(i) = 1_{\{i=i_0\}}$. Both seeds s_1 and s_2 update their respective cell routing tables to point to the nodes whose bucket falls in respectively s_1 and s_2 cells, as well as their routing table to point to their respective neighboring seeds. Figure 2 depicts the `split` operation of cell 1.

`Leave` operation. Let p be a node, c be the cell node p sits in, and s be the seed in charge of c. When node p `leaves` the overlay (either voluntarily or not) then seed s simply discards p from its cell routing table. As presented in Algorithm 4, if p was sitting in s and the population of s undershoots $S_{\min}$ nodes, then p departure provokes the `merging` of cell c with another cell c' as described in the sequel.
Prior to describing the `merge` operation, we introduce the notion of *cell hook* represented in Fig. 2 by black triangles.

Definition 3 (Cell hook). *Let c be a cell in a D-dimensional FixMe overlay. Each corner of c has $2D$ neighbors buckets. The hook of c is the*

first bucket (in the lexicographic order) of these neighboring buckets that does not belong to c.

Proposition 1. *For a non-initial cell, the hook exists and is unique.*

Proof. Consider a cell c. By definition of a cell, c has 2^D corners. Each corner has $2D$ neighboring buckets. Among these neighboring buckets, the set B of buckets belonging to a neighboring cell has ℓ elements, with $\ell \in \{0\} \cup \{D, \ldots, 2D\}$. If c is the initial cell, $\ell = 0$ and thus $B = \emptyset$. Otherwise, c has at least one neighbor. In this case, $B \neq \emptyset$ and thus the hook exists. By definition, it is the first element of B in the lexicographic order. Thus it is unique. ■

`Merge` operation. A cell c `merges` with one of its neighbors c' when the population of its seed undershoots $S_{\min}$ nodes, and thus reverts to a default bucket. The cell c' with which c `merges` is determined as follows. If both c and c' share at least one face (we say that both cells are *sibling*), then both c and c' merge together in a single cell c'. On the other hand, if c has no sibling, then the cell that contains c hook takes in charge cell c. Thus a single seed may be in charge of several cells. In Fig. 2 on the right, the seed of cell 2 is also in charge of cell 5.

4.4 Self-organizing Nodes in Dense Seeds

In the context of QoS monitoring, it is not unusual to observe that a very large number of nodes perceive a quite similar QoS for a set of services. In such cases, FixMe would show cells with very dense seeds, that is seeds with a quite large number of nodes. Thus to keep the scalability property of FixMe, we propose to self-organize these nodes into a structured graph so that the routing cost among them remains logarithmic in their population size. Any structured graph proposed in the literature can be chosen. In this work we use PeerCube [16] essentially because each vertex of the hypercube gather from $S_{\min}$ to $S_{\max}$ nodes, which makes this cluster-based DHT highly robust to churn. Thus, in FixMe overlay as shown in Fig. 3, all the seeds are organized as follows. The first $S_{\min}$ nodes that are in a seed form the *root* of the hypercube, and upon new nodes arrivals, the dimension of the hypercube increases [16]. From the point of view of the neighboring seeds of any other seed s, only the root of the hypercube is visible.

5 Analysis

In this section, we evaluate the complexity of FixMe operations. There is trade-off between the number of seeds in the overlay and the number of nodes in the seeds. The two distributions that illustrate this trade-off are the uniform distribution and the Dirac one. The Uniform distribution maximizes the seeds number, and the Dirac distribution, which concentrates all the nodes in the same bucket, maximizes the dimension of the underlying hypercube.

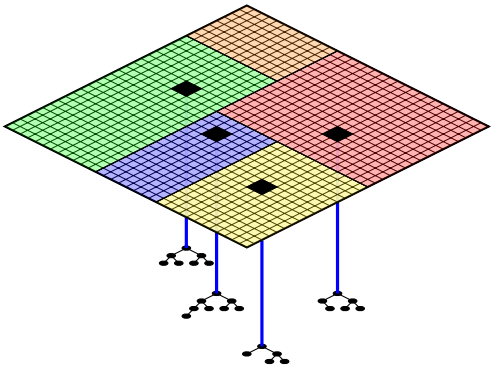

Fig. 3. FixMe cell-layer overlay and the embedded clusterized overlays.

Proposition 2. *The seed routing table has* $2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor$ *entries.*

Proof. As explained in Section 4.3, the distance between the cell centre and each entry $x_{(d,k)}$ of the routing table is bounded by R_d. Let $K_+^{(d)} = \max\{k \geq 1 \mid |x_d - x_{(d,+k)}| \leq R_d\}$ and let $K_-^{(d)} = \max\{k \geq 1 \mid |x_d - x_{(d,-k)}| \leq R_d\}$. Since $|x_d - x_{(d,+k)}| = |x_d - x_{(d,-k)}| = 2^k\rho_d$, we have $K_+^{(d)} = K_-^{(d)} = \lfloor\log_2(R_d/\rho_d)\rfloor$. Let $K^{(d)}$ be this common value. For each dimension d, the routing table has $2K^{(d)}$ entries. Thus, the routing table has $\sum_{d=1}^{D} 2K^{(d)} = 2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor$ entries. ■

Proposition 3 (Node `join`). *If an incoming node is inserted in a seed then, the insertion complexity is* $\Theta(\log H)$*, with* H *the number of nodes populating the underlying hypercube.*

Proof. If the incoming node belongs to the seed, there is no change in the cell layer. Thus, the complexity of this operation is only driven by the insertion in the underlying hypercube. It is well known that the complexity of this operation is $\Theta(\log H)$, where H is the number of nodes in the underlying hypercube. ■

Proposition 4 (Cell `split`). *If an incoming node insertion leads to a seed creation then, the complexity of this operation is* $\mathcal{O}(D)$*.*

Proof. As previously seen, the insertion of a node p might lead to a split operation (*cf.* `join` operation). This operation triggers only one write operation in the routing table of the concerned seed s_1. From the created seed s_2 point of view, node p is inserted in the hypercube root node. This operation is performed in constant time. Nevertheless, seed s_2 needs to build its routing table that will contain its neighboring seeds. As the routing table has $2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor$ entries, and as seed s_1 is necessarily a neighbor of s_2, then s_2 routing table creation will generate $2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor - 1$ `lookup` operations. ■

Proposition 5 (Node `leave`). *The `leave` of a node without topological change requires $\Theta(\log(H))$ messages number, with H the number of nodes in the underlying hypercube.*

Proof. When a node `leaves` its bucket, it is simply removed from its cluster in the underlying hypercube. Two cases are possible: either its cluster remains sufficiently populated, or its cluster has to merge with another one. In the former case, the cluster nodes simply update their view of the cluster. In the later case, $\Theta(\log(H))$ messages have to be sent to merge both clusters (See [16]). ■

When the hypercube has a single cluster populated by exactly $S_{\min}$ nodes (*i.e.*, the root cluster), a node `leave` makes the seed undershoot its population lower bound. Thus the corresponding cell c must `merge` with the cell containing c hook.

Proposition 6. *Let c be a cell, and p_i be the number of seeds that point to c along the ith axis. We have*

$$p_i \leq 2\log_2(n_i/2)\prod_{j=1,j\neq i}^{D} n_j$$

Proof. Let ℓ_j be the length of cell c along the jth axis. In the case where each neighboring bucket of c is a seed, c has at most ℓ_j/ρ_j immediate neighbors on each side. Thus, we have $p_i \leq 2K^{(i)}\prod_{j=1,j\neq i}^{D}\ell_j/\rho_j$. As shown in Proposition 2, we have $K^{(i)} = \lfloor\log_2(R_i/\rho_i)\rfloor$. Moreover, $\forall i \in \{1,\ldots,D\}$ we have $R_i \leq 1/2$ since in a torus unitary space, the farthest point is located at distance $1/2$. By definition $\rho_i = 1/n_i$, thus we have $p_i \leq 2\log_2(n_i/2)\prod_{j=1,j\neq i}^{D} n_j\ell_j$. The space being unitary, $\forall j \in \{1,\ldots,D\}$ we have $\ell_j \leq 1$, and thus we get $p_i \leq 2\log_2(n_i/2)\prod_{j=1,j\neq i}^{D} n_j$. ■

Proposition 7 (Cell `merge`). *If a cell merges, then the `merge` operation requires at most*
$2Dn^{D-1}\log_2(n/2)$ *messages, with* $n = \max_{1\leq i\leq D} n_i$.

Proof. By assumption of the proposition, the size of the corresponding seed s is equal to $S_{\min}$. Thus, when a node `leaves` seed s, this triggers a merge operation. The remaining nodes in s must contact the seed that will take in charge their seedless cell, and notify their predecessors. The number of predecessors p, which is given by $p = \sum_{i=1}^{D} p_i$, satisfies by Proposition 6, $p \leq 2\sum_{i=1}^{D} \log_2(n_i/2) \prod_{j=1,j\neq i}^{D} n_j$. By definition of n, we have $n^{D-1} \geq \prod_{j=1,j\neq i}^{D} n_j$, it follows that $p \leq 2n^{D-1}\sum_{i=1}^{D}\log_2(n_i/2)$ and thus $p \leq 2Dn^{D-1}\log_2(n/2)$. ■

Proposition 8. *The Uniform distribution and the Dirac distribution give rise to a `lookup` operation requiring* $\mathcal{O}(\log N)$ *messages.*

Proof. Suppose that the quality position of the nodes are uniformly distributed. Then the nodes will join fairly all the buckets. By construction, this maximizes the number of seeds and minimizes the dimension of each underlying hypercube. The dimensions of the hypercubes are equivalent and depend on the expected population H in each one. For N large enough, each bucket is a seed, and thus there are $\prod_{i=1}^{D} n_i$ seeds. Thus, since $n_i = 1/\rho_i$, we have $H = N\prod_{i=1}^{D}\rho_i$. As described in Section 4.3, a `lookup` operation is decomposed into three parts, namely, the hypercube traversal at the source of the `lookup` operation, the cells traversal and, the hypercube traversal at the destination of the `lookup` operation). As shown in [16], the dimension of an hypercube populated by H nodes equals to $\log(H/S_{max})$. By setting $S_{\max}$ to $\log N$, the number of messages required is equal to traverse an hypercube is equal to $\log(N\prod_{i=1}^{D}\rho_i) - \log\log N = \mathcal{O}(\log N)$. The cells traversal requires $\mathcal{O}(D)$ messages. Thus the total number of messages required for a `lookup` operation is $\mathcal{O}(\log N)$.
Suppose now that the quality position of the nodes follows a Dirac distribution. Then all the nodes will join the same bucket. The overlay is thus equal to the unique initial cell, and all the nodes belong to the same underlying hypercube. Its dimension is maximal, and thus $H = N$. By an argument similar to the previous one, the total number of messages required for a `lookup` operation is $\mathcal{O}(\log N)$. ■

6 Solving the Isolated Anomaly Detection Problem

We now propose an algorithm that solves the isolated anomaly detection problem. The algorithm, whose pseudo code is presented in Fig. 4, is

cyclically run by any node p, and is made of the following three tasks. Briefly, in Task 1, node p changes its position in FixMe overlay according to the QoS change of its monitored services (if necessary). If this QoS change is diagnosed as an anomaly by its function A, then p determines whether this anomaly is isolated or not (Task 2), and in the affirmative sends a FixMe message to the management operator (Task 3).
Let r be the current round of the algorithm. In Task 1 node p computes its current position $Q(p, r)$. Let b_r be the bucket that corresponds to this position, c_r be the cell that contains b_r, and s_r be the seed in charge of cell c_r. If $Q(p, r)$ differs from p position at time $r - 1$ (we note b_{r-1} the bucket that corresponds to this position), then p `leaves` bucket b_{r-1} and `joins` bucket b_r. If there exists a service i for which $A_i(p, r) =$ `true` then p runs Task 2. The goal of Task 2 is to enable node p to determine whether there are other nodes in the overlay that have experienced the same QoS change as p, that is, nodes that `left` bucket b_{r-1} at the beginning of round $r - 1$ and `join` bucket b_r at the beginning of round r. This is achieved as follows. By construction of FixMe, an hypercube is embedded in the seed s of each cell (see Section 4.4), and all the nodes in that cell point to the cluster root of seed s (see Section 4.3). Let H_r be the hypercube embedded in seed s_r. Then p computes a random key h that depends on both round $r - 1$ and its previous position b_{r-1} (see line 14 of Algorithm 5), and asks the node in H_r that is in charge of key h (by construction of any DHT, such a node always exists) to increment a counter v (initially set to 0 at the beginning of round r). After T time units, Task 3 starts. Node p reads counter v, and if it strictly less than τ (*i.e.*, no more than τ nodes have jump from bucket b_{r-1} to bucket b_r) then p sends a FixMe message to the management node, which ends Task 3.

Theorem 1. *Algorithm 5 solves the isolated anomaly detection problem.*

Proof.
The proof is made by contradiction. Suppose that at round r, *(i)* $k \leq \tau$ nodes experience the same change in their monitored qualities, *(ii)* such a change is large enough to be diagnosed as an anomaly, and *(iii)* none of these k nodes send a FixMe message at the end of the round r.
Let p be one of these nodes. At each round, p executes Algorithm 5, and in particular round r. By assumption (i), p has experienced a quality change and thus moves in the FixMe overlay from its current bucket b_1 to the new one b_2. By assumption (ii), p runs Task 2, and thus the counter tracking jumps from b_1 to b_2 is incremented. By assumption (i) $k - 1$ other nodes proceed as p. Thus at the end of Task 2, the counter value is less than or equal to τ. By Task 3, p (and all the other $k-1$ nodes) sends a FixMe message to the coordinator. Which is a contradiction with assumption (iii). This completes the proof of the theorem. ■

Proposition 9. *Algorithm 5 described in Fig. 4 requires $\mathcal{O}(\log N)$ messages.*

Proof. Straightforward from Property 3. ■

Algorithm 5: p.updatePosition(r:round)

Data: T: delay such that all nodes have moved to their new bucket (if necessary).
Output : The positioning of p in the appropriate bucket, and the sending of a FixMe message if p detects an isolated failure

```
1  begin
2      Task 1
3          r← r+1;
4          oldposition ← p.bucket ;
5          newposition ← Q(p,r);
6          newbucket ← p.lookup(newposition);
7          if newbucket ≠ p.bucket then
8              p.leave();
9              p.join(r,p);
10         end
11     EndTask
12     if ∃i, 1 ≤ i ≤ D, A_i(p, r)=true then
13         Task 2
14             h ← H(oldposition, r − 1);
15             p.incrementValue(h);
16         EndTask
17         Wait Until T;
18         Task 3
19             n ← p.cell.seed.get(h);
20             if n ≤ τ then
21                 send FixMe msg to Management Operator;
22             end
23         EndTask
24     end
25 end
```

Fig. 4. Isolated Anomaly Detection algorithm run by any node p

7 Conclusion

In this work, we have formalized the isolated anomaly detection problem. Such a problem is recurrent in various large scale monitoring applications, and in particular in the cable and telecom industry where it is of utmost importance to make the difference between isolated anomalies and network based anomalies. One of the reasons being a financial one. In this context we have proposed the FixMe tool that pushes monitoring to end devices, and by combining local algorithms to detection functions provides a scalable and efficient solution to the isolated anomaly detection problem. As a future work, we first plan to analyze the evolution of FixMe in a stochastic model to study, in particular, the influence of the distributions on the cells repartition and their sizes. The long term objective is the implementation, and deployment of FixMe.

References

1. Broadband Forum: TR-069 CPE WAN Management Protocol Issue 1, Amend.4, 2011

2. Rabkin, A., Katz, R.: Chukwa: a system for reliable large-scale log collection. In: Proceedings of the International Conference on Large Installation System Administration (LISLA). (2010)
3. Zhao, Y., Tan, Y., Gong, Z., Gu, X., Wamboldt, M.: Self-correlating predictive information tracking for large-scale production systems. In: Proceedings of the International Conference on Autonomic Computing (ICAC). (2009)
4. Desphand, A., Guestrin, E., Madden, S.: Model-driven data acquisition in sensor networks. In: Proceedings of the International Conference on Very Large Databases (VLDB). (2002)
5. Krishnamurthy, S., He, T., Zhou, G., Stankovic, J.A., Son, S.H.: RESTORE: A Real-time Event Correlation and Storage Service for Sensor Networks. In: Proceedings of the International Conference on Network Sensing Systems (INSS). (2006)
6. Vuran, M.C., Akyildiz, I.F.: Spatial correlation-based collaborative medium access control in wireless sensor networks. IEEE/ACM Transactions on Networking (TON) **14**(2) (2006) 316–329
7. Kalman, R.E.: A New Approach to Linear Filtering and Prediction Problems. Journal of Basic Engineering **82**(1) (1960) 35–45
8. Xiong, X., Mokbel, M., Aref, W.: SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). (2005)
9. Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y.: A Threshold-Based Algorithm for Continuous Monitoring of K Nearest Neighbors. IEEE Transactions on Knowledge and Data Engineering **17**(11) (2005) 1451–1464
10. Zhang, Z., Yang, Y., Tung, A.K.H., Papadias, D.: Continuous k-means monitoring over moving objects. IEEE Transactions on Knowledge and Data Engineering **20**(9) (2008) 1205–1216
11. Har-Peled, S., Sadri, B.: How fast is the k-means method? Algorithmica **41**(3) (2005) 185–202
12. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., Shenker, S.: A scalable content-addressable network. In: Proceedings of the SIGCOMM Conference. (2001)
13. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the SIGCOMM Conference. (2001)
14. Lin, J.: Broadcast scheduling for a p2p spanning tree. In: Proceedings of the IEEE International Conference on Communications. (2008)
15. Kovacs, B., Vida, R.: An adaptive approach to enhance the performance of content-addressable networks. In: Proceedings of the International Conference on Network and Computer Science (ICNS). (2007)
16. Anceaume, E., Ludinard, R., Ravoaja, A., Brasileiro, F.V.: Peercube: A hypercube-based p2p overlay robust against collusion and churn. In: Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO). (2008)