REDEMON: Resilient Decentralized Monitoring System for Edge Infrastructures

Roger Pueyo Centelles*, Mennan Selimi*†, Felix Freitag*, Leandro Navarro*

* Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain

† Max van der Stoel Institute, South East European University, North Macedonia

***Abstract*—The Guifi.net community network has evolved during the past 15 years into a telecommunications infrastructure that offers Internet access to more than 80.000 people. The monitoring system currently in place for this network is lagging behind the growth of the infrastructure, requiring manual intervention and counting several single points of failure. In this paper we present REDEMON, a resilient decentralized monitoring system, hosted on distributed and interconnected edge devices, for a reliable, eventually-consistent monitoring of the Guifi.net network, leveraging CRDT-based data structures implemented on AntidoteDB. We developed the REDEMON system as a prototype featuring resilience, decentralization and automation, in order to replace the legacy monitoring system. To assess the system, this prototype was deployed on resource-constraint edge nodes in the Guifi.net production network and evaluated under realistic conditions. The decentralized assignment mechanism successfully achieves setting the minimum number of monitoring servers per network device that satisfies the established system requirements. Besides, by concentrating the workload on the minimum required number of servers running at their maximum capacity, the remaining devices can idle away, reducing the consumption footprint of the system. With regard to computing resources, we measure a moderate CPU and RAM usage by the monitoring system on low-capacity devices, while we observe that a considerable network traffic is required for achieving a resilient and consistent data storage layer. This resilient and decentralized architecture could lay the basis for other edge applications in the cloud computing domain that need to coordinate over distributed and consistent shared data.**

***Index Terms*—distributed monitoring; edge clouds; CRDT;**

I. Introduction

The computing capacity at the network edge is growing and cloud computing, traditionally enabled by infrastructure service provision in large data centers, is moving to the network edge. The increasing availability of edge infrastructures is also pulling applications, which typically run in remote data centers, to operate on distributed edge devices.

This paper discusses a practical cloud computing use case consisting of a distributed network monitoring system that was implemented, deployed and evaluated on distributed edge devices. This real use case exists in the Guifi.net community network [1], where thousands of wireless routers as well as fiber optic equipment are interconnected forming an IP-based communication infrastructure. Figure 1 shows the network map of Guifi.net in Barcelona.

The legacy monitoring system for the Guifi.net infrastructure is built around a centralized database that contains a list with all the network devices, all the monitoring servers, and the assignments between them (i.e., which monitor is in charge of monitoring which devices). This system, while in general terms works and accomplishes its duties, has important limitations regarding robustness and resilience to cope with varying conditions of the network and its infrastructure [1] [2]. Because it lacks such features, it significantly hinders Guifi.net from keeping its sustained growth and improving its operation.

Figure 1. Network map of Guifi.net in Barcelona, where the REDEMON system is operated.

In this paper we present the design, implementation and experimental evaluation of REDEMON, a new resilient monitoring system for Guifi.net that leverages an edge-based eventually consistent database to support distributed and concurrent read and writes from monitoring servers. REDEMON integrates AntidoteDB [3], which implements CRDT-based (Conflict-free Replicated Data Type) distributed data structures [4]. The design of REDEMON supports the decentralized coordination of monitoring servers in the assignment of network devices across them. Furthermore, using AntidoteDB's CRDT implementations allows to automate the replication and distribution of information across different servers, while keeping data persistent, consistent and reliably stored. The main contributions of this paper are the following:

- Architecture and implementation of the REDEMON monitoring system, discussed in Section III.
- Experimental evaluation of REDEMON in the production network of Guifi.net, discussed in Section IV.
- Discussion of the results and achieved features, discussed in Section VI.

Besides them, in Section II we describe the details of the monitoring use case and we analyze the limitations of the legacy system. We review the related work in Section V. In Section VI we discuss our results before the conclusion.

[1] What is Guifi.net? - https://guifi.net/en/what_is_guifinet

II. MONITORING IN GUIFI.NET

A. Use case context

Guifi.net is a bottom-up, citizenship-driven technological, social and economic project with the objective of creating a free, open and neutral telecommunications network based on a commons model [5]. The whole network infrastructure can be seen as a crowdsourced, multi-tenant collection of heterogeneous wired and wireless network devices (more than 35.700 nodes) with a routable IP address, interconnected between them and forming a partially-meshed network. For the heterogeneous network infrastructure Guifi.net has become, monitoring is a key network service necessary for the vital operation of the network. It is a specific challenge for the development of this use case that the new monitoring system must operate using the existing distributed edge devices in Guifi.net [1]. Therefore, usage of remote cloud data centers for storing data and carrying out a centralized control, which a traditional implementation of a monitoring system might suggest, is not an approach that fully matches the characteristics of this network.

B. Legacy monitoring system

The legacy monitoring system currently in production in Guifi.net is built around a central MySQL database, coupled with the Guifi.net website, which lists all the nodes in the network (i.e., devices such as routers, switches, etc.) and assigns them to the different monitoring servers spread all over the network. The current system has three main limitations that make it fragile:

- Each router is monitored by only one server.
- When a monitoring server goes down, this is not automatically reported to the Guifi.net website or the central DB, so network nodes are left unmonitored.
- Data collected about one node are only locally stored in a single monitoring server.

C. New monitoring system requirements

Based on the shortcomings of the legacy monitoring system, we identify the following five main requirements for the new REDEMON monitoring system:

- *Redundancy*: every network device shall be monitored by an arbitrary minimum number of servers, greater than one. This means that monitoring servers should check which network devices have which monitors assigned and, if below the minimum number, autonomously decide to become a monitor for any of these devices.
- *Automated assignment*: the system shall make the assignment between network devices and monitoring servers automatically. On a permanent operation basis, the service should run autonomously without manual intervention.
- *Automated reconfiguration*: the system as a whole shall be able to automatically detect faulty monitors (e.g., due to network or hardware failures) and reassign the affected network devices to functional monitoring servers. This process should be carried out without manual intervention.
- *Data replication*: the collected data shall be replicated and distributed to different parts of the system. In the event of network partition or churn of some of the storage nodes, the data should still be available for being retrieved by the monitoring service from other parts of the network.
- *Load balancing*: the monitoring workload should be balanced over the network and the active monitors rather than concentrated on a few devices.

III. REDEMON MONITORING SYSTEM

A. System model

A decentralized infrastructure such as Guifi.net relies on distributed network resources to reach its full potential. This includes the monitoring system, which plays a critical role in the network operation. In order to ensure the previously stated requirements for the monitoring system (Sec. II-C), our goal is to solve the problem of mapping the available monitoring servers to cover all network devices.

Guifi.net can be modeled as an incomplete directed graph $G = (N, E)$, where N is the set of nodes that compose the network and E is the set of links (wireless or fiber optic) connecting pairs of nodes. Nodes in N without links to other nodes (i.e., isolated nodes) are discarded. The two types of nodes that we consider are: monitoring servers M ($M \in N$) and network devices to be monitored D ($D \in N$). The links are characterized by a given bandwidth B_{ij}, $\forall(i, j) \in E$, and latency L_{ij}, $\forall(i, j) \in E$, while each node has a particular QoS_i, $\forall i \in N$, derived from the real measurements in Guifi.net. For a monitoring system deployment, at most $M_{\max}$ replicas of the monitoring server can be placed. A monitoring server can be deployed in a node only if this node has a QoS_i greater than a minimum threshold (QoS_{min}). A link of a node will be used if its bandwidth is higher or equal to a given threshold (B_e). All the notation required is gathered in Table I.

The *monitoring servers* $\Leftrightarrow$ *network devices* mapping problem must consider the following constraints:

1) Each network device must be monitored by more than one monitoring server

$$\forall d \in \mathbb{D} | d_is_monitored_by_m_i | \sum_{i=1}^{M_{max}} m_i \geq 2 \qquad (1)$$

2) Admission control: At most, k replicas of monitoring servers can be placed in the network:

$$|M| = k \qquad (2)$$

B. Design principles

The main design principles of the REDEMON monitoring system are distribution and decentralization to improve resilience and reliability. To do this, REDEMON leverages

Sets	Description
G	Graph
N	Set of nodes
E	Set of links
Parameters	**Description**
M_{max}	Maximum number of monitor-server replicas
B_{ij}	Bandwidth requirement associated with link (i, j)
L_{ij}	Latency associated with link (i, j)
QoS_i	Quality of node i
QoS_{min}	Minimum quality

Table I
NOTATION OF INPUT VARIABLES

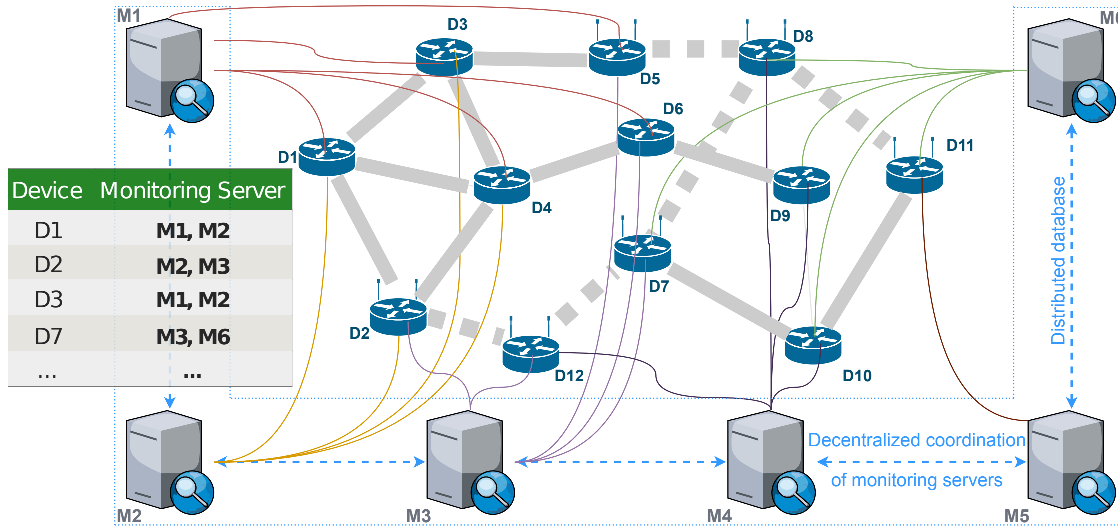

Figure 2. Conceptual depiction of REDEMON. Each network device in the center is monitored by two or more of the monitoring servers that coordinate among each other so that no device remains unmonitored. Solid and dashed links between nodes represent fiber and wireless connections, respectively.

distributed data structures to support the decentralized coordination of monitoring servers. For this purpose, the servers keep a distributed *monitoring servers* ⇔ *network devices* mapping, which they use to dynamically assign (and unassign) devices for themselves to monitor. This dynamic mapping is concurrently modified by any of the participating servers, incurring in operations that can lead to inconsistent data or that break the condition in Equation 1. To overcome this, we use the CRDT technology in the application and delegate data synchronization and consistency to the underlying data storage level, which ensures certain properties (e.g., strong eventual consistency in data replication) to the upper ones.

C. Architecture overview

The new monitoring system aims at solving the limitations of the legacy monitoring system and at providing comprehensive and reliable monitoring data for all network devices despite network partitions and server failures.

A conceptual depiction of the system is shown in Figure 2. There, the group of routers in the center represent the actual Guifi.net network devices, interconnected and creating a meshed network. Around them, several monitoring servers are shown, with dotted arrows between them indicating that they exchange information and coordinate between them. According to the requirements stated in section II-C, each of the network devices (i.e., the routers) is assigned to more than one monitoring server, indicated by the colored lines linking them. Since each server may have different capacity or available resources, some of them are in charge of monitoring more network devices than others. An important element, shown on the left side of Figure 2, is the proposed *monitoring servers* ⇔ *network devices* mapping, designed as a *shared distributed data object*, which is not managed centrally, but dynamically updated in a decentralized and autonomous way by the monitoring servers themselves.

The main functions of the monitoring system are conceived as *fetch*, *assign*, *ping* and *snmp*, explained below.

1) Fetch component: The purpose of the *fetch* component is to fetch the latest network description and feed it to the distributed database. The *monitor-fetch* application implementing it needs to be executed during the bootstrapping of the monitoring system. Its function is to write the required network infrastructure data to the distributed database. Afterwards, it can be run periodically (e.g., hourly) or on-demand, when the network infrastructure accounts for changes (new devices are added, hardware is decommissioned, etc.). The algorithm of the *fetch* component is described in Algorithm 1. As shown there, the *monitor-fetch* application parses a specified CNML (Community Network Markup Language) file and pushes its contents to the AntidoteDB database.

Algorithm 1 Monitor-fetch application

Require:
`CNML` ▷ URL of the CNML file containing the network description
`dbHost` ▷ Hostname/IP address of an AntidoteDB entry point
`dbPort` ▷ TCP port of an AntidoteDB entry point

Phase 1 – Fetch CNML file

1: **procedure** FETCH(`CNML`)
2: `fullCNML` ← *fetch*(`CNML`)
3: `devices[]` ← *process*(`fullCNML`)
4: *dumpDevicesToDB*(`devices[]`)
5: **end procedure**

2) Assign component: The purpose of the *assign* component is to [self-]assign the network devices to be watched by the monitoring server. The monitor-assign application that implements it runs on each monitoring server, and takes care of keeping the global *monitoring servers* ⇔ *network devices* mapping up to date as it is locally updated. For instance, when a network device is not being monitored by the required minimum number of servers, one or more of them eventually start watching it, until the requirement is met. This new assignment is immediately updated to the shared distributed data object and spread all over the network, eventually reaching the rest of monitoring servers. The assignment between monitoring

Algorithm 2 Monitor-assign application

```
Require:
  dbhost                 ▷ AntidoteDB hostname/IP address
  dbPort                 ▷ AntidoteDB TCP port
  id                     ▷ Unique ID of the monitor in the network
  minMonitors            ▷ Min # of monitors a device needs
  maxDevices             ▷ Max # of devices the monitor can watch
  Phase 1 – Monitor self-registration
 1: procedure REGISTRATION
 2:   monitorsList[] ← GetGlobalMonitorsList
 3:   AddMonitorToList(id,monitorsList[])
 4:   UpdateGlobalMonitorList(monitorsList[])
 5: end procedure
  Phase 2 – Monitor Self-assignment
 6: procedure ASSIGN(id)
 7:   numDevices ← 0
 8:   devicesInAntidote[] ← getDevicesInAntidote()
 9:   for each deviceInAntidote in devicesInAntidote[] do
10:     if (id is in deviceInAntidote.monitors[]) then
11:       numDevices++
12:     end if
13:   end for
14:   for each deviceInAntidote in devicesInAntidote[] do
15:     if (sizeOf(deviceInAntidote.monitors[]) < numDevices) && (numDevices < maxDevices) then
16:       assignMonitorToDevice(id,deviceInAntidote)
17:     end if
18:   end for
19: end procedure
  Phase 3 – Monitor Self-unassignment
20: procedure UNASSIGN(id)
21:   devicesInAntidote[] ← getDevicesInAntidote()
22:   for each deviceInAntidote in devicesInAntidote[] do
23:     if deviceInAntidote.monitors[] > minMonitors then
24:       unassignMonitorFromDevice(id,deviceInAntidote)
25:     end if
26:   end for
27: end procedure
  Phase 4 – Global assignment sanitization
28: procedure SANITIZE
29:   monitorsList[] ← GetGlobalMonitorsList
30:   localList ← GetGlobalMonitorList
31:   for each deviceInAntidote in devicesInAntidote[] do
32:     for each monitor in deviceInAntidote.monitors[] do
33:       if (monitor is not in monitorsList[]) then
34:         unassignMonitorFromDevice(monitor, deviceInAntidote)
35:       end if
36:     end for
37:   end for
38: end procedure
  Phase 5 – Monitor Self-deregistration
39: procedure DEREGISTRATION
40:   monitorsList[] ← GetGlobalMonitorsList
41:   RemoveMonitorFromList(id,monitorsList[])
42:   UpdateGlobalMonitorList(monitorsList[])
43: end procedure
```

servers and network devices is dynamic and evolves over time, as new network devices are added to the network or removed from it, or as workload balancing at the monitoring servers requires devices being reassigned from one server to another. The concurrently operating *assign* components do not directly communicate with each other; instead, they indirectly coordinate by means of shared distributed data objects. Last, but not least, the *assign* components can detect if a monitoring server has failed, retiring it from the system and taking over its monitoring duties.

The algorithm of the *assign* component is sketched in Algorithm 2. As shown there, once the *fetch* operation is finished (Section III-C1), servers know the list of nodes to watch (Phase 1), and coordinate with each other indirectly over the mutable data object given by the *monitoring servers* ⇔ *network devices* mapping in order to perform the actual monitoring of all nodes. The objective is to assign every single network device to –at least– the minimum number of 2 monitoring servers. This task can be performed in many ways. For instance, each monitoring server could start picking, at random, nodes not yet being watched and assign them to itself.

3) Ping component: The purpose of the *ping* component is to perform ping probes to the devices assigned to the monitoring server and to write the measurements to the database. The *monitor-ping* application implementing it is found on every monitoring server and takes care of the actual probing of network devices. The software periodically pings the assigned list of network devices for assessing their responsiveness, uptime and network distance (by means of ping packets' round-trip-time). All the collected data are stored to one instance of the distributed database. Replication of the data among all instances ensures that new data are automatically replicated and distributed to all database instances, providing storage redundancy. The algorithm of the *ping* component is described in Algorithm 3.

Algorithm 3 Monitor-ping application

```
Require:
  dbhost                 ▷ AntidoteDB hostname/IP address
  dbPort                 ▷ AntidoteDB TCP port
  id                     ▷ Unique ID of the monitor in the network
  localAssignCheckInt    ▷ Interval to check local assignment
  pingCheckInterval      ▷ Interval to perform ping requests
  Phase 1 – Continuous periodic update of the list of assigned devices
 1: procedure LOCALASSIGNUPDATE(id)
 2:   localAssignTicker ← localAssignCheckInt() * time
 3:   while localAssignTicker do
 4:     localAssignList[] ← RefreshAssignationList
 5:   end while
 6: end procedure
  Phase 2 – Continuous periodic ping requests to assigned devices
 7: procedure PINGREQUEST(id)
 8:   localPingQueryTicker ← pingCheckInterval() * time
 9:   while localPingQueryTicker do
10:     for each deviceToMonitor in localAssignList[] do
11:       for each IPv4 in deviceToMonitor.IPv4[] do
12:         PingData ← getPingData(IPv4)
13:         if PingData.status != OFFLINE then
14:           SavePingData[](deviceToMonitor,id,PingData[])
15:           break
16:         end if
17:       end for
18:     end for
19:   end while
20: end procedure
```

4) Snmp component: The purpose of the *snmp* component is to perform SNMP requests to the devices a monitoring server is assigned to and write the gathered SNMP values to the AntidoteDB database. The *monitor-snmp* application implementing it runs on every monitoring server and takes

care of the actual SNMP requests to the network devices. All the collected data are stored to an instance of the distributed database. Again, replication of the data among all instances ensures that the new data are automatically replicated and distributed to all database instances. The algorithm of the *snmp* component is depicted in Algorithm 4. It can be seen that the *monitor-snmp* application periodically asks the different devices for their SNMP information to get details about their interfaces and the inbound/outbound traffic.

Algorithm 4 Monitor-snmp application

```
Require:
  dbhost                  ▷ AntidoteDB hostname/IP address
  dbPort                  ▷ AntidoteDB TCP port
  id                      ▷ Unique ID of the monitor in the network
  localAssignCheckInt     ▷ Interval to check local assignment
  snmpQueryInterval       ▷ Interval to perform SNMP queries
Phase 1 – Continuous periodic update of the list of assigned devices
 1: procedure LocalAssignUpdate(id)
 2:   localAssignTicker ← localAssignCheckInt() ∗ time
 3:   while localAssignTicker do
 4:     localAssignList[] ← RefreshAssignationList
 5:   end while
 6: end procedure
Phase 2 – Continuous periodic SNMP queries to assigned devices
 7: procedure SNMPQuery(id)
 8:   localSNMPQueryTicker ← snmpQueryInterval() ∗ time
 9:   while localSNMPQueryTicker do
10:     for each deviceToMonitor in localAssignList[] do
11:       if deviceToMonitor.SNMPInterfaces[] == EMPTY then
12:         for each IPv4 in deviceToMonitor.IPv4[] do
13:           SNMPInterfaces[] ← querySNMPInterfaces(IPv4)
14:           if SNMPInterfaces[] != EMPTY then
15:             deviceToMonitor.SNMPInterfaces[] ← SNMPInterfaces[]
16:             break
17:           end if
18:         end for
19:       end if
20:       for each SNMPInterface in deviceToMonitor.SNMPInterfaces[] do
21:         for each IPv4 in deviceToMonitor.IPv4[] do
22:           SNMPInterface ← querySNMPInterfaceData(SNMPInterface)
23:           if SNMPInterface.data != EMPTY then
24:             break
25:           end if
26:         end for
27:       end for
28:       SaveSNMPData()(deviceToMonitor,id,SNMPData[])
29:     end for
30:   end while
31: end procedure
```

D. Distributed data structures

The monitoring system manipulates the data drawn from two sets of objects and creates a mapping between them. The first set contains a list with all the devices in Guifi.net that have to be monitored. All the Guifi.net devices are identified by a unique numeric ID (e.g., 58266), which remains immutable through all its lifespan. Additional information, such as the associated IPv4 addresses (e.g., 10.1.33.35) may be attached as a string-formatted JSON item. The nodes list of the whole Guifi.net contains around $35,700$ nodes, and grows at a rate of 25 nodes per day. The data in this first set is only modified by authoritative updates issued from the Guifi.net website; the monitoring servers only read it but do not modify it.

The second set contains a list with all the active monitoring servers. Servers are also identified by a unique numeric ID, being the servers list a subset of the nodes list (a monitoring server is indeed a device inside the network, with its own IP address, etc. that must be monitored too).

The mapping between the nodes list and the servers list is a collection of one-to-one relations between devices from the two lists. Any monitoring server may modify the mapping between nodes and servers (add, update or remove these relations) at any time. According to different criteria –such as current workload, network status and other– each monitoring server will, for instance, assign itself a number of nodes and will update the *monitoring servers* ⇔ *network devices* mapping accordingly. This assignment (Phase 2 of Algorithm 2) needs to change over time, as new nodes are added to the list, the network conditions change, workload is redistributed, monitoring servers join or exit the pool, etc. As a consequence, each monitoring server continuously –and not in synchronization with the other servers– reads and writes to the shared mapping object.

Given the nature of the application, and in order to successfully deal with concurrent updates of the mapping, eventual data consistency and integrity between the distributed database instances are required. By leveraging these properties, it can be ensured that all network nodes end up being properly assigned to monitoring servers.

E. Implementation

The new monitoring system uses AntidoteDB to implement a CRDT-based geo-replicated and distributed storage back-end. By leveraging on AntidoteDB, the monitoring components can use CRDT-based data structures for concurrent read and write of the assignments between network devices and monitoring servers. Furthermore, using AntidoteDB relieves the development tasks from the complexity of implementing a synchronization protocol for the monitoring servers to manage data coherency through all the system regardless of eventual failures or network partitions. Additionally, AntidoteDB provides an automatic mechanism to replicate and distribute the monitoring data all over the network, avoiding the burden of having to request monitoring data from different servers and assembling them in order to obtain detailed information about a specific device.

The monitoring servers host the designed components (*fetch*, *assign*, *ping*, *snmp*), as depicted in Figure 3. A typical monitoring server (i.e., a full-blown monitor) runs a local AntidoteDB instance that connects with the other AntidoteDB instances running in other places of Guifi.net, in order to provide the underlying highly available, geo-replicated and distributed storage for other monitoring server components to interact with. Also in the same figure, to the right, lightweight monitoring servers consist of the same components, but lack a local AntidoteDB instance; instead, they rely on a remote AntidoteDB instance running on another monitoring server to assist in the coordination process and provide the required storage.

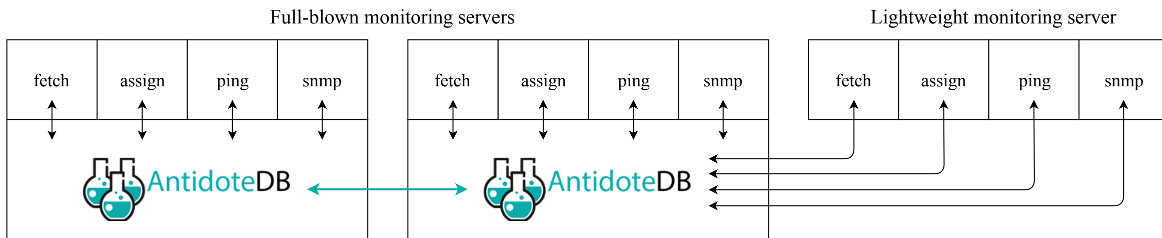

Figure 3. High level architecture of the monitoring servers. Full-blown servers run an instance of the geo-replicated AntidoteDB database and, on top, the four components involved in the monitoring tasks. Lightweight servers do not run a local AntidoteDB instance but rely on remote ones running in other servers.

We developed the four monitoring components (*fetch, assign, ping and snmp*) as a prototype implementation that uses the Go language and interacts with AntidoteDB through its Go client [2]. The source code is available at our GitLab repository [3].

IV. Evaluation

A. Objectives

We first study, in the *assign* operation, the decentralized coordination among monitoring servers. Then we focus our evaluation on the resource consumption of the monitoring system (CPU, RAM memory and bandwidth). In the context of our use case, these metrics are of high importance, since in Guifi.net the devices hosting the monitoring system are not dedicated. A moderate resource consumption of the monitoring system is desirable since the available computing resources on a device and in the network need to be shared with other applications.

B. Testbed

In order to achieve the objective explained above , it is needed to conduct the evaluation of the monitoring system in a real deployment (i.e., production network). In the real Guifi.net environment, monitoring servers consist of different hardware, which can range from resource-constraint single-board computers (SBCs) to desktop computers. In order to represent this situation, we have installed several x86 mini-PCs and Raspberry Pi boards in a wireless mesh network part of Guifi.net (at users' homes) to form a testbed in which these devices operate as monitoring servers. [4]

Figure 4 illustrates the deployed testbed and provides some information about the network characteristics (IP, bandwidth between nodes and RTT). The eight black nodes correspond to Minix Neo Z83-4 devices (Intel Atom x5-Z8350 4-cores CPU @ 1.44 GHz, 4 GB of DDR3L RAM and 32 GB eMMC) running Debian Strech. Each Minix device hosts an AntidoteDB instance. Most of the Minix devices are geographically far from each other with a few hops of wireless links between them. The ten red nodes correspond to Raspberry Pi 3B+ devices.

For the storage of the monitoring system data we run one AntidoteDB instance configured as data center (DC) inside a Docker container per Minix device. Having eight Minix devices, there are a total of eight DCs which are interconnected with each other. The data of each DC are fully replicated on the seven other ones.

[2] https://github.com/AntidoteDB/antidote-go-client

[3] https://lightkone.guifi.net/lightkone

[4] The wireless mesh network is GuifiSants; nodes and network topology can be found at http://dsg.ac.upc.edu/qmpsu/index.php

Minix device running AntidoteDB	Remote RPi monitoring clients
10.1.24.41	4
10.228.201.91	1

Table II
Minix devices where measurements were taken.

For the experimentation the monitoring system clients are located on 10 Raspberry Pi boards. On each Raspberry Pi we have installed four monitoring system components (*fetch*, *assign*, *ping*, *snmp*) to perform all the operations of the monitoring system. We take measurements at two of eight Minix devices, with a different number of connected Raspberry Pi clients (Table II). All ten Raspberry Pi clients were writing assign and monitoring data.

C. Characterization of the assign operation

The objective of this experiment is to observe the evolution of the assignment of network devices when the monitoring clients performing the *assign* operation join and leave. For conducting the experiment, first, using the *fetch* component, a data file with 54 devices of a small region of the Guifi.net infrastructure is stored in an AntidoteDB instance in order to have network devices to be assigned to monitoring servers. For observing the *assign* operation in this experiment, a customized setting with shorter routine periods of 10 sec is configured (instead of the default value of 150 sec). The required minimum number of monitors per device is set to 4 monitors. A maximum number of network devices per monitor is set to 30 devices. The assignment state is dumped every 5 sec, which is half of the period of the assign operations. The experiment is run for around 10 minutes. Up to 10 monitor *assign* clients are joining, one by one, and after approximately 6 minutes, are gradually leaving until having 0 clients at the end of the experiment.

Figure 5 shows the evolution of the number of monitors per device. It can be seen, in the red line, how the number of monitors reaches 10 monitors after around 340 sec. It can be seen in the dotted blue line that, when 9 monitors are running, the required minimum number of 4 monitors per device is assigned to all network devices. When the number of monitors decreases after 400 sec, it can be seen in the blue line that the total monitoring capacity decreases as well and some devices get assigned less than 4 monitors.

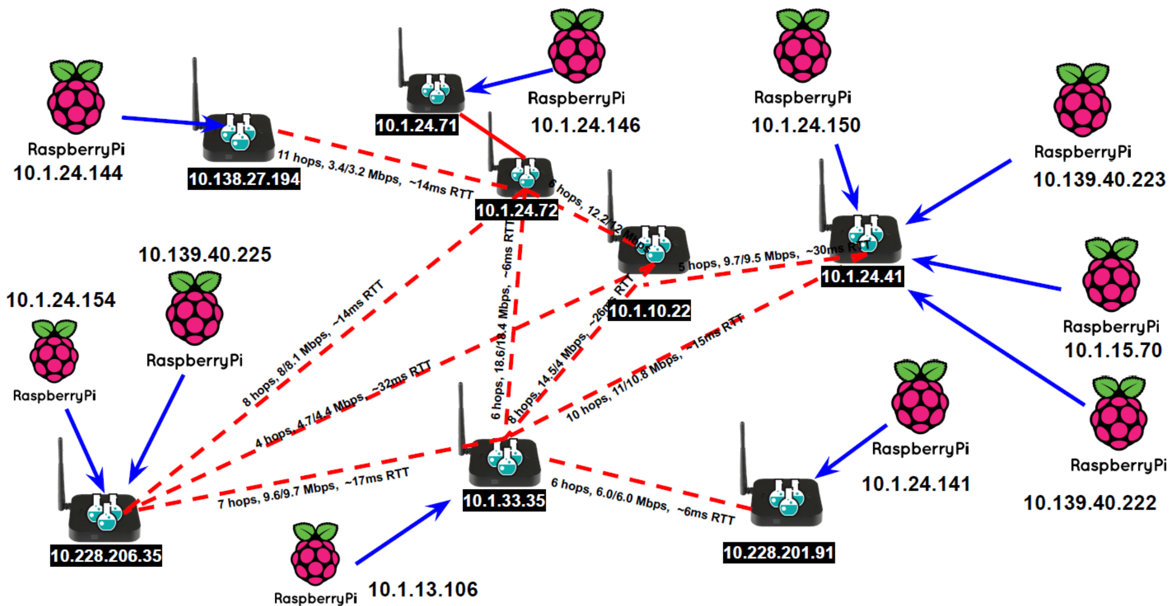

Figure 4. Testbed for REDEMON monitoring system deployment. The testbed is deployed in the GuifiSants wireless mesh network.[4]

Figure 6 shows the evolution of the assigned devices to all monitors. It can be observed that for the 54 network devices contained in the dataset used, 10 operational monitors turn into 216 device assignments in total, which corresponds correctly to the required 4 monitors per device.

Overall, for the *assign* operation it can be seen from the experimental results that, as the monitoring capabilities become available, the decentralized assign operations are able to fulfill the system requirements with regards to assigning the required minimum number of monitors to the network devices. It is worth noting the policy, which the assign component applies for its decision. It can be seen in Figure 5 that the *MaxMonPerDev* value does not go beyond 4, even if more monitoring capacities by a large number of servers are in the system. The reason is that once the configured system requirement of a minimum of 4 monitors per device is achieved, new monitors *do not assign themselves* in addition to monitor this device. This policy actually aims to fulfill the requirements with the minimum number of monitors and operating them at their maximum capacity, which reduces the consumption footprint of the system. Since this might conflict with other requirements from Section II-C, alternative policies could seek to maximize balancing the monitoring task among all monitoring servers. In that case, however, each monitoring server would operate below its maximum monitoring capacity.

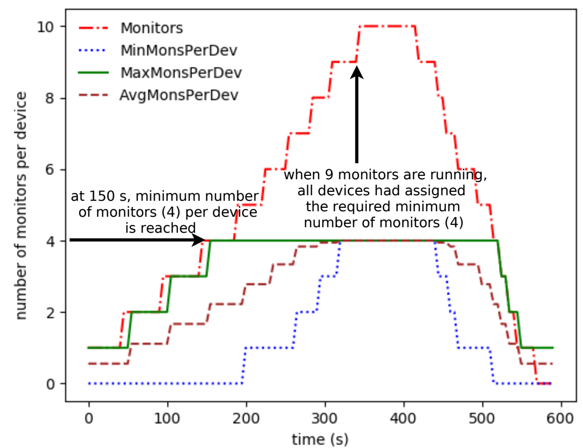

Figure 5. Evolution during ten minutes of assigned monitors per device with increasing number of joining monitors up to minute 6 and decreasing number until minute 10.

D. Results on resource consumption of monitoring operations

1) Fetch operation: The *fetch* component parses a specified CNML file and pushes its contents to AntidoteDB. For these experiments we use the description of the Barcelona sub-network, which consists of 1602 devices. The fetch operation is done from only one monitor-fetch component to an AntidoteDB instance. In our experiment, *fetch* was done from a Raspberry Pi client with IP 10.1.24.150 to the Minix device with IP 10.1.24.41. Link bandwidth with *iperf* was obtained to be approx. 8 Mbps, symmetric. *Traceroute* indicated 3 hops between the two nodes.

CPU and memory consumption: We measure *CPU load and memory consumption* at the Minix node with IP 10.1.24.41, which receives the writes from the fetch operation.

Figure 7 shows the CPU and memory consumption during the fetch operation of the barcelona.xml file. It took around one minute to store these data in the AntidoteDB instance of the Minix device. It can be observed that less than 1 core out of the 4 cores is fully used, and the memory consumption is low with regards to the available 4 GB of RAM.

Bandwidth consumption: We measure *traffic produced* during the fetch operation at the same Minix node.

Figure 8 shows the bandwidth consumption during the fetch operation. Since AntidoteDB is used with full replication of data, the traffic observed is not only produced by the communication with the monitor-fetch operation in the remote

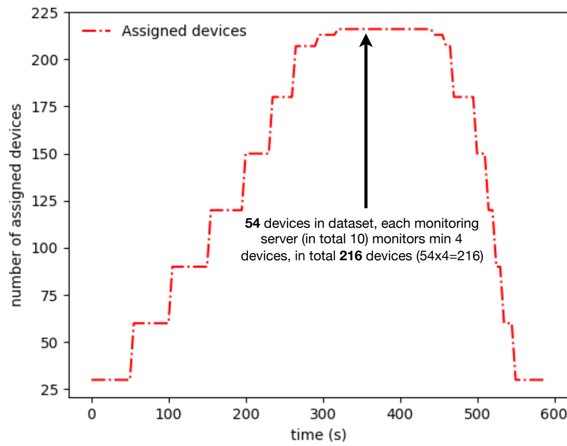

Figure 6. Evolution of the totally assigned devices to monitors as monitors join and leave during ten minutes.

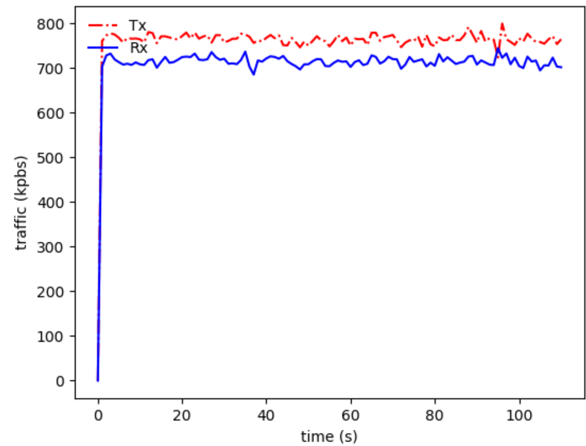

Figure 8. Traffic produced at the AntidoteDB instance during fetch in node 10.1.24.41, to which the monitoring client writes to.

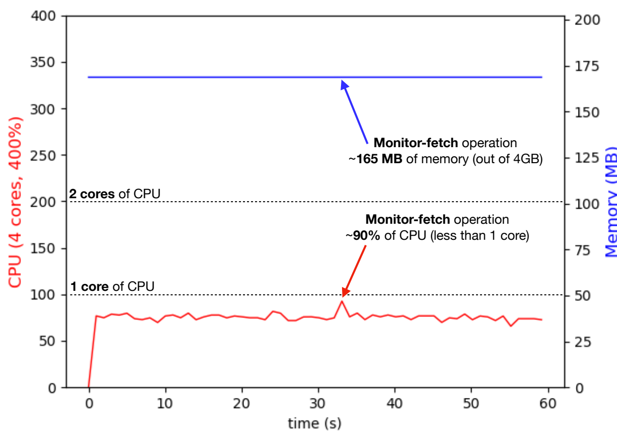

Figure 7. CPU and memory usage of AntidoteDB during fetch in node 10.1.24.41, to which the monitoring client writes to.

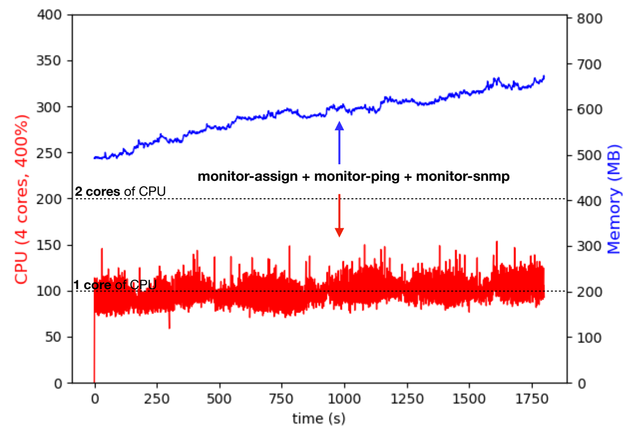

Figure 9. CPU and memory usage of AntidoteDB during assign, ping and snmp in node 10.1.24.41.

Raspberry Pi, but it is also produced from synchronizing the received data with the other AntidoteDB instances.

2) Assign & ping & snmp operations: These three operations are executed on each monitoring server (i.e., the ten Raspberry Pi). The purpose is to assign to the servers the network devices to monitor. Along with monitor-assign, monitor-ping runs on each monitoring server, pinging the assigned network devices and writes the obtained data to AntidoteDB. Also, monitor-snmp runs on each server, requesting SNMP data from the assigned network devices and writing the obtained data to AntidoteDB. The three components use the data storage provided by AntidoteDB. *Assign* uses a shared mutable data object, with concurrent read and writes from each monitor-assign component, while *ping* and *snmp* write immutable monitoring data. For the monitor-assign, monitor-ping and monitor-snmp operations the default settings are applied, in which periodically every 150 sec the ping and snmp monitoring operation is performed to the assigned network devices.

CPU and memory consumption: We measure *CPU load and memory consumption* at two out of the eight monitoring servers that host the AntidoteDB instances at the level of the AntidoteDB Docker container, when executing the assign, ping and snmp operations from the ten remote Raspberry Pi, which are connected to the hosts of the AntidoteDB instances as indicated in Figure 4.

Figures 9 and 10 show the CPU and memory consumption during continuous *assign*, *ping* and *snmp* operations for 30 minutes. It can be observed that around 1 core out of the 4 cores is fully used, and the memory consumption is low with regards to the available 4 GB of RAM.

Bandwidth consumption: We measure *traffic produced* at two out of eight monitoring servers that host AntidoteDB instances, at the level of the Docker container, when executing the assign, ping and snmp operations from the ten remote Raspberry Pi.

Figures 11 and 12 show the bandwidth consumption during the continuous assign, ping and snmp operations. Node 10.1.24.41 is a Minix device, which receives direct data writes from the three remote Raspberry Pi clients. Node 10.228.201.91 receives direct writes of data from one remote Raspberry Pi client. The traffic observed corresponds to these writes and to the synchronization traffic between the data replicas of the AntidoteDB instances. The periodicity that can be observed in Figure 12 can be explained by the period of 150s by which the *assign*, *ping* and *snmp* operations are performed. This periodicity cannot be observed clearly in the node 10.1.24.41

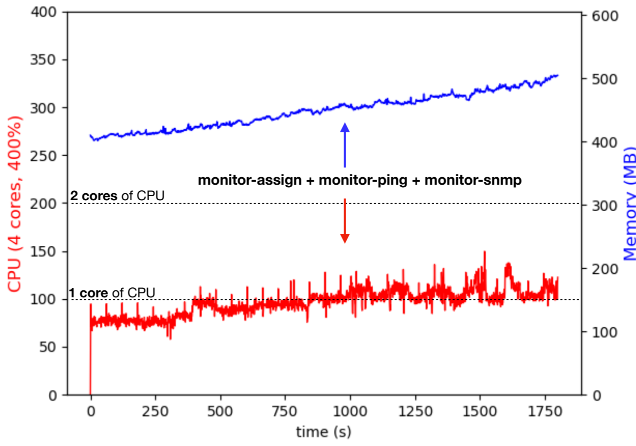

Figure 10. CPU and memory usage of AntidoteDB during assign, ping and snmp in node 10.228.201.91.

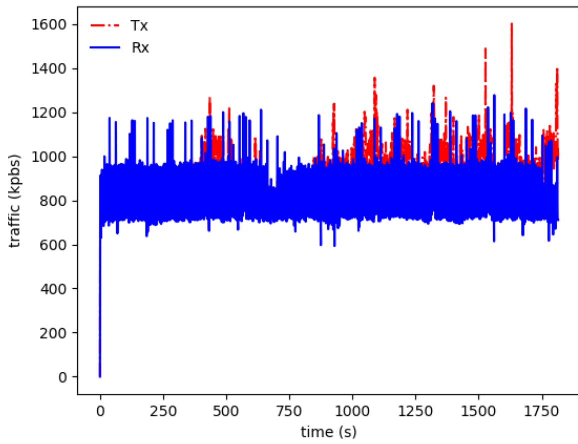

Figure 11. Traffic produced at the AntidoteDB instance during assign, ping and snmp in node 10.1.24.41.

of Figure 11, where the intense writes of their remote clients, each operating the *assign*, *ping* and *snmp*, are more continuous.

V. Related Work

There are many solutions running on top of public and private clouds that monitor cloud resource usage (e.g., CPU, memory, disk and network bandwidth). For instance, Amazon CloudWatch [6] is a monitoring and management service that monitors virtual resources of users such as Amazon EC2 instances. IBM Tivoli Monitoring [7] and HP Open View [8] are other monitoring systems aiming to optimize the performance and availability of IT infrastructures by focusing on the physical resources. GMonE [9] is a general-purpose cloud monitoring tool which proposes a unified cloud monitoring taxonomy based on which it defines a layered cloud monitoring architecture. PCMONS [10] is a private cloud monitoring system that can be adapted for use by cloud telephony providers to gather and centralize monitoring information, which should improve quality of services. PCMONS does not collect the information needed to map virtual resources to physical resources. MonPaaS [11] is an open source adaptive monitoring platform as a service (MonPaaS). MonPaas integrates Nagios [12] and OpenStack. MonPaas monitors physical and virtual resources and also updates any change in physical or virtual infrastructure. The disadvantage of MonPaas is that it consumes extra physical resources. DOCTraMS [13] is system that monitors and disseminates traffic conditions using a decentralized infrastructure.

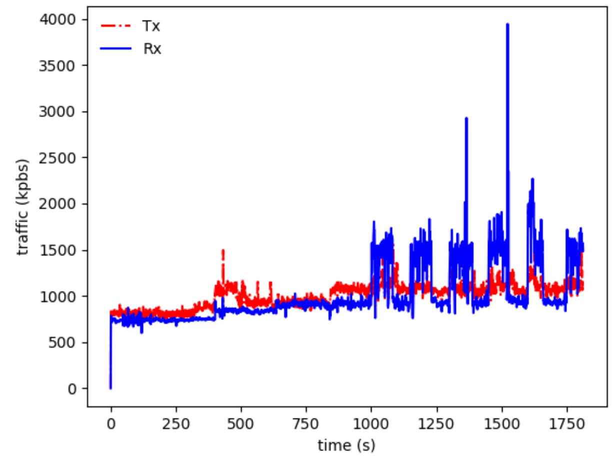

Figure 12. Traffic produced at the AntidoteDB instance during assign, ping and snmp in node 10.228.201.91.

The above mentioned works mostly consider data-centers or micro-data centers as their environment, where in our case distributed resource-constrained devices such as Raspberry Pi boards and mini PCs form the monitoring system infrastructure. Furthermore, in the conditions of Guifi.net the individual monitoring servers are not under a centralized control. In [14], we presented an initial version of the monitoring system, however it was limited to the *fetch* and *assign* operation. In REDEMON, all the monitoring operations are developed and the system is evaluated in a realistic testbed environment.

VI. Discussion of Results

Achieved features: With reference to the requirements, we conduct in Table III a comparison of the achieved features of REDEMON (the new monitoring system) with those of the legacy monitoring system. We can observe that with REDEMON we achieved an increased resilience of the monitoring system to the conditions in Guifi.net, where network partitions and server failures may happen, and where low capacity computing nodes are geographically distributed.

Robustness: The experiments were conducted in the real Guifi.net network such that the testbed nodes were exposed to all real network conditions [1] [5]. The workload was varied by experimenting with the default and other configurations of the monitoring system. A concrete operation limit of the monitoring system for a determined network or resource situation could not be identified, but it is in general clear that incrementing in magnitudes the periodicity of the *assign*, *ping* and *snmp* operations (default is 150 sec) will need a careful timing. For instance, in SNMP requests to the network devices, there are a few core routers in Guifi.net with a very large number of interfaces. An SNMP request to such routers consumes more resources than those to a low-cost wireless router used at the network edge.

Multi-tenancy: Full-blown monitoring servers and lightweight monitoring servers were deployed on Minix devices

REDEMON features	Legacy system
Automated assignment: The developed assign component is started without any previous network device assignment. It assigns to itself a number of network devices to be monitored according to a configured monitoring capacity of the server.	The network devices to be monitored are manually assigned to monitoring servers by the network administrator.
Automated reconfiguration: The assign component review with a configurable determined periodicity the current monitor-device mapping and reconfigures according to the current situation (e.g. deletes unresponsive servers, increases monitors for network devices which are undermonitored).	There is no automated update of the initial monitor-device mapping.
Redundancy: Each assign component checks periodically that every networking device is monitored by several servers (i.e., the monitoring servers check which network devices have less monitors and decide autonomously to become a monitor for any of these devices).	Each network device is monitored by only one monitoring server.
Load balancing between servers: Self-assignment decisions take into account the monitoring server capacity by configuration.	There is no specific mechanism to achieve load balancing among servers.
Data replication: The collected data is replicated on the distributed AntidoteDB instances. In the event of network partition or churn of monitoring servers the data is still available on the replicas.	The monitoring data storage is done at the local server and is lost in case of network partition or server failure.

Table III
COMPARISON OF REDEMON FEATURES WITH THE LEGACY MONITORING SYSTEM.

and Raspberry Pi boards. The testbed nodes were not dedicated to our experiments, but were running other services as well (independently run by the node owner) [15]. A low resource consumption of the storage system is important in order to allow multi-tenancy on a node without the monitoring system affecting other applications. Overall, we observed on the used Minix devices a moderate CPU and memory consumption by the monitoring system, leaving sufficient computing resources available to operate other services on the same devices. Bandwidth consumption of the monitoring system, however, may be an issue if it runs on a node with a very poor link.

VII. CONCLUSION AND OUTLOOK

The computing capacity at the network edge grows, pulling applications that traditionally run in remote data centers to operate on distributed edge devices. In this paper, a practical use case of cloud computing, consisting of a distributed monitoring system for the Guifi.net community network, was designed and evaluated. Its implementation integrated a distributed storage service leveraging AntidoteDB. In order to coordinate the monitoring servers, a shared distributed data object was applied, which uses AntidoteDB's CRDTs for providing strong eventual consistency of the data.

Results obtained were: first, concurrent writes to the database were successfully carried out from multiple locations with a number of workloads of different intensity, allowing to correctly perform the policy of the assign operation; second, CPU and memory consumption of the monitoring system on the edge nodes was moderate, where CPU was from around a forth up to half of the available cores. Memory consumption from 400 to 700 MB was low for the available 4 GB of RAM in the Minix devices that were used; third, traffic produced by the synchronization of the replicas in the distributed database was considerably high with observations of up to around 3 Mbps, which might be an issue to take into account for low bandwidth wireless links.

The design of the monitoring system applies a decentralized coordination among nodes. Each node reads the instantaneous coordination state to control its individual actions (i.e., which devices to monitor). This decision then starts the actual monitoring operation, which is conducted like in a traditional centralized system. It remains to be seen for future work if this design of organizing the operations into a decentralized control and a centralized processing can be generalized to other edge-based applications.

ACKNOWLEDGMENT

This work was supported by the European H2020 framework programme project LightKone (H2020-732505), by the Spanish State Research Agency (AEI) under contracts PCI2019-111850-2 and PCI2019-111851-2, and the Catalan government AGAUR SGR 990.

REFERENCES

[1] M. Selimi, L. Cerdà-Alabern, F. Freitag, L. Veiga, A. Sathiaseelan, and J. Crowcroft, "A lightweight service placement approach for community network micro-clouds," *Journal of Grid Computing*, vol. 17, no. 1, pp. 169–189, Mar 2019.

[2] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga, "Practical service placement approach for microservices architecture," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 401–410.

[3] "AntidoteDB: A planet scale, highly available, transactional database," https://www.antidotedb.eu/, 2019.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[5] D. Vega, R. Baig, L. Cerdà-Alabern, E. Medina, R. Meseguer, and L. Navarro, "A technological overview of the guifi.net community network," *Computer Networks*, vol. 93, pp. 260 – 278, 2015.

[6] Amazon, "Amazon CloudWatch," https://aws.amazon.com/cloudwatch/.

[7] IBM, "IBM Tivoli Monitoring," https://www.ibm.com/support/knowledgecenter/en/SS3JRN_7.2.0/com.ibm.itm.doc/itm_install06.htm.

[8] HP, "HP BTO OpenView," http://www.hp.com/hpinfo/newsroom/press_kits/2010/HPSoftwareUniverseBarcelona2010/HP_Applications_Portfolio_brochure.pdf, 2019.

[9] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, and G. Antoniu, "Gmone: A complete approach to cloud monitoring," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2026 – 2040, 2013.

[10] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall, "Toward an architecture for monitoring private clouds," *IEEE Communications Magazine*, vol. 49, no. 12, pp. 130–137, December 2011.

[11] J. M. Alcaraz Calero and J. G. Aguado, "Monpaas: An adaptive monitoring platformas a service for cloud computing infrastructures and services," *IEEE Transactions on Services Computing*, vol. 8, no. 1, pp. 65–78, Jan 2015.

[12] "Nagios: The Industry Standard In IT Infrastructure Monitoring," https://www.nagios.org/, 2019.

[13] T. T. de Almeida, J. A. M. Nacif, F. P. Bhering, and J. G. R. Júnior, "Doctrams: A decentralized and offline community-based traffic monitoring system," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 3, pp. 1160–1169, March 2019.

[14] R. Pueyo Centelles, M. Selimi, F. Freitag, and L. Navarro, "Dimon: Distributed monitoring system for decentralized edge clouds in guifi.net," in *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*, Nov 2019, pp. 1–8.

[15] M. Selimi, A. M. Khan, E. Dimogerontakis, F. Freitag, and R. P. Centelles, "Cloud services in the guifi.net community network," *Computer Networks*, vol. 93, pp. 373 – 388, 2015, community Networks.