MORPHEMIC

Design of a self-healing federated event processing management system at the edge

Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models In the Cloud

H2020-ICT-2018-2020
Leadership in Enabling and Industrial Technologies: Information and Communication Technologies

Grant Agreement Number
871643

Duration
1 January 2020 –
31 December 2022

www.morphemic.cloud

Deliverable reference
D2.1

Date
22 February 2021

Responsible partner
ICCS

Editor(s)
Yiannis Verginadis

Reviewers
Amir Taherkordi
Paweł Skrzypek

Distribution
Public

Availability
www.morphemic.cloud

Executive summary

The constant upsurge of the connected devices and services along with the ample numbers of heterogeneous data streams and their relay among different processing and persistence architectures, has augmented the demand for adequate and cost-efficient infrastructures to host them. This need is covered by Cloud computing as a paradigm that offers on-demand resources and enables its users to seamlessly adapt applications to the current demand. More recently, this has been extended to using multiclouds as a means to vest on the true power of resources or services provided independently to the cloud service vendor. Of course, this has further increased the complexity but also the importance of monitoring and analysis software solutions. The gravity of efficient monitoring of multicloud applications is highlighted by the fact that it serves as a knowledge-base for deriving corrective actions like scaling.

In this deliverable, we first report on the analysis of the most prominent open-source tools used for monitoring and we describe a time-series based approach for persisting such data to be used in the future for forecasting purposes. Secondly, we introduce the design of a configurable near real-time federated monitoring mechanism called EMS. EMS will be used as a performance monitoring system that aggregates and propagates information that may trigger application reconfigurations. Last, we discuss the self-healing capabilities of this federated event processing management system that will serve as the critical basis for reactively and proactively triggering the reasoning and subsequently the reconfiguration process in the Morphemic platform.

Author(s)

Verginadis Y., Patiniotakis I., Tsagkaropoulos A., Totow J.-D., Raikos A., Mentzas G., Apostolou D., Tzormpaki D., Magoutas Ch., Bothos E., Stefanidis V.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643

Revisions

Date	Version	Partner
18.12.2020	1.0	ICCS
29.12.2020	2.0	ICCS
11.01.2021	2.5	UPRC
15.01.2021	3.0	ICCS
17.01.2021	3.5	UPRC
10.02.2021	4.0	ICCS,UPRC
22.02.2021	Final	ICCS

Table of Contents

List of Figures

List of Tables

Glossary

Acronyms	Definition
API	Application Programming Interface
CAMEL	Cloud Application Model Language
CEP	Complex Event Processing
CP	Constraint Programming
CPU	Central Processing Unit
EDA	Event Driven Architecture
EMS	Event Management System
EPA	Event Processing Agent
EPM	Event Processing Manager
EPN	Event Processing Network
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
JMX	Java Management Extensions
QoS	Quality of Service
RAM	Random Access Memory
ROI	Return on Investment
RRDtool	round-robin database tool
SLO	Service Level Objective
SNMP	Simple Network Management Protocol
SWIM	Scalable Weakly-consistent Infection-style Process Group Membership Protocol
TLS	Transport Layer Security

UI	User Interface
UML	Unified Modelling Language
VM	Virtual Machine

1 Introduction

1.1 Scope

Nowadays, enterprise applications constantly impose increasing resource requirements to support an always ascending number of end-users as well as deliver them appropriate Quality of Service (QoS). This has significantly challenged the computational capacity and efficiency of the modern infrastructural resources. An abundance of cloud service offerings is accessible and is being rapidly embraced by the majority of small and medium enterprises based on its various advantages to traditional computing models. However, at the same time, the data intensive application requirements set new research dimensions that challenge the endorsement of cloud resources offered solely by a cloud service vendor. Specifically, such modern data-intensive applications entail the ample endorsement of multiclouds as deployment models, as a means to consider the advantages of cloud computing in its full extent. In this sense, adequate monitoring mechanisms provide key tools for managing such multicloud applications and ensuring their QoS, even in acute workload fluctuations scenarios. It is essential to apply efficient resource and application monitoring techniques to collect data from dispersed and heterogenous resources, understand the current status of the application and its hosting infrastructure but also facilitate predictions and mitigation actions that may enhance the overall application performance. Performance is always an issue for enterprise applications, either to support a particular Return on Investment (ROI) or to provide guaranteed QoS to end users [1][2].

A well-known and typical way to implement proper resource management in order to enhance the application performance, refers to the back-end support for *resource* and *application* monitoring [3][4][5]. Resource monitoring addresses the measurement of monitoring information, such as CPU load, RAM usage, network inbound and outbound, disk usage or any other resource-related metric by employing continuous or discrete data collection procedures [3]. Application monitoring refers to the analysis of the application behaviour regarding its computation and communication aspects along with the detection of execution patterns that enhance resource management decisions [5]. Both application and resource monitoring can be performed through distinct levels of intrusiveness concerning the application itself. The monitoring mechanisms can be classified as active or passive, considering the amount of disturbance they inject into the system [6],[7]. Active monitoring refers to the simulation of synthetic transactions into the application environment to measure results while passive monitoring refers to the collection of data using various methods (e.g., SNMP, log scanning, Syslog, HTTP, etc.). In an attempt to obtain the advantages from both types of monitoring, some approaches try to combine active and passive mechanisms, which can significantly increase the respective cost and overhead. In most cases, it is necessary to find a trade-off to monitor the system in an optimal way. Therefore, there is a continuous discussion on the monitoring quality and how to achieve it. There is a need to consider some trade-offs for designing and setting up the monitoring system in terms of the used aggregation methods and sampling periodicity. These kinds of trade-offs have an impact on the consumption of network and computational resources for the monitoring process [6].

Especially, in a multicloud environment with heterogenous resources that may reach even the edge of the network, the monitoring system should not only be able to configure the aggregation means and the periodicity of the monitoring metrics collected but also provide a distributed processing infrastructure [8][9]. This is important for two reasons: as this firstly implies a minimum footprint on the network resources used for monitoring purposes (in comparison to a centralised approach) and secondly it avoids bottlenecks and single point of failure incidents. This becomes evident when we consider that in the case of centralized monitoring, all resource states and information are sent to a centralized monitoring server. These are continuously pulled as metric values from each monitored component, allowing for a quite controlled management of any cloud or multicloud application. Nevertheless, this also presents some very concrete disadvantages, such as the single point of failure and lack of scalability. This means that at any given time, if the monitored application exceeds the capability of the centralised monitoring server, only vertical scaling can be considered [15]. Moreover, in the case of multicloud applications where, by default, there is a need to aggregate monitoring data from a vast number of dispersed hosts, the high network traffic can lead to bottlenecks which frequently result in faulty or incomplete monitoring data. A decentralized approach can alleviate most, if not all of these problems [8][9].

Considering all of the above, we build on our previous work with respect to a decentralised cloud application monitoring system. We refer to the Event Management System (EMS) [8][9] that we will extend to create a federated event processing management system that copes with heterogenous resources in multicloud and edge environments. In such environments, connectivity issues, resources failures or any other issues that may cause downtime are quite frequent. If such issues are detected in resources that host the monitoring functionalities of a federated system, then repercussions might be immense. Therefore, we also focus on enhancements with respect to self-healing capabilities,

which constitute a necessity for critical systems that span several cloud service vendors, network and geographical boundaries.

In this context, next we explain the notion of federation. According to [10] a centralized network refers to an interconnected structure in which all nodes send their data to one central node (a server) which then sends the data to the intended recipient, while in a distributed network there is no central server, each node is connected to various other nodes and data is transferred through whichever nodes allow for the shortest route to the recipient. In the same work [10], the decentralized network is also defined as a distributed network of centralized networks. The notion of federation has been associated with independent and self-governing entities that transfer a set of their responsibilities to the central entity that unites them all. In the literature the notions of federation and decentralization are often interchangeably used while in reality the notion of decentralization should be a superset concept that encapsulates both distribution and federation [11]. In the cloud domain, a federated approach has been defined [12] as a structure in which "a single entity serves as a gateway to different independent cloud solutions to solve the limited interoperability, as different technologies can be unified and abstracted towards the consumers". Also, in [13][14] cloud federation refers to "different architectures and levels of coupling among distributed cloud instances". Therefore, in the context of Morphemic, we refer to *federated EMS as the capability of the event management system to deploy and maintain a distributed network of self-sustained centralized subnetworks that serve as gateways to different monitoring and event processing agents that are coupled in different levels across multiclouds and edge resources.* The different levels in *coupling* refer to either direct connections among nodes for propagating raw monitoring metrics (e.g., from a resource constrained device) or the use of different nodes (depending on their capacity) for aggregating raw information and detecting complex event patterns to propagate. The *self-sustainability* refers first to the automatic and independent, from a central server, allocation of aggregation and event processing responsibilities to certain nodes of the monitoring network and second their resilience against failures, without the need of manual intervention or orchestration from a centralized network entity.

In general, resource and application performance monitoring, load prediction and resource management present a strong correlation towards improved performance in enterprise systems. EMS will be used as a performance monitoring system that aggregates and propagates information that may trigger application reconfigurations. In addition, by monitoring and aggregating monitoring metrics (e.g., CPU or RAM load every minute) it is possible with appropriate techniques to forecast the values of these metrics in a future time (e.g., CPU or RAM load in the next hour). This prediction can serve as input for the deployment reasoning process, launching a Virtual Machine (VM) or a container to deliver resources before the actual problem or request occurs [6]. Morphemic tries to focus on this correlation so as to proactively optimise the multicloud applications management. Therefore, in WP2 we also focus on issues on how to persist vast monitoring data and constitute them available for advanced forecasting mechanisms, which will be able to predict imminent problems and trigger proactively the reconfiguration process.

1.2 Document Structure

This deliverable continues with an analysis of real-time performance monitoring tools in Section 2. Specifically, we discuss on the aspects of monitoring tools with a special emphasis *on the way that monitoring data can be collected* by prominent monitoring tools. We provide an overview of 20 well-known commercial and open-source tools and provide a comparative analysis that will lead us in the selection and integration of a monitoring probes technology into the Morphemic event management system. This constitutes one of the first extensions of EMS towards a straightforward way to collect raw monitoring metrics. Next, in a similar manner we provide a discussion on time-series databases that will have a critical role in the Morphemic architecture for persisting, maintaining and preparing training data for the forecasting mechanism that will undertake the role of predicting (workload) trends. In Section 3, we discuss deployment and configuration aspects of monitoring probes and the selected time-series database. Section 4 begins with a short overview of the Event Management System (EMS) introduced in the H2020 Melodic project as an efficient and distributed monitoring system which is appropriate for multicloud applications. The main focus of Section 4 is on the design and implementation of significant extensions to EMS towards a federated and self-healing event management system, able to cope with heterogenous resources that can reach the edge of the network. Lastly, Section 5 concludes this report and discusses the next steps in terms of the WP2 work.

2 Analysis of Real-time Performance Monitoring Tools

2.1 Comparative Analysis of Prominent Monitoring Tools

It is well-known that monitoring is an imperative process to ensure the efficient performance of a computing system. It can be distinguished in resource and application monitoring. The first refers to the cumbersome task of collecting data from physical or virtualized environments (e.g., CPU utilization, network jitter, packet losses, etc.). Such data can

assist towards maintaining the resources' availability at high efficiency and also detecting abnormal situations (e.g., SLO violations) that require immediate mitigation actions (i.e., reconfiguration). We refer to software that tracks and publishes such data as *monitoring probes* that involve at least two basic dimensions when it comes to resource monitoring: computation and network. Computation monitoring aspects refer to monitoring data that allow the inference of the status of real or virtualized resources (e.g., CPU Speed, CPU utilization, RAM utilization, disk throughput, VM startup/acquisition/release time/up-time). Network monitoring aspects are related to the tracking of network-layer metrics such as one-way and two-way delays, network jitter, throughput, packet loss, available bandwidth, capacity, and traffic volume [3]. On the other hand, application monitoring is an equivalently important process of measuring the performance of different kinds of applications. This monitoring kind is essential since it can provide monitoring information to multicloud management platforms (as Morphemic) in order to improve the efficiency and timing of computing jobs. One of the main goals of the application monitoring is to extract application patterns [3]. Such patterns can be used to provide performance feedback as well as to offer important knowledge for the application components optimization. The objective is to guarantee that the service level objectives (SLOs) for the application are satisfied and in order to do so a complete awareness of the applications' running state is required. This is not an easy task, especially for multicloud applications and Big Data frameworks [15] as it requires a *cross layer monitoring* scheme in order to determine the main factors that impact the quality and performance of different applications types [17]. In general, the cross layer monitoring implies the use of different type of monitoring probes depending on the cloud technology stack: i) IaaS level – monitor resource utilization such as CPU and Memory usage; ii) PaaS and SaaS level - status of system services, uptime, availability etc. For example, in the case of a Hadoop deployment there are metrics such as MapReduce processing time, job turnaround, shuffle operations, etc. [15]. Therefore, the monitoring probes that should be used along with their configuration is highly dependent on the application type (e.g., data transfer quality and rate for video streaming applications, process and network latencies for batch processing applications).

This cross-layer monitoring is even more challenging when discussing big data frameworks deployed on multicloud-based environments. This is true due to the fact that components of individual applications can be distributed on various cloud layers as well as on multiple VMs or other hosting resources [15]. For this purpose, the monitored probes should be automatically set up and connected for transmitting across all the cloud resources used by the applications in order to have a complete picture of the current status of individual applications at a given time.

In this section, we focus on a comparative analysis of the most prominent monitoring tools that could be integrated in Morphemic for tracking all the monitoring parameters (both for applications and hosting resources) that are required for guaranteeing the SLOs provided by the DevOps. The objective is to pinpoint the most appropriate monitoring probes that can enhance the federated event management system with the configurable volumes of raw monitoring data. This is an important extension of the Melodic EMS [16] which initially it was based on a limited list of resource monitoring probes and a number of application-specific ones that were installed under the instructions of the Melodic Executionware. With this extension, any available monitoring metrics can be measured and aggregated in a unified way with any application-specific metrics that may be defined by the DevOps.

According to several classifications [18], a monitoring tool can perform one or more of the following actions on monitoring data: (i) collect, (ii) transport, (iii) process, (iv) store, and (v) present. A data collector or else a monitoring probe is usually either a daemon running on the monitored host or an agent scraping data from monitoring APIs exposed by the resource being monitored (e.g., JMX). Monitoring data can also be collected via code instrumentation, as developers may use APIs to collect data [18]. Below we briefly introduce, in alphabetical order, each one of the most prominent monitoring tools that we examined.

2.1.1 Overview of Prominent Monitoring Tools

Amazon CloudWatch

Amazon CloudWatch[1] is a vendor specific web service for real-time monitoring of AWS services (e.g., EC2). Specifically, it provides data on resources utilization such as CPU, disk, network as well as monitoring information on the status of other resources like Amazon relational database service (Amazon RDS), DynamoDB, Amazon EBS volumes, Amazon S3, AWS Lambda, etc. For providing memory, disk space, or load average metrics one needs to run additional software on the instance side, i.e., CloudWatch Agents that can be deployed on Windows and Linux operating systems. The monitoring data are aggregated and provided through AWS management console or through dedicated Web APIs. Moreover, Amazon CloudWatch accepts custom metrics that can be submitted programmatically via Web Services API and then monitored the same way as all other internal metrics (e.g., CPU usage). The internal metrics can be published with a rate of up to 1-minute while the custom metrics with up to 1-second granularity. It

[1] https://aws.amazon.com/cloudwatch/

also supports setting up of basic alarms on internal and custom metrics that enable the auto-scaling feature to dynamically add or remove EC2 instances. As a commercial service there is charging by the number of monitoring instances observed.

Apache Chukwa
Apache Chukwa[2] is an open source data collection system for monitoring large distributed systems focused on the Hadoop Distributed File System (HDFS) and the Map/Reduce framework. It includes a flexible and scalable toolkit for monitoring, displaying and analysing results to make the best use of the collected data. Although it is designed to collect the monitored data, it provides the users with a toolkit able to better understand the collected data [15]. That is, it is capable of analysing and displaying the results of different runs of the monitored software. It uses agents to acquire HDFS related data which are persisted into files until a 64MB chunk size is reached or a given time interval has passed. Then the DemuxManager and the PostProcessManager filter and analyse the data collected.

Azure Cloud Monitoring
Microsoft offers the Azure Monitor[3] service to provide insights across workloads by monitoring applications, analysing log files, and identifying security threats. This vendor specific service provides a full view of the utilization, performance, and health of applications, infrastructure, and workloads without having to download any extra software as it is inbuilt into Azure. It delivers services for collecting, analysing, and acting on telemetry from cloud and on-premises environments while it offers advanced analysis of monitoring data for correlating infrastructure issues. Furthermore, this service supports the addition of any custom metric deemed necessary for the user and present alerts on them. All data collected by this service fits into one of two fundamental types, metrics and logs. Metrics are numerical near real-time values that describe some aspect of a system at a particular point in time, in a lightweight way. Logs contain different kinds of data organized into records with different sets of properties for each type. Telemetry such as events and traces are stored as logs in addition to performance data so that it can all be combined for analysis.

Cacti
Cacti[4] is an open-source, network monitoring and graphing tool designed on top of the round-robin database tool (RRDtool). It enables the polling of monitoring data at predetermined intervals and graph the results. It is generally used to graph time-series data of metrics such as CPU load and network bandwidth utilization. Cacti is mainly used for monitoring network traffic by polling for example network switches or routers via Simple Network Management Protocol (SNMP). It involves two types of monitoring means: a PHP script suitable for smaller installations, or a C-based poller which can scale to thousands of hosts. It utilizes a data push model, i.e. the data is collected and pushed to a multicast group or server. In addition, it supports data gathering on a non-standard timespan while it enables custom data-gathering scripts.

Collectd
Collectd[5] is an open-source UNIX daemon that collects, transfers and persists performance and network related data. It is a widely used tool, offered directly to DevOps or through other monitoring platform that mainly use it for collecting metrics. It uses Java Management Extensions (JMX) to extract valuable metrics related to the JVM which also enables the monitoring of Big Data frameworks (since most of them are Java-based complex frameworks). Furthermore, it follows a modular design which means that data acquisition and storage is handled by plug-ins in the form of shared objects while the daemon implements the capabilities of filtering and relaying of the data. Therefore, especially the daemon requires minimum resources, constituting useful also for embedded devices.

Datadog
DataDog[6] is a well-known commercial monitoring solution that provides full stack monitoring for cloud-scale applications, focusing on servers, databases, tools, and services, through a SaaS-based data analytics platform. It offers a dashboard, alerting, and visualizations of metrics and covers all major public and private cloud technology providers AWS, Microsoft Azure, Google Cloud Platform, Red Hat OpenShift, VMware, and OpenStack (more than 350 integrations offered out-of-the-box). Through a Go-based agent, it can handle infrastructure monitoring, application

[2] http://chukwa.apache.org
[3] https://docs.microsoft.com/en-us/azure/azure-monitor/overview
[4] https://www.cacti.net/
[5] https://collectd.org/
[6] https://www.datadoghq.com/

monitoring as well as log management. Its backend is built on top of a number of open and closed source technologies including D3, Apache Cassandra, Kafka, PostgreSQL among others. DataDog advertises that it is a turn-key solution for aggregating metrics and events for full devops stack.

Dynatrace
Dynance[7] is a commercial software intelligence platform based on artificial intelligence to monitor and optimize application performance and development. It provides observability of the full solution stack to deliver application performance monitoring. It is able to discover, map, and monitor applications, microservices, container orchestration platforms (e.g., Kubernetes), and IT infrastructure running in multicloud and hybrid cloud environments and provides automated problem remediation. Dynatrace consists of OneAgent for automated data collection, SmartScape for continuously updating topology mapping and visualization, PurePath for code-level distributed tracing, and Davis, a proprietary AI engine, for automatic root-cause fault-tree analysis. Furthermore, it provides SaaS and managed service deployment models.

Google Cloud Monitoring
The Google Cloud Monitoring[8] is a vendor specific service that provides visibility into the performance, availability, and health of applications and infrastructure in the Google Cloud environment (Cloud, Anthos, Kubernetes, Istio, etc.). It offers automatic out-of-the-box metric collection for Google Cloud services which is visualized in dashboards while it supports monitoring of hybrid and multicloud environments through the installation of an open-source agent[9]. This agent is a Collectd[10]-based daemon that gathers system and application metrics from virtual machine instances and sends them to the monitoring service. Furthermore, this service supports custom metrics to monitor application and business-level information while it can automatically infer.

Hadoop Performance Monitoring
Hadoop Performance Monitoring [19] is a big data framework focused tool, designed as a built-in solution for finding issues in Hadoop set-ups as well as providing visual representation of the available tunable parameters that may increase the performance of Hadoop. It is considered to be a lightweight monitoring tool for Hadoop servers. An advantage of this tool is that it is built in the Hadoop ecosystem and easy to use [15]. Although built-in, it lacks performance, a good example where it lags can be considered the time spent in garbage collection by each of the tasks [15]. It provides YARN REST APIs and is capable of visualising node-level log hierarchy (i.e. stderr, stdout, and syslog).

Icinga
Icinga[11] is a popular open-source monitoring application which was originally created as a fork of the Nagios monitoring system. It is written in C++ and PHP for cross-platform monitoring with additional database connectors and features in comparison to Nagios. It requires a master component to be hosted only on Linux in order to aggregate, process and visualize monitoring data. Its REST API allows administrators to integrate extensions while it permits the funnelling of monitoring data from other domain specific monitoring applications. It offers an alerting service that based on certain preconfigured thresholds it notifies the appropriate contact persons for performing mitigation actions. Moreover, it offers analytics to discover relation and patterns through visualizations. Icinga can monitor network services (e.g., SMTP, POP3, HTTP, NNTP, ping, etc.) and aspects of the host resources (e.g. CPU load, disk usage, etc.) while it is open to extensions that might aggregate application-level monitoring data. Nevertheless, Icinga seems to have a high learning curve, while adding new hosts and services dynamically is not straightforward. Last, it offers dedicated modules to monitor private clouds (e.g., based on VMWare, OpenStack), public clouds (e.g., AWS, Microsoft Azure, GCP) and can cope with monitoring tasks in hybrid clouds.

LogicMonitor
LogicMonitor[12] is a commercial cloud-based monitoring platform that is able to cope with hybrid infrastructures. It is an extensible platform which is able to monitor several aspects of the IT stack and applies machine learning to further process and exploit monitoring data to optimize the IT environment. It provides many integrations to monitor cloud

[7] https://www.dynatrace.com/
[8] https://cloud.google.com/monitoring/
[9] https://cloud.google.com/monitoring/agent
[10] https://collectd.org/
[11] https://icinga.com/
[12] https://www.logicmonitor.com/

resources from different vendors (e.g., AWS, Azure, Google Cloud, etc.). Specifically, it supports the installation of collectors on Virtual Machines or Servers in any cloud instance to be able to use out-of-the-box infrastructure performance metrics. Moreover, it provides metrics for container and microservices monitoring as well as network, database and application-level performance aspects to be presented in a single unified platform.

Munin

Munin[13] is a free and open-source network monitoring and infrastructure monitoring software application. It is written in Perl and provides graphs which are accessible over a web interface. It offers plug and play capabilities through 500 monitoring plugins. It is lightweight enough to monitor even edge devices. Munin follows a master/node architecture in which the master is connected to all the nodes to pull data at regular intervals. It then stores the data in RRD files and updates the graphs accordingly. Moreover, TLS-based communication can be enabled for secure master node exchange of monitoring data. The Munin master requires Perl 5.10 or newer while the nodes can run on older Perl versions while all of them can be installed on Windows or Unix-based platforms.

Nagios

Nagios[14] is an open-source application able to monitor systems, networks and infrastructure. It basically offers monitoring and alerting services for servers, switches, applications and services. It comprises of a free basic edition called Nagios Core and an enterprise edition called Nagios XI which includes among others a built-in web configuration GUI, an integrated DB, a backend API, SNMP Trap Support and Database support. In the comparison Table 1 we report on the Nagios XI. It follows a centralized approach in which a centralized server collects the monitoring data, but it is possible to create a static hierarchy of Nagios servers that mitigate the disadvantages of a centralized server. Nagios provides a way of monitoring both Cloud based resources as well as in-house infrastructures through SNMP monitoring. It serves as the basic event scheduler, event processor, and alert manager for elements that are monitored, and it is implemented as a daemon written in C and designed to run natively on Unix based systems. It has been successfully used in mission critical applications that require per second monitoring, but it requires complex configurations and compilations by experienced Linux engineers to vest its full power.

Netdata

Netdata[15] is a popular open-source software that runs across physical systems, virtual machines, applications, and IoT devices to collect real-time metrics (e.g., CPU usage, disk activity, bandwidth usage etc.). It offers fine-grained monitoring capabilities (about hosts, network, containers, databases etc.) to enable DevOps to quickly identify and troubleshoot issues and make data-driven decisions to maintain the quality of applications. Netdata consists of a lightweight daemon that is responsible for collecting and displaying information in real-time. It is written in C, Python, and JavaScript, and uses minimal resources (i.e., about 2% on a single core system). The Netdata daemon (i.e., Netdata agent) can run on any GNU / Linux kernel to monitor any system or application, and is capable of running on PCs, servers, and embedded Linux devices. The agents although they are autonomous their metrics can be aggregated by the Netdata Cloud which is a cloud-based centralized management system. Netdata is well-suited for distributed systems and includes auto-detection, event monitoring, and machine learning to provide real-time monitoring. Furthermore, it offers health alarms powered by its Advanced Alarm Notification System that pinpoints infrastructure issues and any vulnerabilities in the topology or application at hand. In addition, it is designed to be installed on anything utilizing a Linux kernel, without interrupting any of the applications running on it while it operates according to the memory requirements specified by the user, using only idle CPU cycles. It contains certain default plugins to collect key system metrics, but it is highly extensible by using a dedicated plugin API. The metrics collected are unlimited and can reach per-second granularity. One of its latest and popular plugins involve the integration to collect data from Vnstat[16] (a popular open-source console-based network traffic monitor). Netdata follows a community-first approach, with hundreds of contributors, thousands of GitHub stargazers and millions of users worldwide.

New Relic

NewRelic[17] is a proprietary SaaS solution for monitoring both the infrastructure in a traditional, hybrid or cloud environments as well as the applications running on them. It provides a serverless monitoring tool that can handle all

[13] http://munin-monitoring.org/
[14] https://www.nagios.org/
[15] https://www.netdata.cloud/
[16] https://humdi.net/vnstat/
[17] https://newrelic.com/

sorts of tasks by means of code instrumentation and support milliseconds speed with respect to the monitoring data. New Relic aims at intelligently managing complex and ever-changing cloud applications and infrastructure while it places all the data in one network monitoring dashboard following a centralised approach. Nevertheless, the insights-based queries that it supports are quite limited for enterprise scale purposes.

Openstack Telemetry service

The Telemetry[18] service of OpenStack is an open-source project which collects measurements about the utilization of the physical and virtual resources comprising deployed private clouds (i.e., Openstack environments). It also involves the persistence of this data for subsequent retrieval and analysis, while it supports triggering actions when defined thresholds are met. Telemetry comprises of three sub-projects: i) Aodh[19] which exploits monitoring metrics to issue predefined alarms; ii) Ceilometer[20] which is the main data collection service and iii) Panko[21] which is an event, metadata indexing service, acting as a REST API for Ceilometer. All these systems are mainly used for billing purposes, but they are valuable for the following up on the health of the deployed infrastructural services. Therefore, they enable the delivery of the so-called counters and events (relevant to the QoS of the hosting and networking environment) in traceable and auditable way. All of them (provided in YAML format) are easily extensible to support new projects, and agents doing data collections which are independent of the overall system can be integrated. Such monitoring data are published to various targets including data stores and message queues.

Opsview

Opsview offers commercial products that specialise in IT infrastructure and cloud monitoring software for on-premises, cloud and hybrid IT environments. It offers the Opsview Cloud and Opsview Monitor products as scalable monitoring platforms that can scale to tens of thousands hosts. The Opsview Monitor exploits the open source Ansible automation framework for deploying and managing the lifecycle of the monitoring agents. Furthermore, it offers auto-discovery features to automatically monitor new workloads as they are deployed on bare metal, virtual machines or in the cloud. Last, it provides a wide and extensible range of monitoring metrics that target at applications public, private and hybrid clouds, containers, networks and databases. The Opsview monitor collector uses a TLS-encrypted message bus (i.e., RabbitMQ) to push periodically monitoring data to the Opsview monitor orchestrator.

Prometheus

Prometheus[22] is a widely used open-source system for monitoring and alerting which is often used with Grafana[23] for monitoring data visualisation. It has an active developer and user community and it offers several useful features. It produces time series data per monitoring metric and per autonomous node, in the form of key/value pairs and supports a flexible query language (i.e., PromQL). The measurements are collected through the HTTP protocol, following a pull model while it can push time series through an intermediary gateway to allow ephemeral and batch jobs to expose their metrics. The monitoring targets are usually statically configured but there is also the option of service discovery. It is integrated with special-purpose exporters for services like HAProxy, StatsD and Graphite. Last, it allows the definition of alerting rules which generate alerts that are sent to a centralised Alertmanager. This Alertmanager is responsible for managing those alerts and sending out notifications via methods such as email, on-call notification systems, and chat platforms.

SequenceIQ

SequenceIQ[24] is an open-source tool for monitoring Hadoop clusters which has been merged since 2019 with the Cloudera Manager[25]. The SequenceIQ architecture is based on Docker containers and the ELK stack, that is, Elasticsearch[26], Logstash[27] and Kibana[28]. The use of client and server containers provides a clear separation between the Hadoop deployment and the monitoring tools. The server container implements the aggregators of monitoring data and in particular, Kibana is used for visualization and Elasticsearch for consolidation of the monitoring metrics. Through Elasticsearch the horizontal scaling and clustering of multiple monitoring components is supported [15]. The

[18] https://wiki.openstack.org/wiki/Telemetry#OpenStack_Telemetry
[19] https://github.com/openstack/aodh
[20] https://github.com/openstack/ceilometer
[21] https://github.com/openstack/panko
[22] https://prometheus.io/
[23] https://grafana.com/
[24] https://github.com/sequenceiq
[25] https://docs.cloudera.com/cdp-private-cloud-base/7.1.3/concepts/topics/cm-introduction-to-monitoring.html
[26] https://www.elastic.co
[27] http://logstash.net
[28] https://www.elastic.com/producs/kibana

client container involves the deployment of the software tools that are to be monitored and contains Logstash, collectd modules. Logstash connects to the Elasticsearch cluster as a client and stores the processed and transformed metrics data. Because of containerization one can easily add or remove components from the system without affecting the overall monitoring platform.

Shinken

Shinken[29] is an open source network monitoring software application which was proposed as a new development branch of Nagios 4. It is written in Python to watch hosts and services, gather performance data and alert users when error conditions occur and again when the conditions clear. Shinken mainly monitors network services (e.g., SNMP, SMTP, POP3, HTTP, SSH etc.) but it can also provide some basic monitoring metrics from the hosting resource (CPU load, disk usage etc.). In addition, it supports the definition of a network host hierarchy using parent hosts to detect and distinct between hosts that are down and those that are unreachable. It provides monitoring metrics on the basis of scheduled runs of host and service checks that are frequently polled and persisted either in files or relational DBs. It also supports remote monitoring through SSH or SSL encrypted tunnels. Moreover, it allows users to develop and plug in their own service checks according to their needs. A limitation of Shinken is that it requires on all hosts running a daemon the same versions of python packages and distribution package core. Last, it provides the ability to calculate KPIs based on state and performance data through basic aggregation functions and exports data to data visualization modules (e.g. PNP4Nagios[30], Graphite[31]).

Zabbix

Zabbix[32] is a popular open-source monitoring tool for networks, servers, VMs and cloud services. It uses SNMP to monitor network devices but also local or cloud-based servers while it supports multiple metrics such as bandwidth, CPU and memory utilization, device health in general as well as configuration changes. It exploits the monitoring data to issue alerts with the ability to run local scripts in response to alerts triggered by monitoring events. The monitoring tool is capable of monitoring thousands of servers, VMs, network or IoT devices, and other resources. Zabbix is open source and employs diverse metric collection methods with techniques such as agentless monitoring, calculation and aggregation, and end-user web monitoring. Furthermore, it provides root cause analysis to pinpoint vulnerabilities. Zabbix monitoring configuration can be done using XML based templates which contain elements to monitor. Last, it supports agentless monitoring via SNMP, TCP and ICMP checks, as well as over IPMI, JMX, SSH, Telnet and using custom parameters. Zabbix supports a variety of near-real-time notification mechanisms, including XMPP.

2.1.2 Monitoring Tools Comparison

We provide a brief overview of 20 well-known monitoring tools and compare them by listing the following information in Table 1:

- License - open-source or commercial
- Monitoring Level – at which level the monitoring is focused on: host/application/network/database/big data framework. These levels were recognised as the superset of available monitoring types that we found in this analysis.
- Processing – which refers to the kind of processing on collected monitoring data supported: basic (i.e., statistical aggregation capabilities, e.g., maximum, minimum, average)/define thresholds/cep (complex event processing)/root cause analysis/define actions(e.g. alerts))/forecasting.
- Output Types – which refers to how the monitoring data can be collected (either pushed by a monitoring agent to a monitoring server or pulled from monitoring agents by a specific monitoring server or monitoring stack) and whether on not they are persisted in files, databases and or as time-series.
- Distribution – refers to the monitoring topology used for aggregating monitoring data, i.e., centralized when a unique monitor server is used, statically distributed where a number of statically defined peers are considered for a processing monitoring data coming from neighbour monitoring nodes and dynamically distributed these peers can change at run-time according to the reconfigurations of the distributed application or the changes in the monitoring topology.
- Number of metrics supported – mentioned as low/medium/high depending on how monitoring metrics are provided out-of-the-box

[29] http://www.shinken-monitoring.org/
[30] https://docs.pnp4nagios.org/
[31] http://graphiteapp.org/
[32] https://www.zabbix.com/

- Configuration – which indicates whether or not it is easily allowed to change measurement rate of certain monitoring parameter and/or the output periodicity in which monitoring values are transmitted, persisted and processed. The alternative indications are: Low (static output/measurement rate), Medium (either output or measurement rate are configured dynamically), High (both can be configured at run-time)
- Custom metrics supported - a Boolean indication on whether or not custom monitoring probes can be accommodated (especially useful for appending application-related metrics)
- Footprint – which refers to how lightweight are the monitoring agents used (i.e., Low/Medium/High – where low indicates the possible installation on resource-constrained devices, e.g., Raspberry Pi)
- Hosting Provider Dependent – which refers to whether or not the monitoring agents introduced can be used on only one specific cloud or not.

The outcome of this comparison is presented in Table 1.

Table 1 Monitoring Tools Comparative Analysis

Monitoring Tool	License	Monitoring Level	Processing	Output Types	Distribution	Number of metrics supported	Configuration	Custom metrics supported	Footprint	Hosting Provider Dependent
Amazon CloudWatch	Commercial	host/database/application(lambda functions)	basic/alerts	pushed-based/persisted (DB)	centralized	medium	medium	yes	medium	yes
Apache Chukwa	Apache v2.0	big data framework	basic	pull-based/persisted (file)	centralized	low	low	yes	medium	no (but big data framework specific)
Azure Cloud Monitoring	Commercial	host/application/database	basic/alerts	pull-based/persisted (file/DB)	centralized	high	medium	yes	medium	yes
Cacti	GPL2.1	network/partially host	alerts	pull-based/persisted (DB)	statically distributed	low	medium	yes	low/ medium	no
Collectd	GPL2.0	host/network/extensible to handle big data frameworks monitoring	no	push-based/ persisted (file)	centralized	medium	low	yes	low	no
Datadog	Commercial	host/network/application/ database	advanced/root cause analysis	push-based/persisted (DB)/time-series	centralized	high	high	yes	medium	no
Dynatrace	Commercial	host/application/network/ database	alerts/root cause analysis/ forecasting	pull-based/persisted (file/DB)/time-series	distributed	high	high	yes	medium	no
Google Cloud Monitoring	Commercial	host/application/network/ database	basic/alerts	push-based/persisted (file/DB)	centralized	high	medium	yes	medium	yes (but is offers a collectd-based agent for multiclouds)
Hadoop Performance Monitoring	Apache v2.0	big data framework	basic	pull-based/persisted (DB)	statically distributed	low	low	yes	medium	no (but big data framework specific)
Icinga	GPL2.0	host/network	alerts	push-based/persisted (DB)	dynamically distributed	high	medium	yes	medium	no
LogicMonitor	Commercial	host/application/network/ database	alerts/forecasting	pull-based/persisted (DB)	centralized	high	high	yes	medium	no
Munin	GPL2.0	host/network	alerts	pull-based/persisted (file)	Centralized	high	medium	yes	low	no
Nagios	GPL2.0 enterprise edition	host/network	basic/alerts	pull-based/persisted (DB)	centralized/statically distributed	high	medium	no	medium	no

Netdata	GPL3.0	host/network/database - highly extensible to support application, big data frameworks	basic/define thresholds/alerts	pull-based/persisted (file/DB)/time-series	centralized	high	high	yes	low	no
New Relic	Commercial	host/network/application	advanced/root cause analysis	pull-based/persisted (DB)/time-series	centralized	high	high	yes	medium	no
Openstack Telemetry	Apache v2.0	host/network	alerts	push-based/persisted (DB)	centralized	medium	low	yes	medium	yes
Opsview	Commercial	host/application//network/ database	alerts	push-based/persisted (file/DB)/time-series	statically distributed	high	high	yes	medium	no
Prometheus	Apache v2.0	host/network/application	alerts	pull-based/push-base/persisted (DB)/time-series	centralized/ statically distributed	high	high	yes	low	no
SequenceIQ	Apache v2.0	big data framework	basic	push-based/persisted (DB)	dynamically distributed	medium	low	yes	medium	no (but big data framework specific)
Shinken	AGPL3.0	network/partially host	basic	pull-based/persisted (file/DB)	statically distributed	medium	medium	yes	medium	no
Zabbix	GPL2.0	host/network	basic/alerts	pull-based/persisted (DB)	statically distributed	medium	medium	no	low	no

2.1.3 Monitoring Tools Selection in MORPHEMIC

Based on the above mentioned comparative analysis of prominent monitoring tools, we examined the main aspects of several commercial and open-source projects that are widely used. We found out that the majority of these tools follow a centralised approach while only two of them (i.e., Icinga and SequenceIQ) provide the appropriate means to support a real dynamically distributed monitoring topology, a fact that denotes the importance of our federated EMS approach. Also, the majority of these tools focus mainly on the aggregation of monitoring data to visualise them and let the DevOps to decide on mitigation actions based on the current status of the application. Only a few of them support advance processing capabilities and root cause analysis (i.e., Datadog, Dynatrace, New Relic) while only one of them supports proactive mitigation actions suggestions (Dynatrace). Last, the majority of the tools focus on hosting resource and network monitoring level indicating the diversity required to cope with different application types monitoring.

As mentioned above we conducted this comparative analysis in order to recognise the most promising software tool with respect to its monitoring probes in order to enhance our federated event management system. Based on our analysis we selected Netdata, a popular open-source monitoring solution that introduces lightweight monitoring agents that are flexible enough to be integrated with the event driven architecture (EDA) of the EMS system. Specifically, Netadata agents require just 2% of a single core system to collect a vast amount of different monitoring parameters which are highly and dynamically configurable in terms of measurement rate and periodicity of collection. Its very low footprint and cloud/resource provider independence along with the straightforward way to introduce custom application-level probes constitute Netdata an appropriate monitoring measurement solution for EMS. Last, it consists of a wide community of github contributors which result in constant improvements and addition of new monitoring capabilities.

We note that the outcome of this analysis was successfully validated with real experimentation and testing of the Netdata capabilities over heterogeneous resources such as public and private cloud resources, resource constrained devices (i.e., Raspberry Pi) and installation on accelerators such as FPGAs. This work was concluded with a successful integration of Netadata as multiple, extensible and dynamically configurable monitoring probes to EMS.

2.2 Analysis of Time-Series DBs

The Morphemic platform, a multi-cloud management system, proposes an innovative approach for achieving reactive and proactive adaptation of cloud applications. This adaptation refers to scaling and architecture modification. The management or the configuration under which cloud applications run is ensured using information related to the application performance provided in the CAMEL model and metrics about the current application performance aggregated through EMS.

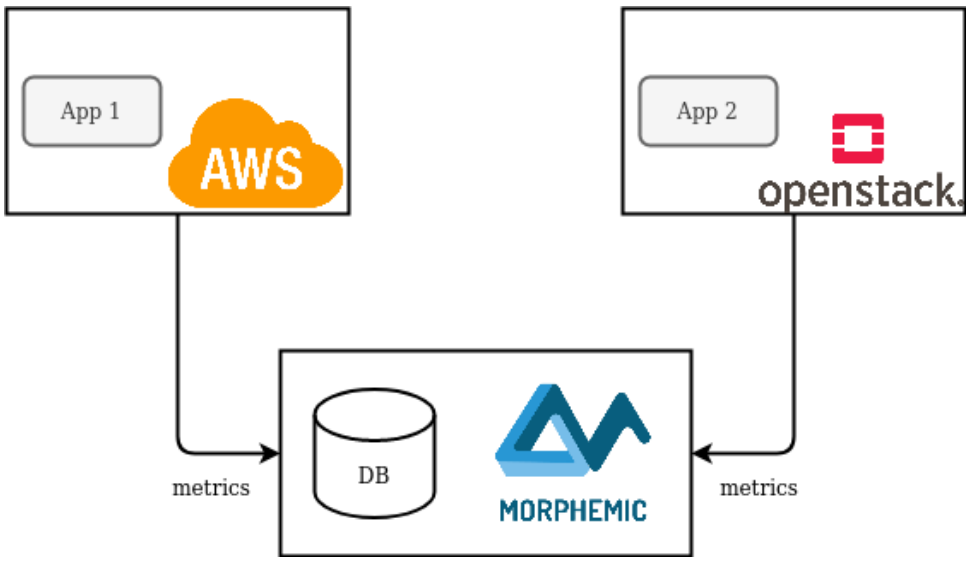

Figure 1. Persisting monitoring metrics in Morphemic

Those metrics could be produced by the application itself or by the cloud environment where cloud application components are running. Metrics must be collected, stored then analysed to discover correlations among metrics, topologies and application performance and constitute the learning set for predicting trends or imminent failures (to be used for proactively adapting multicloud applications). Therefore, Morphemic must implement a system capable of storing metrics. For ensuring the compliance of the real application performance with the desired performance, a

periodic comparison of both performance aspects must be performed. Measurement must be taken contently for different components interacting with the deployed application. This implies a significant number of metrics to be transferred from each cloud provider to the Morphemic platform. Considering the collection interval of those metric measurements it is important for Morphemic platform to adopt an adequate database for metrics (time-series database) which can support high write speeds.
Morphemic platform has been designed to minimize the computer resources consumption where it is running, thus all components must satisfy a performance requirement in terms of consumed computing resources. Therefore, performance (e.g., memory consumption, CPU usage etc.) of components deployed represents an important criterion for optimizing resources consumed by Morphemic.
The stored metrics must be accessed for constructing the datasets; a very important Morphemic feature for enabling time-series forecasting module and cloud application performance model mechanism. Thus, the storage system must present low latency and enable multi-metric retrieve methods. The latter refers to being able to query metrics stored based on several metric fields (e.g., name, application, component name, time range, etc).
Many time-series databases are currently available and can be used in the Morphemic platform. We analyse some of them according to their write speed, their performance in terms of memory and CPU consumption along with the latency of read capability.
Time-series databases comparison for a cloud environment, focus essentially on performance criteria however, the performance indicators must be selected according to cloud environment requirement. It does not matter how well a reference point database works, if it does not satisfy Morphemic needs. We will focus on the write performance, deployment, extensibility, integration and data model. Properties such us time series compression, scability won't be mandatory for the comparison. With that in mind, we start by comparing InfluxDB[33], OpenTSDB[34], TimescaleDB[35] and Graphite[36].

InfluxDB
InfluxDB is a time-series database (TSDB), as an open-source software available under the MIT licence. It was used in the Melodic project as an fast, highly-available storage for the retrieval of time-series data. InfluxDB has no external dependencies and its deployment is simple and fast. InfluxDB can be executed as a containerized application. A? SQL-like language is supported, allowing the construction of complex queries where data points can be selected and filtered using different fields. Each data point consists of the name of metric, the corresponding value, timestamp and other related information in the form of key values. When grouped by a set of key-value pairs called sets of tags, they define a sequence. Finally, the strings are grouped by a string identifier to form a measure. Values can be 64-bit integers, 64-bit moving points, strings, and Booleans. Points are indexed according to time and their labels. The conservation policies are defined in a measurement which control the way data is subdivided and deleted. Continuous queries are performed periodically, storing the results as a target metric. Different InfluxDB clients have been developed in most of programming languages allowing interaction, integration and communication with other business logics.

OpenTSDB
OpenTSDB, Open Time-series Database is a time-series database built on top of HBase which is a non-relational distributed database. Therefore, OpenTSDB is scalable and has a very good performance in a distributed environment. It also has a good writing performance. OpenTSDB is composed of three main components: tCollector, Time-series Daemon (TSD), and HBase. One instance of tCollector is deployed on each server. It is responsible to periodically pull metrics' data from processes running on the server and the operating system. TSDs receive data from the tCollectors and push data to the HBase backend storage system. In other words, the Time-series Daemon is responsible for interacting with HBase and store/retrieve data. Upon receiving queries, TSD scans HBase and retrieves relevant data. All communications are done via TSD RPC and Hadoop RPC, therefore all components are stateless which allows scale operations. There can be as many TSDs as needed to handle the workload as the system scales.
OpenTSDB provides two official query interfaces: HTTP/REST API and Telnet style command line API. However, there are various open-source front-end clients that encapsulate the official APIs.

[33] https://www.influxdata.com/
[34] http://opentsdb.net/
[35] https://www.timescale.com/
[36] https://graphiteapp.org

TimescaleDB
TimescaleDB is also an open-sourced time-series database which is based on SQL premises. It has been designed to make SQL scalable for time-series data. TimescaleDB is built on top of PostgreSQL and packaged as a PostgreSQL extension, therefore providing automatic partitioning across time and space as well as full SQL support. TimescaleDB provides different clients written in many programming languages for establishing a TimescaleDB or interacting with it. Since it is based on SQL premises, time-series data must be structured according to the collection or the table which will store them. However, the fact that it is based on PostgreSQL, thus using SQL language for data manipulation, it has the advantages of being able to handle huge amounts of data and it presents very good scaling performance in addition to the fact that it uses a well-known query language .

Graphite
Graphite is a time-series database available on an Apache licence. Graphite was developed by Orbitz Worldwide, Inc. and released as open-source software in 2008.
Graphite is designed to handle numeric data. For example, Graphite would have a good performance when storing a response time (numerical value) which changes over time. Graphite can handle a big number of metrics coming from several servers. Several Graphite clients have been developed for interacting with Graphite. Metrics sent to Graphite should contain the metric name, value and the timestamp. Except for storing metrics, Graphite can also render graphs of data on demand.
The tool has three main components:

- Carbon - a Twisted daemon that listens to time-series data,
- Whisper - a simple database library for storing time-series data (similar in design to RRD),
- Graphite webapp - A web application that renders graphs on-demand using Cairo library.

Analysis Results
The analysis of InfluxDB, Graphite, OpenTSDB and TimescaleDB provides us enough information to select the time-series database which will cover the need of the platform regarding the storage and retrieval of time-series data. In addition, we consider reusability aspects taking into account the Melodic platform which is used as the basis for Morphemic. Therefore, the best candidate is InfluxDB since its performance in terms of writing and retrieving data was valuable. InfluxDB is already integrated to Melodic, therefore this choice encourages the reusability of components, which is an important gain in terms of resources.

3 Monitoring Probes and Persistence Deployment and Configuration

3.1 Deploying and Configuring Monitoring Probes

Monitoring probes are used to provide measurements of various infrastructure or application-specific aspects. As mentioned above Netdata will be used for this purpose. This data is used in Morphemic reasoning process to evaluate the operational state of each application node or the condition of the (multi-cloud) application in total. For this purpose, monitoring probes are deployed in each application node and the collected measurements are passed to the local EMS client for processing and propagation. EMS undertakes the processing and propagation of the monitoring probes outputs to detect situations that should trigger new reasoning and reconfiguration cycles. The way that this is performed is explained in detail in section 4.
Different multi-cloud applications require different variables to be monitored and used, with regard to triggering application reconfiguration. These variables, along with the probes that capture their values, are declared in the CAMEL models of the applications. During application deployment, the translator component of EMS server analyses CAMEL model and extracts the needed variables and probes. It moreover extracts the corresponding value collection rates. In Melodic this information was passed to the Adapter component, which in turn instructed Executionware to install the needed probes to application VMs. In contrast, in Morphemic, the EMS server uses this information to install the needed probes to application VMs on its own. For VM or node-related information the well-known Netdata monitoring suite is used. For application-specific variables, the CAMEL model must specify how the needed probe can be installed.

3.2 Deploying and Configuring Time-Series DB

The time-series DB will be deployed in Morphemic, as part of the Persistent Storage component to persist metrics, however, this deployment should be part of a module that delivers all functionalities according to Section 2.2. The

Morphemic platform requires a mechanism for storing and retrieving metrics. Metrics exposed by applications and cloud infrastructure are gathered by EMS then pushed to the Morphemic system. Based on the frequency of metric collection, a stream connection is necessary for enabling a permanent link between the EMS client deployed on cloud providers and the Morphemic platform. At the Morphemic side, a receiver or consumer, a component connected to the streaming system and subscribe to a dedicated topic will decode metrics and convert them to the right format according to the time-series database technology. As stated in Section 2.2 about the time-series database, metrics received from EMS will be translated into the InfuxDB format. The table below describes the metric format as mentioned in the InfluxDB protocol specification.

Table 2 Metric format in InfluxDB

Field	Description	Required	Usage on Morphemic
Measurement	Name of the measurement. InfluxDB accepts a string type for this field	True	This field contains the name of the application for optimizing metrics information retrieval
Timestamp	This field repesents the data point collection date time. If not specified, InfluxDB will set the insertion time.	False	This field will not be filled by Morphemic for synchronization reason.
Fields	Key/value	True	This field will be used to specify metric name and its value. Many metrics from the same component can be specified if the component exposes more than one metric.
Tags	Other infomation related to the measurement as key/value	False	Tags will be used as metadata for a measurement. Information such as cloud provider where the component is deployed or the configuration of the component (container, VM, big data job etc.)

JSON data format will be used for transferring metrics from cloud providers to Morphemic. Example: given the application FCR [36] exposing the response time (i.e., response_time), the number of users (i.e, number_users) and running with 2 CPU and having 2 instances. The data point inserted into InfluxDB could look like:

{"measurement": "fcr", "field":{"response_time": 45, "users": Y, "cpu_usage": 20}, "tags":{"instance": 1}} or {"measurement": "fcr", "field":{"response_time": 30, "users": 6, "cpu": 2 }, "tags":{"instance": 2,"cpu_core":2}}

Application's specific metrics having the same collection time can be inserted into the same data point. This approach simplifies the dataset making process. However, pre-processing is required for grouping those metrics.
The events (metrics) pushed by the EMS agent deployed on the cloud have the following format:

{"metricName": "name", "metricValue": value, "timestamp": time, "application": "applicationName", "level": 1…}

As explained above, the time-series DB will be deployed as part of the persistent storage for which the initial architecture is shown in Figure 2.

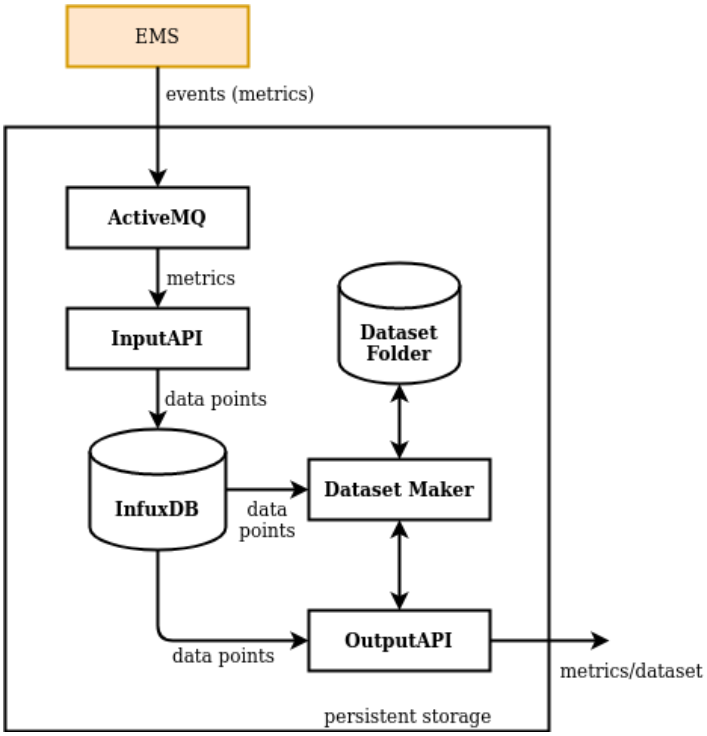

Figure 2. Morphemic's Persistent Storage

In order to deliver all functionalities required by Morphemic, persistent storage is composed of the following components:

- **ActiveMQ broker**: A local broker is needed for receiving all events pushed from different clouds where applications managed by Morphemic are deployed. Events will be pushed into a specific topic for being consumed locally.
- **InputAPI**: The InputAPI is the main consumer of events pushed into ActiveMQ broker. The InputAPI has been designed for being able to multiply channels for increasing the number of events (messages) received. The functional architecture of this component is depicted in Figure 3:

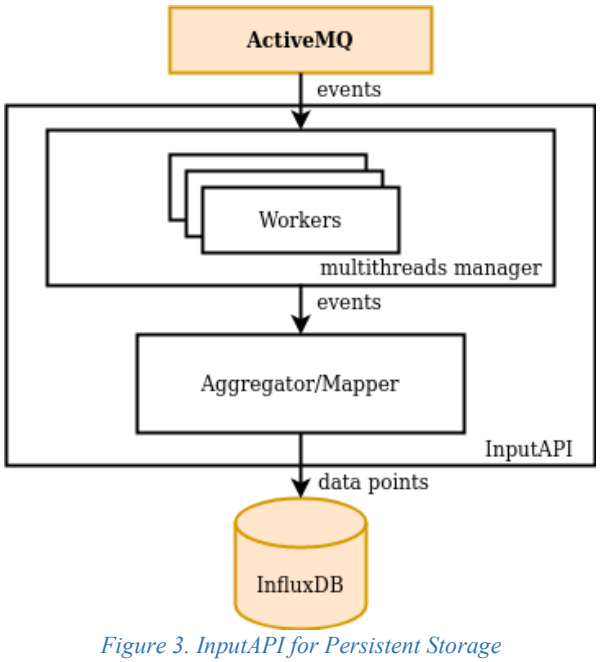

Figure 3. InputAPI for Persistent Storage

- The **aggregator/mapper** firstly group events based on the application field and timestamp, then data points will be constructed before being sent to InfluxDB.

- **Dataset maker**: This component creates datasets given the name of an application. Specifically, it provides a library for querying the Time-Series DB and pre-processing the data in order to constitute them usable as training sets by the implemented forecasting algorithms.
- **OutputAPI**: The outputAPI gives any other component of Morphemic access to metrics stored in InfluxDB and also allows the creation of dataset through a REST call.

4 Self-Healing Federated Event Processing Management System

4.1 EMS in MELODIC

EMS was introduced in the H2020 Melodic project [16] as a distributed application monitoring system, which is used by Melodic Upperware for monitoring the operation of the cross-cloud applications it deploys. It is able to collect, process and deliver, monitoring information pertaining to a distributed, cross-cloud application, according to CAMEL model specifications, especially considering the defined SLOs. The aggregated monitoring information are used by Upperware to trigger reactively the reasoning process and issue decisions on reconfigurations when and if needed. The big advantage of the EMS approach is its decentralized nature which is considered as ideal for multiclouds applications, since it provides a hierarchical filtering of the monitoring information avoiding bottlenecks and excessive use of network bandwidth.

EMS undertakes the task of deploying a network of agents for collecting monitoring information from the monitoring probes as events, process them using distributed and complex event processing techniques, and forward the results to Upperware (e.g. Metasolver). A CAMEL model specifies the needed monitoring information and the kind of processing required, as these have been defined through SLOs. We note that both the installation of monitoring probes and the deployment of EMS agents was the responsibility of Executionware under the orchestration of the Melodic workflow. In Morphemic this is significantly improved, constituting EMS the responsible entity that manages all the monitoring aspects. EMS is a distributed application monitoring system that comprises of a server integrated in Melodic Upperware, named Event Processing Manager (EPM), and several clients, named Event Processing Agents (EPAs). EPM and EPAs formulate a network of nodes for distributed event processing, called Event Processing Network (EPN). This network is orchestrated and controlled by EPM. Specifically, the following list describes the main EMS functionalities:

- Analyse the CAMEL model of a cross-cloud application in order to extract the required (by other Melodic-platform components) monitoring information along with the processing needed.
- Deploy (through Executionware) EMS clients, which are called Event Processing Agents (EPAs) to each distributed application node that hosts an application component (to be monitored).
- Configure each EPA to collect (from sensors) and forward the needed events, and also apply the required complex event processing rules.
- Provide the required information (specified in the CAMEL model), either by updating the application CP model, by publishing events (any interested party may subscribe to receive them), or by requesting Melodic-platform to reconfigure the distributed application (e.g., when certain SLOs are violated).

The fine-grained functionalities of EMS are provided in [16], but for completion reasons we provide the initial architectural details of the system. First, we provide a high-level architectural view EPM through the UML component diagram depicted in Figure 4.

EPM comprises the following sub-components:

- **Translator:** provides a two-step process involving the analysis of the CAMEL model to produce a multi-root Directed Acyclic Graph and also the Generation of EPL rules and other related information.
- **Client install component:** provides the necessary instructions on how to install an EPA to the application VM defined in the application deployment model.
- **Baguette Server:** is responsible for the deployment and management of the Event Processing Network. Specifically, it designates EPAs installed in each application VM, to the appropriate grouping, by sending the corresponding configuration. It also collects VM identification information sent from EPAs. The Baguette server encapsulates an SSH server used to accept incoming connections from EPAs. These connections are used to send configurations or other commands to EPAs.
- **Control Service**: Coordinates and oversees the functioning of EPM. It also interacts with the Upperware control process through the REST API and furthermore offers a few EPM management and debugging functions (as REST endpoints as well).
- **Web console component:** Provides a dashboard for monitoring the functioning of the local event broker.

- **Broker-CEP Service:** encapsulates an event broker instance and a CEP engine instance, hence Broker-CEP provides event brokerage and complex event processing capabilities. The Consumer sub-component depicted inside Broker-CEP is used to forward the event broker messages into the CEP engine in order to be processed.

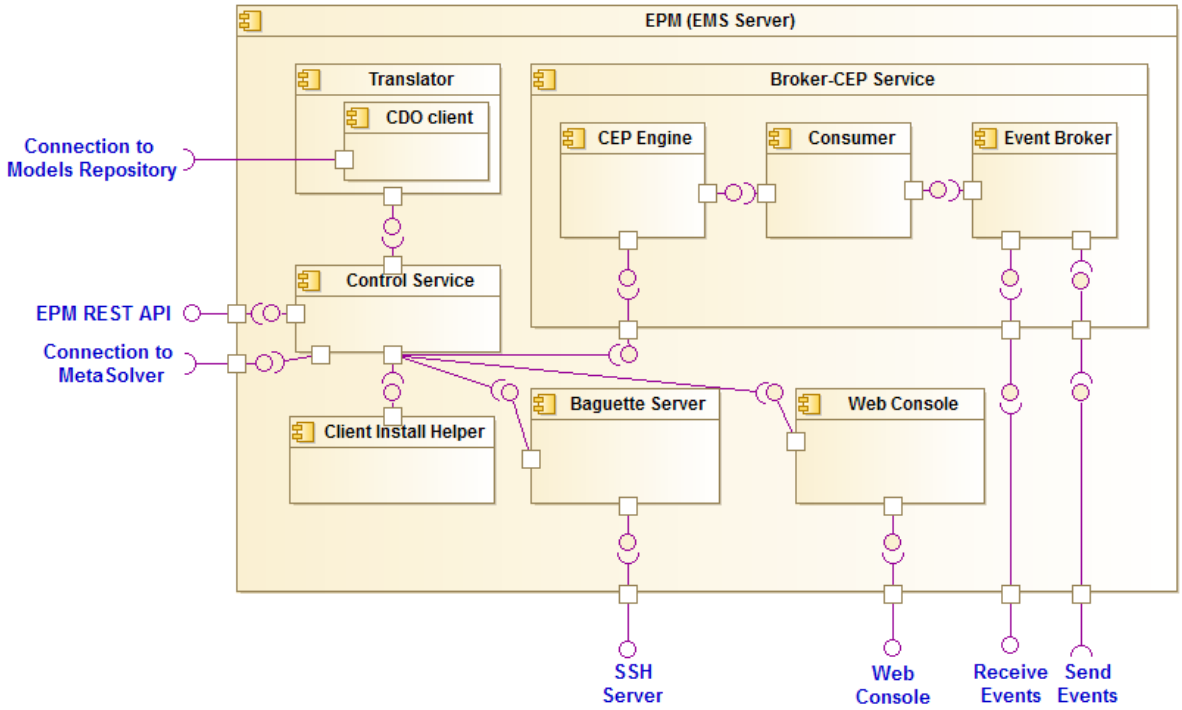

Figure 4. EPM (EMS server) Component diagram

In a similar manner we repeat the high-level overview of the EPA architecture through the following UML component diagram, for better understanding of the system on which we build on for developing the Morphemic federated and resilient event management system. As it is shown in Figure 5 the EPA comprises the following sub-components:

- **SSH Client:** provides secure communication with the Baguette server through an EPM connection interface.
- **Executor component:** Configures and starts the Broker-CEP Service based on the instructions by the Baguette Server.
- **Broker-CEP service:** provides the local Complex Event Processing engine (i.e., Esper[37]) along with the local Event Broker (i.e., ActiveMQ[38]) that collects and propagates data messages to other Virtual Machines components in the distributed hierarchy of Virtual Machines in our cloud environment. This client can undertake the role of per instance, per host, per zone, per region or per cloud grouping.

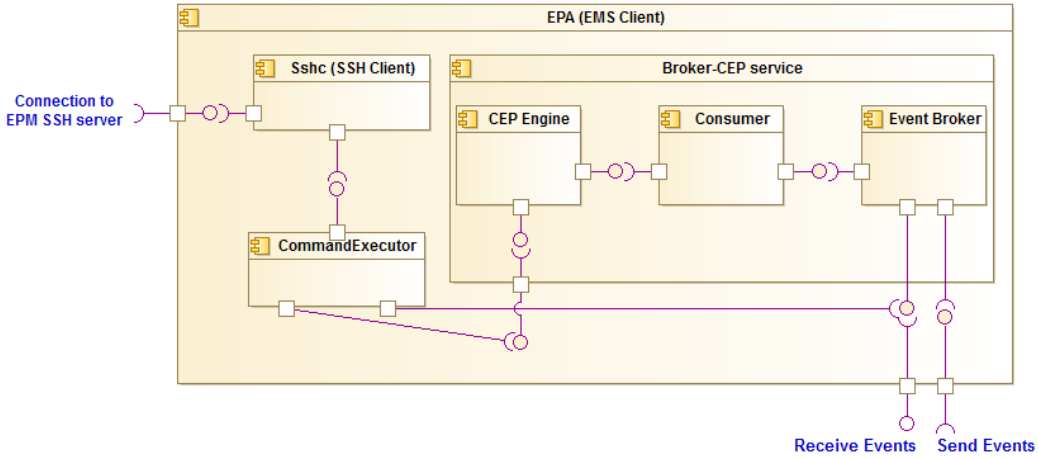

Figure 5. EPA Component diagram

[37] https://github.com/espertechinc/esper
[38] https://activemq.apache.org/

4.2 EMS with Self-Healing Capabilities

Several self-healing features have been introduced to EMS in the context of Morphemic as means to guarantee the resilience of the monitoring service across dispersed cloud and edge resources. As hosting resources move closer to the edge of network the possibility of encountering node failures or connectivity issues rise significantly. Therefore, it is important to properly safeguard Morphemic's EMS by introducing the ability for the monitoring system to self-heal. These new features are based on a new, decentralized EMS topology orchestration (i.e., federated EMS), which in turn is based on the new EMS client clustering approach (we note that a brief analysis of the related work with respect to clustering techniques is provided in section 4.4). Furthermore, EMS server and EMS client architectures have been updated to support the use of plugins, which can introduce new features to EMS (including additional self-healing capabilities). These plugins enable the seamless integration with components that introduce new capabilities (e.g., resource Health Checker) as it is described in section 4.3.

4.2.1 Federated EMS Approach

The initial version of the EMS server, as it was implemented in the context of the Melodic project, controls most aspects of EMS clients' operation and their interconnections, being responsible to send configurations and assign certain monitoring-related roles to EMS clients. EMS clients are running at the nodes that host component instances of the multi-cloud application (typically VMs). The EMS server is also able to select which EMS clients will assume the role of the measurement aggregators (also called *brokers*) for a specific scope[39] (host, zone, region, or cloud), and configure the rest of the clients in the same scope to forward their events (measurements or calculated metrics) to the designated aggregator.

This centralized control of the EMS network is easier to implement and comprehend but it comes with certain limitations, particularly if the EMS server (which exercises the control) crashes or malfunctions. In such a case it is impossible to reassign roles to EMS clients and subsequently, if an aggregator fails, no new aggregator will be selected, resulting in loosing events and monitoring functionality. Additionally, EMS in Melodic could not cope with EMS client crashes, which again result in loosing events from certain scopes.

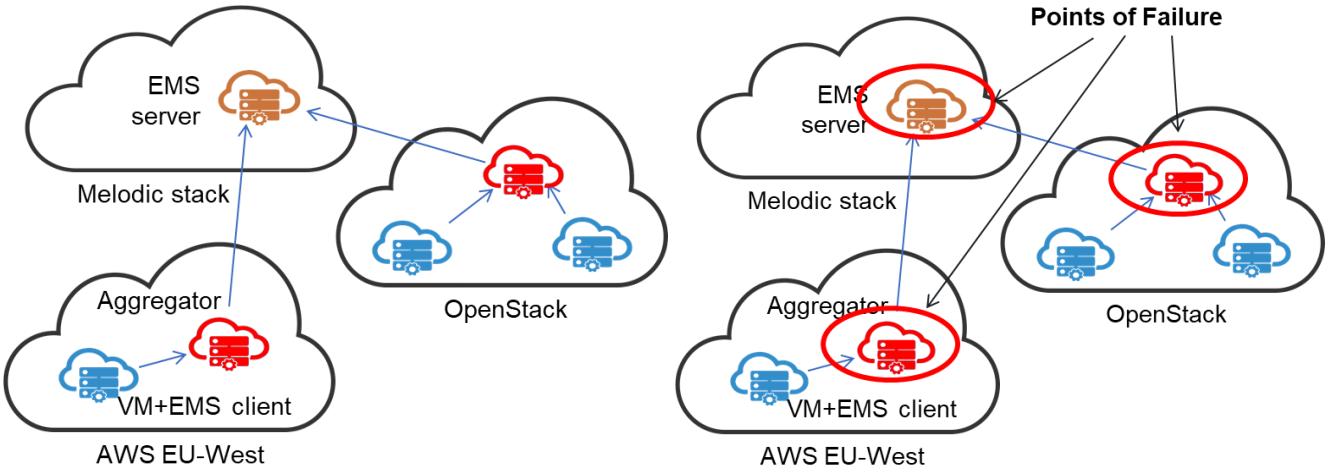

Figure 6. EMS topology example and Points of failure in Melodic

Figure 6 provides an example of a multi-cloud application deployment, where each application hosting node (i.e., VM) gets an EMS client collocated with the application component software. Arrows indicate the event flow between EMS nodes (EMS clients, aggregators and EMS server). In the second figure the possible points of failure with wide impact in EMS topology are also indicated, i.e., EMS server and aggregators.

In Morphemic we envisage a decentralized approach where certain functions of the EMS server will be shifted to EMS clients. Specifically, EMS clients will be responsible for tracking the availability of other close by EMS clients. To this end, neighbour EMS clients will automatically formulate a *local cluster*. It is possible a single multi-cloud application encompasses several local clusters, for instance one at each cloud service provider where application VMs are deployed. EMS clients participating in the same local cluster are called *peers*. Apart from their monitoring functionality they are also acting as cluster nodes by checking the availability of their peers and participating in the cluster's distributed protocols under an aggregator (as it is explained in Section 4.2.4). We also note that the extended

[39] We note that the scope may refer to an availability zone or a region or a whole cloud service provider, where some of the multi-cloud application VMs, along with their EMS clients, can be deployed. The scope depends on (a) the features offered by cloud service providers with regard to providing the zone or region of a deployed VM, (b) the application's CAMEL model, and (c) the configuration of EMS.

EMS is able to deploy Netdata agents collaborating with EMS clients for aggregating raw monitoring metrics (as described in section 3.1). Next, Figure 7 provides a high-level view of a new EMS client deployment and furthermore depicts joining to the local cluster and participating in the availability checking protocol.

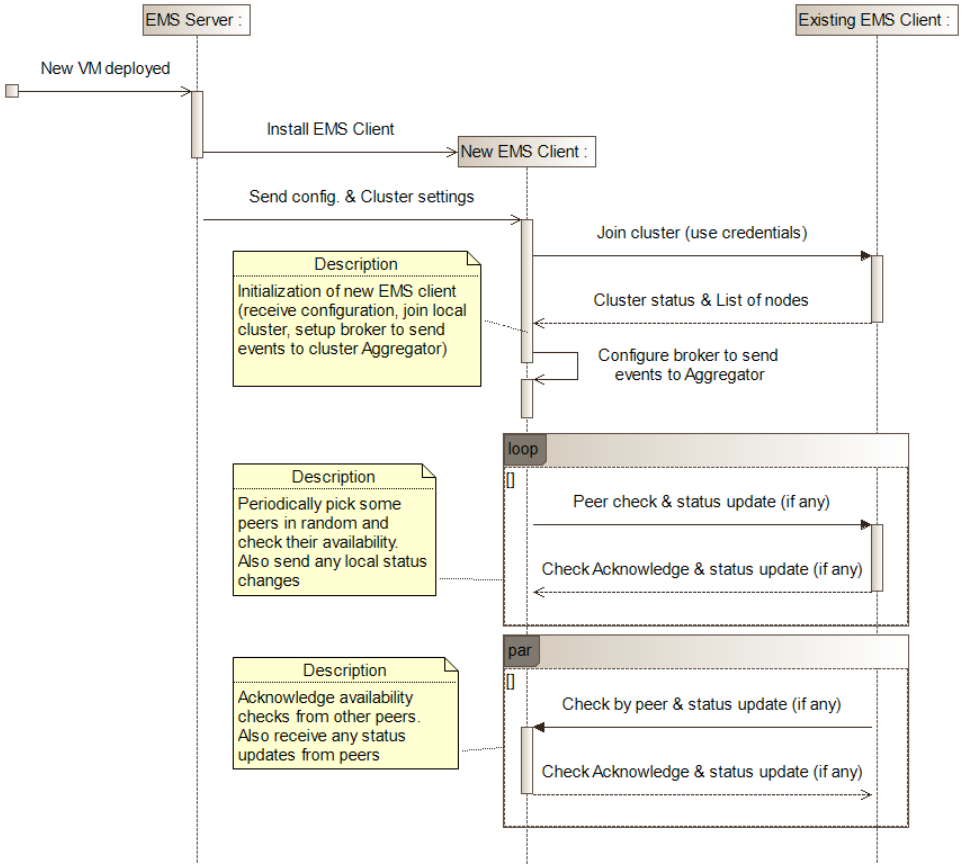

Figure 7. High-level process flow of a new EMS Client joining its local cluster

The enhanced EMS server will be able to coordinate the formulation of local clusters by sending appropriate information to enhanced EMS clients about their peers and also facilitating the secure exchange of credentials among them. After the cluster formulation, the EMS server will just receive notifications about cluster nodes' status as well as about any aggregator changes (please refer to Section 4.2.4 for more information of aggregator selection).
EMS clients in each local cluster will continuously check the availability of their peers, using a distributed availability checking (or membership) protocol. This protocol must be immune to short-term disruptions of operation or connectivity of peers, but it must be able to detect long-term or permanent disruptions and remove problematic peers from the cluster. Modern cluster management software frameworks provide such features. If a problematic peer recovers it can still (re-)join the cluster. Moreover, the availability checking protocol can trigger special actions that will attempt the recovery or repair of problematic peers, thus offering an extra level of self-healing capability to the local cluster. Such actions might involve connecting to a problematic peer using alternative communication channels (for example SSH) or restarting a certain EMS client.

In the envisaged decentralized approach, EMS clients will also be responsible for selecting an aggregator for their scope, in case the current one fails. According to EMS network topology architecture[40], in each scope, one of the EMS clients can act as aggregator by receiving the events of its peers, probably processing them to compute scope-wide metrics (e.g., averaging a metric per availability zone), and forwarding them to the next hierarchical level of EMS

[40] EMS topology architecture provides for a multilevel hierarchy, where lower-level nodes forward their events (conveying either raw measurements or computed values) to the next level, where further event processing can take place, possibly combining the values of events forwarded from several underlying nodes. In practice, only two-level and three-level topologies have been tried in Melodic. The former topology requires that EMS clients (Level 1) will forward their events directly to EMS server (second level). The latter topology provides for an intermediary level per scope/group (Level 2), where EMS clients (Level 1) will forward their events. Level 2 comprises those EMS clients acting as Aggregators and receiving the events of the Level 1 clients running in the same scope/group. Subsequently, the aggregators process and forward the events to EMS server (Level 3). For notation uniformity, EMS server is always considered Level 3 in Melodic, even when a two-level topology is used.

topology. Constituting EMS clients responsible for selecting the aggregator of their scope eliminates one possible point of EMS failure that existed in EMS initial version. Specifically, since the EMS server is no longer required to select and appoint EMS clients for aggregators, an EMS server failure will not impact the replacement of a failing aggregator. Its peers will automatically select another client to become the aggregator in their scope.
This decentralized, cluster-based approach will allow the EMS monitoring network to remain operational and retain its structure, even if a number of clients become unavailable for some time, by being able to self-heal failures like losing aggregators. This will also allow faster response in case of an EMS client failure since its peers will be notified earlier than EMS server and will be able to take immediate action. Finally, the work of EMS server with regard to managing EMS network is distributed to EMS clients. In short, the new features the envisaged cluster-based, decentralized EMS will include:

- maintaining an *alive nodes* list (per cluster) that is immune to short-term connectivity disruptions,
- detecting failures (temporary or permanent) of EMS nodes and triggering actions, and
- selecting cluster aggregators faster and in a more fault-tolerant manner than before.

The EMS server, in the envisaged decentralized approach, will still be responsible for providing configuration to the new EMS clients, and furthermore will provide a list of adjacent EMS clients (nodes), residing in the same scope, along with the needed security credentials. New EMS clients will use the adjacent nodes list and credentials, in order to join to the cluster (or create it if they are the first in their scopes).

4.2.2 EMS with self-healing capabilities

Self-healing is the capability of a system to repair itself and recover from problematic situations that might occur. In EMS, self-healing means the capability to replace monitoring functionality lost due to an EMS node failure or and the capability of repairing the malfunctioning nodes. A prerequisite to achieve such self-healing capabilities in the Morphemic monitoring mechanism is the federated EMS.
We need to distinguish between three different cases of self-healing, depending on the type of node that failed.

- ***Aggregator loss healing***. It refers to selecting a new aggregator in a cluster, in case the previous one unexpectedly fails. This self-healing feature is implemented as part of the federated EMS aggregator selection (see Section 4.2.4). Cluster nodes will store their measurements locally and forward them to the new aggregator, when it becomes available.
- ***Simple node loss healing***. When any other cluster node (except from aggregator) fails, the aggregator will try to repair it by following this protocol:
  - Try to contact to the lost node's EMS client's ActiveMQ broker,
  - Try to connect to the corresponding VM (using SSH)

  In this case the self-healing scenarios involve the following:
  - If the VM is unresponsive, the EMS server will cancel its credentials and remove it from the EMS topology and other cluster nodes will ignore it. If at a later time, VM (and EMS client) recovers, it will try to connect to the EMS server from the top (as it happens during application deployment).
  - If the VM is responsive, the aggregator connects through SSH and restarts EMS client, which will connect to the EMS server from the top (as it happens during application deployment).
- ***EMS server loss healing***. It refers to restoring EMS server malfunctioning or replacing it if it completely fails. Nodes forwarding their measurements to (malfunctioning or lost) the EMS server will store their measurements locally and forward them to the new EMS server, when it becomes available.
  In case the EMS server container crashes, the Docker will try to restart it, according to Morphemic's Docker Compose specification. Additionally, an external EMS server health check service can be used to run health checks about the EMS server's responsiveness (broker, REST API) and sanity, and force the EMS server restart if needed. Optionally, there can be a stand-by EMS server instance that will be activated (by health check service) in case the main EMS server instance fails. In this case, the replacement EMS server instance will reconfigure all EMS clients to forward their messages to it.

The EMS client software architecture will be changed in order to allow the use of (external) self-healing plugins. These plugins are effectively extending EMS client capabilities by introducing additional health checks and recovery procedures. Such plugins can trigger aggregator selection or change node's status based on external information or stimuli. For instance, they might change node status from *Candidate* to *Not Candidate* and vice versa based on VM available resources.

4.2.3 Clustering Aspects in EMS

The implementation of the enhanced EMS uses the Atomix[41] framework for building EMS clients' local clusters. As stated in the framework's site *"Atomix is an event-driven framework for coordinating fault-tolerant distributed systems using a variety of proven distributed systems protocols."* Atomix aims at addressing limitations of previous, open source distributed systems, while simultaneously providing an easy-to-use, reliable, and scalable solution. Compared to the well-known Apache ZooKeeper[42], Atomix provides a more flexible and usable API. Also, it does not require the existence of coordination servers to work, like ZooKeeper does, but it can be embedded in other systems (like EMS client). Atomix API is influenced by Hazelcast's[43] easy-to-use and simplified API, but it is backed by algorithms and protocols providing strong consistency guarantees. Atomix is open source project and is licensed under the Apache 2.0 license.

We are particularly interested in (a) the clustering functionality provided by Atomix, (b) the capability to synchronize node statuses and properties throughout a cluster, and (c) the broadcasting service it offers. The former refers to the ability of Atomix to form a cluster and then continuously monitor the availability of cluster nodes. Cluster formation involves the use of a bootstrap protocol that each node must run. The first node deployed will create the cluster while the subsequent nodes will contact one of the existing nodes in order to introduce themselves to other cluster members and receive current cluster status. For the enhanced EMS, a list of adjacent nodes (peers) will be provided by the EMS server. Upon contacting one of the peers, the joining node will receive a full list of cluster members. EMS server groups EMS clients based on the *location* of the VMs they are deployed on. Location refers to an availability zone or a region or a whole cloud service provider (i.e. scope), where a VM has been created and operates. EMS server operates based on the assumption that VMs residing in the same location are close enough and can communicate quickly and reliably. For this reason it groups clients in the same location together and drives the formulation of a local cluster in each location/scope, by providing different cluster settings and adjacent client lists per location/scope[44].

After successfully joining a cluster, each node executes a protocol for monitoring the availability status of other cluster nodes. The default protocol provided by Atomix is based on the SWIM algorithm [20]. The protocol dictates that each node, periodically queries the status of a number of randomly selected peers by sending a "ping" message. If ping is not acknowledged within a certain time window, the peer is marked as *suspected* and the node retries for a number of times. If the peer still does not acknowledge the queries, the node will notify other peers about the unresponsive node. When the majority of nodes verify a peer is unresponsive that peer is declared dead and removed from the cluster (as well as the *alive nodes* list maintained in each node). Of course, if the dead node recovers it can re-join the cluster at any time.

The second clustering feature of Atomix refers to the ability to replicate the status of a node to its peers, and vice versa, as soon as any status change occurs. This feature provides a (near-real time) synchronization mechanism between peers. A node's status includes node properties, which are string-based, key-value pairs. The synchronization mechanism guarantees that all peers have the same view of other node properties (i.e., identical copies). Some inconsistencies might occur for very short periods just after a status change, until changes are transmitted to all peers. The node status and properties synchronization provide the means for implementing the distributed aggregator selection protocol that is described next (see Section 4.2.4).

The last feature of Atomix that we adopt is the broadcasting service, which involves the transmission of messages to all cluster nodes. The distributed, aggregator selection protocol requires broadcasting messages for notifying peers about aggregator changes or triggering a new aggregator selection iteration.

4.2.4 Aggregator selection process

Aggregator is a selected EMS client that receives measurements collected by adjacent EMS clients participating in the same local cluster (peers) and performs scope-wide computations on monitoring data. These computations are defined as complex event processing rules and produce new metrics (in the form of complex events), which are subsequently forwarded to the next level (typically the EMS server). The measurements can possibly be filtered and pre-processed at the source EMS clients, before sent to aggregator. For example, the instant CPU load at each node might be smoothed (per node) and then sent to the cluster aggregator to calculate the cluster-wide CPU load average.

---

[41] Atomix web site: https://atomix.io/

[42] ZooKeeper web site: https://zookeeper.apache.org/

[43] Hazelcast web site: https://hazelcast.com/

[44] For example, if clients A and B are deployed in VMs at Amazon and C, D in VMs at Google, EMS server considers two locations (Amazon and Google) and will create two groups (A,B) and (C,D). Suppose that A is deployed first. Then A will create a local cluster in Amazon cloud. Next, C is deployed in Google cloud. C will create another local cluster in Google. Next, B is deployed in Amazon. EMS server will notify it that a cluster already exists and A is a member, so new client B will contact A and join the Amazon cluster. The same goes with client D. It will contact client C and join the Google cluster.

In the federated EMS, aggregators are selected per cluster using a distributed selection algorithm. Aggregator selection will typically occur during multi-cloud application deployment or redeployment, or if an aggregator node is lost from the topology.
When a multi-cloud application is deployed (or redeployed), the following high-level steps occur in EMS as depicted in the BPMN diagram of Figure 8:

- Executionware notifies EMS when each application VM is created and started.
- The EMS server installs and launches EMS clients on all application component hosts.
- Each EMS client, when boots, it connects to the EMS server and receives its configuration that includes cluster settings, a unique *node id*, an adjacent nodes list and credentials (needed for joining cluster), node's properties (like number of cores, memory, disk available), as well as settings specific to aggregator selection, like a node scoring function. Moreover, the EMS server provides an initial value for node property *"Node Status"* which is a focal setting of aggregator selection algorithm. This property is synchronized along with node properties, across all cluster nodes.
- The EMS client will attempt to join the local cluster as soon as it receives its configuration, using the adjacent nodes list and credentials.

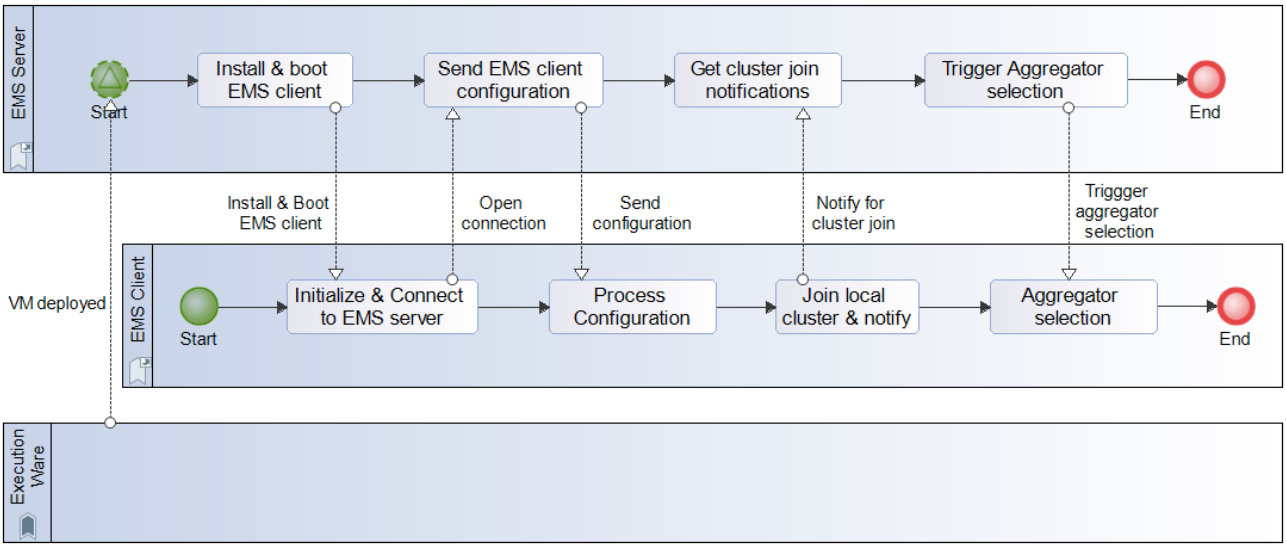

Figure 8. EMS Client installation and initialization

- At this phase, no EMS client acts as aggregator in any cluster yet.
- When application deployment/redeployment completes, the EMS server signals EMS clients (at all local clusters) to select an aggregator per cluster.
    - More advanced scenarios are also possible, for instance start aggregator selection when at least two EMS clients exist in the same scope, or after a certain period of time. However, for the time being we will implement the approach mentioned above (i.e., selection when deployment completes).
- Upon receiving the aggregator selection signal each EMS client executes the aggregator selection protocol described next in section 4.2.4.1.

4.2.4.1 Aggregator selection protocol

The aggregator selection protocol specifies that the following steps are taken by each EMS client for successfully setting up the federated and resilient monitoring topology. These steps have been depicted in the BPMN diagram of Figure 9.

- The EMS client gets the alive nodes list (from cluster software) and reads the *Node Status* properties of all nodes (including itself) in a list. It then filters out those nodes with status denoted as *Not Candidate* or *Retiring*. This way a list of nodes capable to serve as aggregators is created.
- It applies the node scoring function to each node in the aggregator-capable nodes list. Variable values used in node scoring function are taken from each node's properties. An example of node scoring function (provided by the EMS server) is:

$$score_{inst} = 0.2\frac{CPU_{inst}}{CPU_{max}} + 0.8\frac{RAM_{inst}}{RAM_{max}}$$

Variables *CPU* and *RAM* will take their values from each node's properties. Therefore, values must be provided for these properties of each node or else defaults can be used.

- It selects the node with the highest score for aggregator. The Atomix framework ensures that all nodes have the same view of node properties using SWIM protocol for synchronizing. Therefore, all nodes will calculate the same node scores and select the same node for aggregator. In case of two or more nodes having scores equal to the highest score, the node with the lexicographically highest *"Node Id"* will win and assume the role of aggregator. Node Id is a Universally Unique Identifier (a 32-character string of letters or numbers) assigned by the EMS server.
- If a node is not the one selected for aggregation and its current status is:
    - *Candidate*, it will remain in the *Candidate* status and take no further action.
    - *Broker*, this means it is the aggregator stepping down. It will remain in *Broker* status until the new aggregator declares it is ready to serve incoming monitoring data. Then this node will change its status to *Not Candidate* until it clears all aggregator settings. After this it will change to *Candidate* status (so it will have the opportunity to get selected again in future reconfigurations).
    - *Initializing*, this means this node has been selected as the aggregator but then another node was selected, before this one completes the aggregator preparations. Such a situation might occur when a new node (with properties resulting to a new highest score) is joining the cluster while at almost the same time current aggregator is declared dead. Due to network latency, the cluster status between nodes might go out-of-sync for a short period and result in two nodes declaring themselves as winners (because one is not yet aware of the better, new node). In this case this node must interrupt preparation and step down, and by no means should it declare itself as the new aggregator.
    - *Retiring*, is the status of a previous aggregator that wishes to step down (based on its current resources available). It will remain in this status until the new aggregator declares it is ready to serve. Then this node will change its status to *Not Candidate* so it will not be selected again. It can however change its status back to *Candidate*, when it will again be able to serve as aggregator.
- If a node is the one selected and its current status is:
    - *Candidate*, it will immediately change its status to *Initializing*, broadcast a message to other nodes and start preparing to act as an aggregator (this involves configuring its message broker topics and the complex event processing engine with aggregator's CEP rules, included in EMS client configuration). When the preparation completes, the node will change its status to *Broker* and broadcast to other cluster nodes a message including the settings needed in order to forward events to it (this includes at least the Broker's URL and credentials).
    - *Broker*, means that the current aggregator has been selected again, so nothing needs to change.
    - *Initializing*, means the current node has been reselected before completing its aggregator preparation. The node will continue its preparation for becoming the new aggregator as described in *Candidate* case.

When a node with *Initializing* node status receives a broadcast message from another node declaring itself as the new aggregator with status *Initializing* (i.e., the new aggregator is preparing; it's not yet ready to serve), it must interrupt aggregator preparation and switch its status back to *Candidate*. When a node receives a broadcast message from another node declaring itself as the new aggregator with status *Broker* (i.e., ready to serve), then if this node's status is

- *Initializing* - it must interrupt aggregator preparation taking place and switch back to *Candidate* status.
- *Broker* - it must temporarily switch its status to *Not Candidate*, reverse aggregator settings (i.e., step down), and eventually switch status from *Not Candidate* to *Candidate*.
- *Any status* - it must immediately apply the brokering settings sent with the broadcast message. These settings include the URL and credentials of new aggregator's broker.

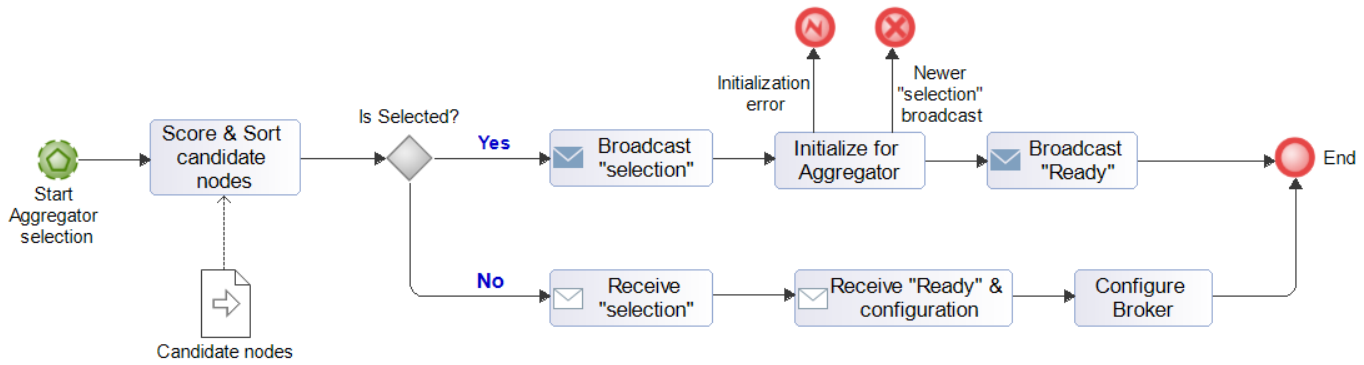

Figure 9. Aggregator Selection process

The node status property controls whether a node can be selected as a cluster aggregator or not, as well as the selection phase. The Node status values can be:

- *Not Candidate* - the node cannot become an aggregator. This is the case of nodes running in resource-constrained devices as some Edge devices are.
- *Candidate* - this is the initial status of all nodes, if they are not running in resource-constrained devices.
- *Broker* - is the status of the node that acts as the local cluster aggregator.
- *Initializing* - is the status of the node that has been selected for aggregator, while it prepares itself for acting as an aggregator. When preparation completes, the status changes to *Broker*.
- *Retiring* - is the status of the current aggregator, if for any reason it needs to yield aggregator role (for example because VM experiences an increased load). In this case the aggregator selection protocol must be activated with a broadcast message. When another node becomes aggregator, the retiring node's status changes to *Not Candidate*. It is possible to switch status back to *Candidate*, if the situation that demanded the retirement is no longer applicable (for example load returned to a normal level).

The types of messages broadcasted are:

- *Aggregator selection message* - sent by EMS server after cluster initialization, or from EMS client clustering subsystem because the previous aggregator has been removed from cluster, or by a retiring aggregator to initiate the selection of its successor.
- *Aggregator initializing message* - the node selected as aggregator informs other nodes it starts preparing.
- *Aggregator ready to serve message* - the selected aggregator is now ready to serve and sends along the needed brokering settings.

Figure 10 gives an overview of the state-transitions of a node with regards to the aggregator selection life-cycle.

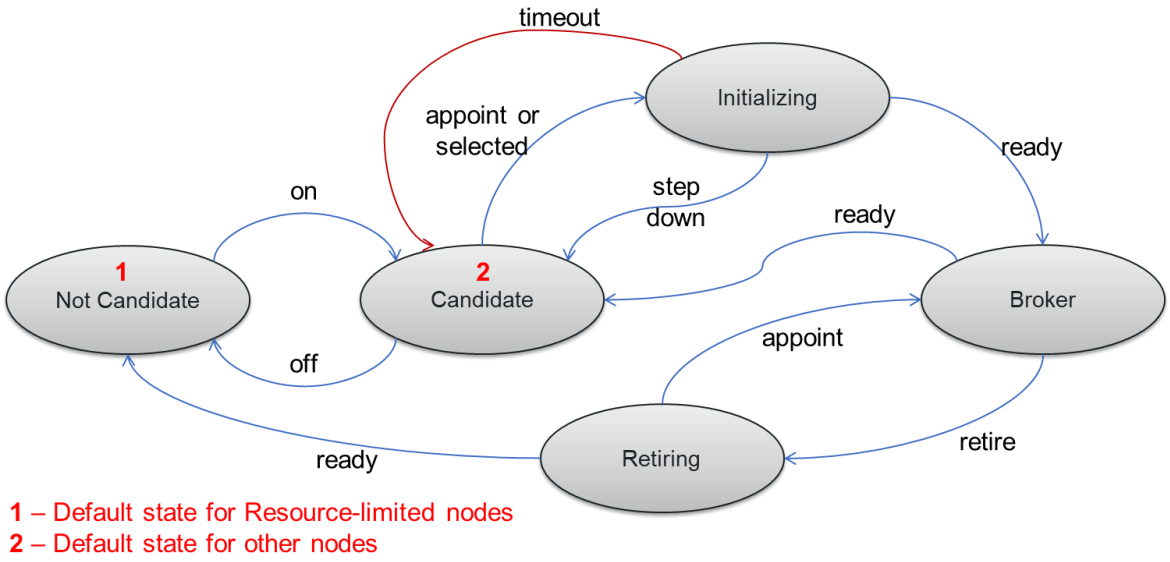

Figure 10. Aggregator State-Transitions

During normal application operation (i.e., any deployment or redeployment has been completed), no aggregator change takes place unless:

- the node serving as aggregator is removed from the cluster, either due to a VM or EMS client failure or a prolonged network disruption,
- the node serving as aggregator decides to retire because of an exceptional situation that came up or because the node must be decommissioned,
- the EMS server appoints a specific node as cluster aggregator. This capability will be added in a future version of EMS.

4.3 Architecture of the self-healing federated EMS

EMS server and EMS client software architectures have been revised and extended to support the above mentioned federation and self-healing capabilities. Moreover, this extended architectural design considers the use of plugins. These plugins can extend certain aspects of the EMS server or clients respectively, possibly adding new functionality.

Plugins can be added and configured through standard EMS server and client configuration files as it is explained below.
Figure 11 depicts the new EMS server software architecture, where new components are marked with different fill colour, namely *Local Topology Manager*, *Client Installer* and *Plugin Framework*.

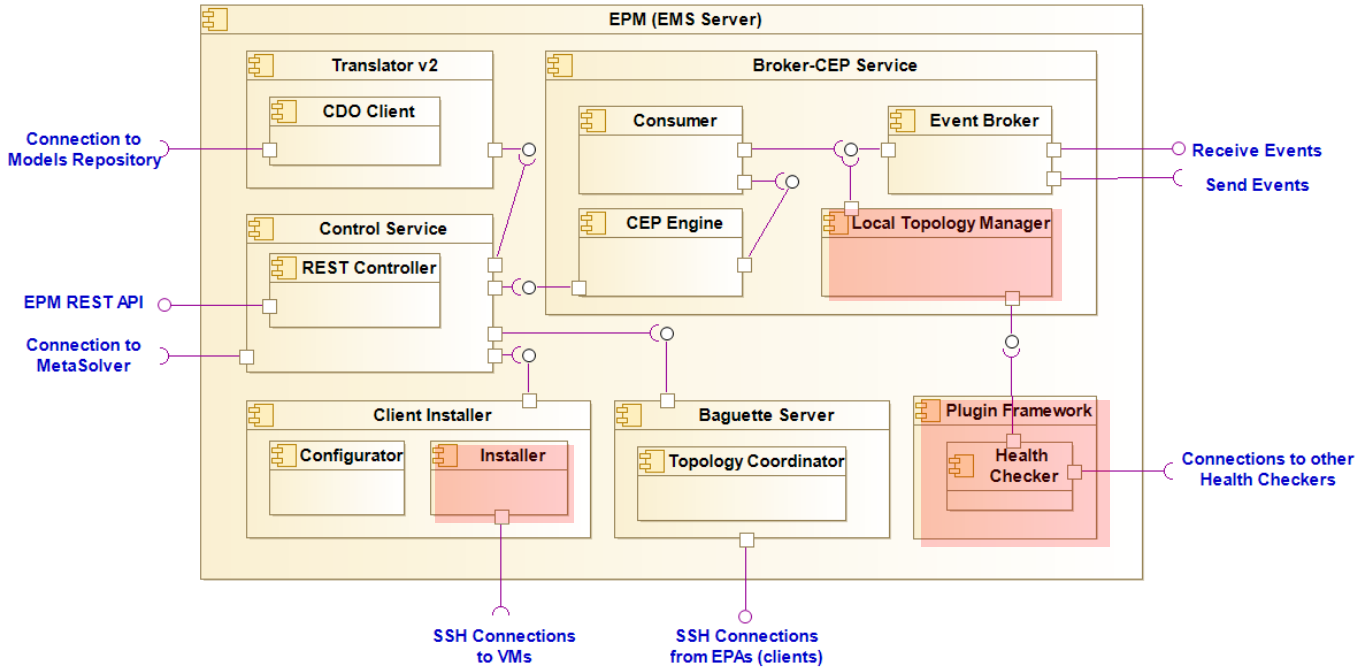

Figure 11. Revised EMS server architecture

Next the new components of EMS server (marked denoted with different colour in the figure above) are briefly described.

- *Local Topology Manager*. It is responsible for joining the EMS server in the local cluster of EMS servers. It also maintains a view of the available EMS servers in the cluster. One of them is the active server (i.e., the Aggregator) that EMS clients are aware of and interact with. The remaining are stand by servers. If the active EMS server becomes unavailable, Local Topology Managers of stand-by servers will select a new active EMS server, using the Aggregator selection protocol described in Section 4.2.4.1. The new active EMS Server will reconfigure the EMS clients with its address and credentials. Active EMS server replicates its current state (including configuration and information of EMS clients) to stand-by servers or persists it to a common storage so that if any stand-by server becomes the active one it will able to recover the work state of the previous active server.
- *Installer*. It is part of the Client Installer component of EMS server. When Executionware announces a new application node to EMS server, the control service of EMS server will pass the relevant information to Client Installer component. In turn Client Installer will use the Configuration sub-component to prepare the required configuration and then engage the Installer sub-component, which will connect to the new application node using SSH protocol. It will subsequently execute a number of predefined jobs, captured as sets of OS-specific instructions, for installing and configuring an EMS client and a Netdata agent. Predefined jobs can be customized in order to carry out additional tasks.
- *Plugin Framework*. It is a programmatic API that enables the registration and interaction of plugins with EMS server. Plugins extend the EMS server by introducing new capabilities. For instance, a Health Checker plugin is bundled with EMS server, which can check the availability of other (stand-by) EMS servers. Health Checker plugins inform the Local Topology Manager when other EMS servers malfunction or become inaccessible.

Figure 12 depicts the new EMS client software architecture where new components are marked with different fill colour, namely *Local Topology Manager* and *Plugin Framework*.

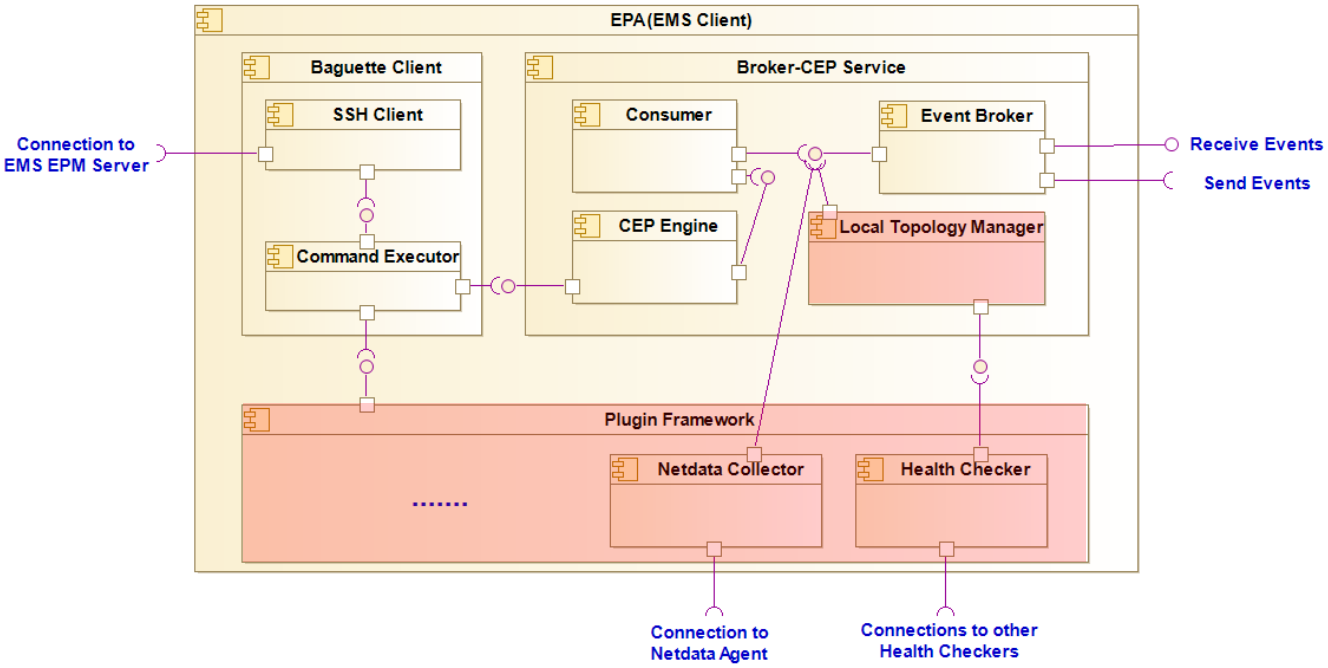

Figure 12. Revised EMS client architecture

Next the new components of EMS client (marked denoted with different colour in the figure above) are briefly described.

- *Local Topology Manager*. It is responsible for joining the EMS client in the local cluster of EMS clients. It also maintains a view of the available EMS clients (nodes) in the local cluster. Local Topology Manager is also responsible for executing the Aggregator selection protocol if the cluster aggregator is lost. It also updates the Broker-CEP component configuration when a new aggregator publishes its address and credentials. Eventually, it is notified about any changes regarding the active EMS server and updates Broker-CEP configuration accordingly.
- *Plugin Framework*. It is a programmatic API that enables the registration and interaction of plugins with EMS client. Plugins extend the EMS client by introducing new capabilities. As shown in Figure 12, a Netdata Collector plugin is bundled in EMS client installation package. It extends EMS client for contacting its local Netdata agent and retrieving the configured measurements. It subsequently posts the retrieved measurements to local EMS event broker as events. Additionally, a Health Checker plugin is also included, which periodically connects to the event brokers of adjacent EMS clients to verify they are operating properly.

More plugins can be added in the future, for instance health check plugins, or recovery plugins. Application-specific scenarios are also possible, like plugins collecting and publishing application related events. EMS clients can also be extended to monitor the resource availability of the local VM and change the state of local cluster node from *Candidate* to *Not Candidate* and vice versa.

Last, we note that this architecture is currently been validated in a number of preliminary experiments that exploit resource constrained devices, located close to the edge of the network. Netdata monitoring probes and EMS clients have been successfully installed on Raspberry Pi 3 devices to propagate monitoring measurements to the EMS Server. This work to be reported as part of the next EMS release, it will continue towards the consideration of further improvements that will allow for the efficient monitoring data aggregation from edge resources.

4.3.1 EMS self-deployment

As mentioned above, EMS server has been enhanced with a new, configurable self-deployment procedure, by extending its Baguette Client Installation component. EMS self-deployment is based on the notion of connecting to already deployed application nodes and executing predefined sets of node-specific instructions. These instructions intend installing EMS client as well as the appropriate monitoring probes (e.g., Netdata) on application nodes, but they can be used to carry out additional tasks.

Specifically, during application deployment or redeployment, Executionware announces the addition of each new application node, using the dedicated EMS server REST endpoint. This triggers the Baguette Client Installation subsystem to connect to them, using the provided network addresses and credentials (typically using SSH), and executes a number of predefined instructions sets corresponding to application node types (VM or other) and operating system families. These instruction sets are part of the EMS server configuration.

For the time being, the default self-deployment configuration includes three instruction sets for installing EMS client and Netdata monitoring system on Linux-based VMs. In the future, more instruction sets can be included to support additional node types, operating systems, and sensors.

- The first instruction set checks if the necessary system tools are present in the application node. These tools include Java™ Runtime Environment needed for running EMS, and `sudo` and `tar` commands for unpacking EMS installation package to the target path. It also checks the necessary permissions to folders and system tools have been granted.
- The second instruction set installs EMS client. It first copies the necessary installation files from EMS server to application VM and sets the needed permissions. It subsequently generates EMS client configuration tailored for that particular node and copies it to the VM as well. Eventually it launches EMS client.
- The third instruction set is used to install the Netdata monitoring system on the deployed VM. This set copies Netdata configuration tailored to that particular VM and subsequently executes the Netdata installation commands. Netdata is an extensible and configurable monitoring solution offering the possibility to deploy additional sensors in the future. Even the deployed application components (installed in the same VM) can also log their measurements to Netdata. Subsequently, the local EMS client can contact Netdata and retrieve the needed measurements.

EMS self-deployment brings several advantages to EMS. In Melodic, Executionware was responsible for installing and launching EMS clients, after it (first) queried EMS server for retrieving the needed instructions and configuration. It was also responsible to install a number of limited infrastructure sensors (like CPU load, RAM and disk usage sensors). Self-deployment simplifies the process by moving this responsibility to EMS. At the same time, it reduces the complexity of Executionware since it no longer installs EMS clients or sensors. Removing the dependency of EMS on Executionware (along with the inherent operational assumptions), eliminates one possible point of failure, makes EMS self-contained (since it moves monitoring-related tasks to EMS), and makes the process more controllable and clean.

It is worth mentioning again that EMS self-deployment is configurable (by adding new instruction sets) and it can be used in different deployment scenarios in the future. For instance, it can be used to install the system tools (like Java™ Runtime Environment) needed to run EMS client, or install monitoring systems beyond Netdata, or enable additional system services.

4.4 Related work with respect to clustering and leader election techniques

In this section we highlight notable approaches in failure detection and process membership, as well as leader election techniques, which are related to our work. These efforts provide the background to understand the selection of SWIM[45] as the membership protocol to be used, as well as our aggregator selection methodology which is outlined in Section 4.2.4.

4.4.1 Failure detection and process membership

Katti et al. [22] mention failure detection/consensus algorithms which are based on 'heartbeat' or 'ping' messages. These algorithms are simple and can be relatively easy to implement, however they are known to either require a significant amount of information to be stored for each process (in the case of heartbeat-based algorithm) or to lead to false positives (i.e., failure detection) in cases where the network is unreliable (ping-based algorithm). Further, the algorithm proposed in their work does not cover unreliable networks or process recovery. The shortcomings of these algorithms limit their scalability and the potential to use them in multiple node/process networks.

The need to limit network load has been recognized previously, in approaches such as HiScamp [23] which employs a hierarchical self-organized membership protocol for gossip-based dissemination. SWIM [21] tries to reduce the network load by decoupling the membership update dissemination function from the failure detection function. The failure detection part works by selecting a random node and sending it a ping message, expecting an acknowledgement in return. If it does not receive this, it selects k peers to probe this node through a ping request message. An optimization of this failure detection is to first mark a node as “suspected” and mark it as dead just after a timeout. The dissemination part involves transmitting membership information by piggybacking the failure detection messages instead of using a multicast operation.

SWIM exhibits high scalability; resilience and it is easy to manage [24]. It has also been used as the starting point for further extensions and improvements [25] and as a reference benchmark [26]. Moreover, it is used in well-known

[45] Scalable Weakly-consistent Infection-style Process Group Membership protocol

software products such as Consul, Nomad and Serf [25] and the Swift language server ecosystem [27]. As such, we have elected SWIM to support our failure detection and process membership scheme.

4.4.2 Leader Election approaches

Various algorithms have been suggested for leader election in distributed topologies consisting of nodes. The Bully algorithm [28], declares a node of the topology as a coordinator, when it possesses the highest identifier. The algorithm is based on the transmission of status messages to nodes with higher identifiers than the sending node. When no replies arrive to a particular node, this node assumes the position of a leader and notifies nodes with lower identifiers of the change in leadership.

In [29], Frederickson et al. proposed the use of message processes for leader election in a synchronous ring. Each message process carries the integer identifier of the node which sends the process to other nodes of the ring. Nodes with smaller integer identifiers kill processes from nodes with larger identifiers when they arrive at them. Eventually the sole process which returns to its originator will cause the originator node to be elected as the leader.

Awerbuch [30], presents an algorithm for leader election based on the construction of a spanning tree. During the initial phases of the algorithm, a forest of directed trees is created, which spans all nodes in a network. The root node of each tree is the 'leader' of the tree. As the algorithm is executed, larger trees are allowed to 'invade' and 'conquer' smaller trees (whose leaders then become inactive). When a tree is 'conquered', its nodes recognize as their leader the leader of the larger tree. When two trees of the same level (a measure of tree size) need to be merged, the tree having a leader with a bigger identity conquers the other tree.

In another effort, Malpani et al. [31] present two leader election algorithms for mobile ad hoc networks. The first (for which a proof of correctness is also provided) can handle only a single topology change, while the second allows for multiple concurrent link failures. Another approach for mobile ad hoc networks is provided by Vasudevan et al. in [32], which is also formally verified. The use of node identifiers is instrumental in electing a leader, in all of these algorithms. In [33] the authors try to minimize the number of leader election processes to improve energy efficiency. To achieve this, they build upon [32] so that every node keeps a leader list rather than a single leader. Similar to the previous approaches, this work also elects leaders, based on the value of their identifiers.

The approaches discussed above, use an ID value to determine the leader of the topology, but this value is not associated with the processing capacity of a node. A possible extension (which is also expressed in [32]) of these algorithms could involve using the values expressing the processing capability of each node (e.g., number of cores, amount of RAM) to determine the ID value.

In [34], the leader election process is tuned in order to select a near-optimal agent as a leader using a decentralized algorithm. A leader is defined as optimal, when it is well connected to the network. The measurement of how well-connected to the network a node is, it is done using easy to check statistical properties of the network graph. These properties are subsequently correlated to the cost function which should be minimized. Thus, the leader election is done in both a time and computationally-efficient way, as opposed to the calculation of a Ricatti equation for every potential leader. In another effort, Han et al. [35] attempt to optimize leader election in hypercube networks, minimizing the degree of interference. The degree of interference is a custom metric defined by the authors intended to measure the total network resource usage. However, in our approach we allow for networks which have a more dynamic topology and are not restricted to the hypercube structure.

In [36] a leader election algorithm for wireless ad hoc networks is described. The authors propose the usage of the remaining power of a node and the node degree (i.e., the 1-hop connections from/to this node) as the metrics which should guide the leader election. Moreover, they demonstrate a significant improvement in the messages required to be exchanged using their algorithm compared to [28]. In [37], Vasudevan et al. describe two leader election algorithms for wireless ad-hoc networks, named SEFA and SPLEA. Both algorithms proceed in iterations to elect a leader node. In SEFA the node, which is elected as leader has the highest utility, as determined by a utility function which is shared by all nodes to grade candidate leaders. In SPLEA each node can use a different function to grade a leader (allowing the elector node's preferences to be accounted for) and casts its vote on the candidate node with the higher utility value. In each iteration each node accepts as its parent the node with the most votes, and eventually the sole remaining parent node is the leader. All utility functions can accept multiple metrics, however, they are defined considering attributes pertaining to particular nodes and not considering aggregations (e.g., minimum value). Furthermore, these algorithms are targeted at network topologies which do not change during election. Park et al. [38] describe a protocol which uses information from each process in the form of a weight variable (e.g., process battery lifespan) to determine the value of each node. The election of the leader builds upon the GMDA group member detection algorithm. It is performed in three phases – the spreading phase (of election messages), the collection phase (of the identity and weight of each process) and the completion step which actually chooses the process with the highest value and broadcasts the leader to all processes. This indicates that performing an aggregation over the values (e.g., finding the maximum value) is feasible, which is relevant to our approach. Biswas et al [39] describe a leader

election algorithm for bidirectional ring topologies which uses a weighted utility function including system variables (CPU, RAM, Bandwidth) and the node failure rate, to determine the leader coefficients of nodes. Lower leader coefficients indicate nodes which should be preferred in the election process.
Our approach embraces the concept of using system characteristics for leader election, by allowing the usage of a variety of monitoring metrics to determine the leader of the topology. Further, we are able to use aggregated information derived from the system characteristics of other nodes in the topology (e.g., maximum amount of RAM). Moreover, our approach is not bound to the usage of any particular monitoring topology, operating in a non-probabilistic manner. Considering the processing capacity of each node to elect the leader allows for better monitoring load handling – especially useful in the cases of non-trivial monitoring workloads.

5 Conclusions & Future Work

This deliverable discussed the ongoing work on an advanced monitoring system that enables the Morphemic platform to aggregate and manage valuable monitoring data coming from an unbounded number of dispersed application component instances of a multicloud application. This event-driven performance monitoring system called EMS is designed to cope with vast amounts of monitoring data in a decentralised and resilient manner. The purpose is to efficiently exploit this data in order to reactively and proactively optimise the multicloud application configuration and deployment topology. In order to accomplish this, we started with an analysis of the most prominent open-source tools used for monitoring and also discussed a time-series based approach for persisting such data to be vested in the future for forecasting purposes. Based on this, we introduced the design of the federated EMS mechanism. Moreover, we introduced self-healing capabilities for constituting EMS a resilient monitoring system that is able to cope with dispersed monitoring sources.
The next steps of this work involve the design and implementation of additional EMS extensions that will be able to digest monitoring data coming from different monitoring probes and systems (e.g., InAccel's unified monitoring layer for FPGAs). Furthermore, we will investigate the appropriate means for considering hosting resource awareness when EMS server allocates monitoring roles and duties to EMS clients for constituting EMS even more efficient for digesting data from edge resources.

References

[1] Chen, J.,Wang, C., Zhou, B.B., Sun, L., Lee, Y.C., Zomaya, A.Y.: Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing, pp. 229–238. ACM, New York (2011). https://doi.org/10.1145/1996130.1996161. http://doi.acm.org/10.1145/1996130.1996161
[2] Singh, S., Chana, I.: A survey on resource scheduling in cloud computing: Issues and challenges. J. Grid Comput. 14(2), 217–264 (2016)
[3] Aceto, G., Botta, A., De Donato, W., Pescape, A.: Cloud monitoring: a survey. Comput. Netw. 57(9), 2093–2115 (2013)
[4] Agarwala, S., Poellabauer, C., Kong, J., Schwan, K., Wolf, M.: System-level resource monitoring in high-performance computing environments. J. Grid. Comput. 1(3), 273–289 (2003)
[5] Fatema, K., Emeakaroha, V.C., Healy, P.D., Morrison, J.P., Lynn, T.: A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. J. Parallel Distrib. Comput. 74(10), 2918–2933 (2014)
[6] da Rosa Righi, R., Lehmann, M., Gomes, M.M. et al. A Survey on Global Management View: Toward Combining System Monitoring, Resource Management, and Load Prediction. J Grid Computing 17, 473–502 (2019). https://doi.org/10.1007/s10723-018-09471-x
[7] Morton, A.: Active and passive metrics and methods (with hybrid types in-between). RFC 7799 (Informational) (2016)
[8] V. Stefanidis, Y. Verginadis, I. Patiniotakis, and G. Mentzas, “Distributed complex event processing in multiclouds,” in Proceedings of the 7th European Conference on Service-Oriented and Cloud Computing, September 2018.
[9] D. Baur, F. Griesinger, Y. Verginadis, V. Stefanidis and I. Patiniotakis, "A Model Driven Engineering Approach for Flexible and Distributed Monitoring of Cross-Cloud Applications," 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), Zurich, 2018, pp. 31-40, doi: 10.1109/UCC.2018.00012.
[10] Baran, P. “On Distributed Communications Networks.” Communications Systems, IEEE Transactions on 12.1 (1964): 1–9. Print.
[11] Amsterdam university of applied sciences. Beyond distributed and decentralized: what is a federated network? https://networkcultures.org/unlikeus/resources/articles/what-is-a-federated-network/
[12] E. Elmroth, F. G. Marquez, D. Henriksson and D. P. Ferrera, "Accounting and Billing for Federated Cloud Infrastructures," 2009 Eighth International Conference on Grid and Cooperative Computing, Lanzhou, China, 2009, pp. 268-275, doi: 10.1109/GCC.2009.37.
[13] A.J. Ferrer et al., “OPTIMIS: A Holistic Approach to Cloud Service Provisioning,” Future Generation Computer Systems, vol. 28, no. 1, 2012, pp. 66-77
[14] R. Moreno-Vozmediano, R. S. Montero and I. M. Llorente, "IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures," in Computer, vol. 45, no. 12, pp. 65-72, Dec. 2012, doi: 10.1109/MC.2012.76.
[15] Drăgan, I., Iuhasz, G. & Petcu, D. A Scalable Platform for Monitoring Data Intensive Applications. J Grid Computing 17, 503–528 (2019). https://doi.org/10.1007/s10723-019-09483-1
[16] Y. Verginadis, C. Chalaris, I. Patiniotakis, V. Stefanidis, F. Paraskevopoulos, E. Psarra, B. Magoutas, E. Bothos, E. Anagnostopoulou, K. Kritikos, P. Skrzypek, M. Prusiński, M. Różańska (2019) D3.4: Workload optimisation recommendation and adaptation enactment. H2020 Melodic deliverable.
[17] Villalpando, L.E.B., April, A., Abran, A.: Performance analysis model for big data applications in cloud computing. J. Cloud Comput. 3(1), 1–20 (2014)
[18] Damian A. Tamburri, Marco Miglierina, Elisabetta Di Nitto, Cloud applications monitoring: An industrial study, Information and Software Technology, Vol. 127, 2020, https://doi.org/10.1016/j.infsof.2020.106376.
[19] Venner, J., Wadkar, S., Siddalingaiah, M.: Pro Apache Hadoop. Apress (2014)
[20] SWIM: The scalable membership protocol. Available online at https://www.brianstorti.com/swim/ . Retrieved on 8/1/2021
[21] A. Das, I. Gupta και A. Motivala, «SWIM: scalable weakly-consistent infection-style process group membership protocol», in Proceedings International Conference on Dependable Systems and Networks, Washington, 2002.
[22] A. Katti και D. J. Lilja, «Efficient and Fast Approximate Consensus with Epidemic Failure Detection at Extreme Scale», in 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Cambridge, 2018.
[23] A. J. Ganesh, A.-M. Kermarrec και L. Massoulié, «HiScamp: self-organizing hierarchical membership protocol», in Proceedings of the 10th workshop on ACM SIGOPS European workshop, 2002.

[24] A. Dadgar, J. Phillips και J. Currey, «Lifeguard: Local Health Awareness for More Accurate Failure Detection», in 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Luxembourg City, 2018.
[25] A. Dadgar, J. Phillips και J. Currey, «Lifeguard : SWIM-ing with Situational Awareness», 2017.
[26] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens and J. Dongarra, "A failure detector for HPC platforms," The International Journal of High Performance Computing Applications, vol. 32, p. 139–158, 1 2018.
[27] Swift Cluster Membership. Available online at https://swift.org/blog/swift-cluster-membership/. Retrieved on 8/1/2021.
[28] H. Garcia-Molina, «Elections in a distributed computing system», IEEE transactions on Computers, p. 48–59, 1982.
[29] G. N. Frederickson και N. A. Lynch, «Electing a leader in a synchronous ring», Journal of the ACM (JACM), vol. 34, p. 98–115, 1987.
[30] B. Awerbuch, «Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems», in Proceedings of the nineteenth annual ACM symposium on Theory of computing, 1987.
[31] N. Malpani, J. L. Welch and N. Vaidya, "Leader election algorithms for mobile ad hoc networks," in Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications - DIALM '00, Boston, Massachusetts, United States, 2000.
[32] S. Vasudevan, J. Kurose και D. Towsley, «Design and analysis of a leader election algorithm for mobile ad hoc networks», in Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004., Berlin, 2004.
[33] S. Lee, R. M. Muhammad and C. Kim, "A Leader Election Algorithm Within Candidates on Ad Hoc Mobile Networks," in Embedded Software and Systems, vol. 4523, Y. Lee, H. Kim, J. Kim, Y. Park, L. T. Yang and S. W. Kim, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, p. 728–738.
[34] T. Borsche και S. A. Attia, «On leader election in multi-agent control systems», in 2010 Chinese Control and Decision Conference, Xuzhou, 2010.
[35] S. C. Han και Y. Xia, «Optimal Leader Election Scheme for Peer-to-Peer Applications», in Sixth International Conference on Networking (ICN'07), Sainte-Luce, Martinique, France, 2007.
[36] H.-C. Cahng και C.-C. Lo, «A Consensus-Based Leader Election Algorithm for Wireless Ad Hoc Networks», in 2012 International Symposium on Computer, Consumer and Control, Taichung, 2012.
[37] Vasudevan, Sudarshan, et al. "Secure leader election in wireless ad hoc networks.", in UMass Computer Science Technical Report 01–50 (2001).
[38] S. Park, S. Yoo and B. Kim, "An election protocol based on group membership detection algorithm in mobile ad hoc distributed systems," The Journal of Supercomputing, vol. 74, p. 2239–2253, 5 2018.
[39] A. Biswas, A. K. Maurya, A. K. Tripathi and S. Aknine, "FRLLE: a failure rate and load-based leader election algorithm for a bidirectional ring in distributed systems," The Journal of Supercomputing, vol. 77, p. 751–779, 1 2021.
[35] InfluxDB write Protocol, https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/, retrieve on 15/01/2021
[36] Melodic D6.2, Use cases Implementation and feedbacks, https://melodic.cloud/wp-content/uploads/2020/03/D6.2-Use-Cases-Implementation-and-Feedback-1st-Iteration.pdf, retrieve on 17.01.2021