Real-Time Telemetry System for Emergency Situations using SWITCH

George Suciu, Cristiana Istrate, Dorinel Filip, Vlad Poenaru, Andrei Scheianu
R&D Department
BEIA Consult International
Bucharest, Romania
george@beia.ro

Matej Cigale
School of Computer Science and Informatics
Cardiff University
Cardiff, UK
CigaleM@cardiff.ac.uk

***Abstract*—A real-time telemetry system can bring a major difference when situations of emergency arise. This enables people and authorities to save lives and property in case of disasters. In case of floods, a warning issued with enough time before the event will allow for reservoir operators to gradually reduce water levels, people to reinforce their homes, hospitals to be prepared to receive more patients, authorities to prepare and provide help. Such a system collects data from real time sensors, processes the information using tools such as predictive simulation, and provides warning services or interactive facilities for the public to obtain more information. Taking this into account, we propose the development of a use case for environment monitoring in case of a disaster based on environmental legislation, disaster classification and their effects, developing a regional and national data repository with monitored data, knowledge and good practices applied in disaster management. SWITCH (Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications) addresses the urgent industrial need for developing and executing time critical applications in Clouds. The typical implementation for a disaster early warning system is based on a server running virtual machines, while SWITCH will enable to run the components in container.**

Keywords—telemetry; SWITCH; monitoring server; notification system; Cloud; real-time.

I. INTRODUCTION

A real-time telemetry system for emergency situations can be used by people and authorities in order to be aware of the events that they are about to encounter. Events like floods may be less dramatic if a warning would be issued in time, so that the operators could gradually reduce water levels. Also, people would reinforce their homes, hospitals might better prepare for receiving more patients and authorities could organize to provide help.

The main idea of an ideal disaster warning system is to minimize prevention costs and increase prevention efficiency in case of flood and other possible disaster events. In the proposed use case, the telemetry system handles around 500-1000 sensors from different regions in Romania. The values of some attributes characterizing the entities that are relevant to the application, will be calculated based on the combination of measures captured from multiple sensors. The typical implementation for a real-time system for emergency situations is based on a server running virtual machines, while SWITCH will enable to run and deploy the components in a container.

A real-time telemetry system for emergency situations represents a complex architecture which raises problems when it needs to be implemented in various environments and scaled up in those situations. The ecosystem developed by SWITCH can solve these problems by unifying the deployment and management of the architecture into a simple solution, while using containers for the components. One such component is the telemetry station (entities may be of several types - e,g. several types of stations), and depending on the sensors mounted on the station, there are defined the attributes and metadata of the station entity.

This paper is structured as follows: Section 2 deals with the detailed description of related work regarding real-time telemetry system for emergency situations. Section 3 describes the business case underlying the use case and maps this into a set of requirements and the corresponding technical and functional specifications. Section 4 presents a comprehensive architecture for the application is defined, while Section 5 describes its integration within the SWITCH environment. Section 6 discusses the results of implementing an early prototype, together with the description of the software test plans. Finally, Section 7 draws the conclusions.

II. RELATED WORK

The issue of arising emergency situations has been addressed for decades now. Depending on the application specific, the objective functions and optimization methods are different. Telemetry systems bring a major difference when such emergencies arise.

In Almaty region of Kazakhstan, for instance, real-time sensors process the information using tools such as detection systems for earthquake, fire and gas disasters [1]. These systems consist of a sensor network, a disaster information mapping server, an SNS module and a web server. Earthquake Early Warning System (EEWS) of Japan Meteorological Agency (JMA) provides citizens with severe warnings regarding earthquakes that are stronger than “intensity 5 lower” by mobile phones, radio, and TV, while they also provide critical information about tsunamis.

Another area of implementation of a warning system is the detection and prediction of severe weather using commercial cloud services [2]. They provide the required network capability to perform the real-time operation of detection from the radars to the cloud service instance, making the process automated based on the results of weather detection algorithms. The benefit of such a platform fits the needs of many weather applications because it requires fewer resources and less computation when there are no weather events at present or in the near-term future switching to high computation resources only during severe weather conditions [3,4].

Another application using telemetry sensors is related to optimizing the use of water resources in agriculture. This system consists of the distributed wireless sensor network of the soil and the moisture, temperature and color sensors. It can provide sustainable agriculture even in the water scarcity areas. The Productivity in agriculture could also be increased by using the automated irrigation system. Moisture and temperature sensors are being placed in the root zone of the crops while the controller unit is used to manage the irrigation motor thereby controlling the water flow to the field. This controller is also programmed with threshold values of the temperature and moisture content [5].

One specific real-time telemetry system for emergency situations is an early warning system in disaster management based on Libelium technology [6]. Due to the devastation caused by La Liboriana river flood in 2015, the National Unit for Disaster Risk Management of Colombia was forced to monitor and compile information on the main rivers to prevent similar tragedies. The objective was to control the behavior of the river basins and to obtain real-time information signal when the limits are exceeded by generating alerts. Because the 3G coverage in the area was powerless, the project demanded a complex communications system and a 900MHz mesh network was implemented to resolve this issue. On one hand, data had to be stored and shown locally for the decision-makers to decide whether or not to activate the Early Warning System (EWS). On the other hand, data was also stored in cloud for those who weren't directly related to the decision-making process. Data is stored locally in a Raspberry Pi 3B which process the information gathered. The data is then sent to the Meshlium IoT Gateway, where it is stored and forwarded to the Eagle.io cloud platform using a 3G cellular communication protocol. In case of such alert, the control unit activates the sirens for the people to evacuate the risk areas, thus offering security and preventing the community from natural disasters.

III. FUNCTIONAL DESCRIPTION

Disaster early warnings enable people and authorities to save lives and properties. In case of floods, a warning issued earlier enough before the event occurred would allow for reservoir operators to gradually reduce water levels, people to reinforce their homes, hospitals be prepared to receive more patients, authorities to prepare and provide help [7, 8, 9]. But, there is a trade-off between timeliness, warning reliability, the cost of a false alert, and damage avoided as a function of lead time, which must be modeled to determine the cost efficiency of the outcome [10, 11].

The essential structure of any EWS depends on the objectives of the system to provide important, timely information on specific phenomena to end-users and decision-makers, thereby enabling effective response [12]. Depending on factors like the spatial and temporal scale of a specific environmental degradation, the geographic area, size of the phenomenon, and the objectives of the monitoring program, some systems may not be considered as fully integrated EWS [13, 14, 15].

All well integrated EWSs tend to contain four major components:

- Information and multidisciplinary data collection on the phenomenon;
- Evaluation, processing, and analysis of collected data;
- Dissemination of warning information to policy-makers and final users;
- Implementation of an effective and timely response to the early warnings issued.

The Elastic disaster early warning system application represents a "cloudified" early warning solution for natural disasters. The application collects data from real time sensors, processes the information, and provides warning services for the public. The system should be capable to collect and processing the sensor data in real time, and thus allowing very rapid response to urgent events. Besides this the application should provide sufficient reliability and availability, and scalability to the increasing number of sensors.

The proposed use case functional architecture diagram and real-time constraints are presented in Fig. 1:

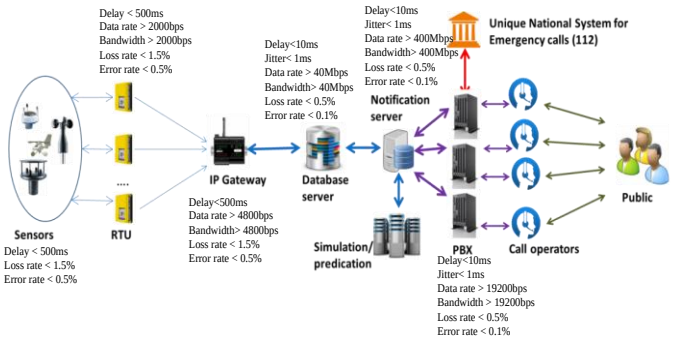

Fig. 1. Functional diagram for elastic early warning system

The implementation of this type of system faces several challenges, because it must:

- Collect and process the sensor data in nearly real time;
- Detect and respond to urgent events very rapidly (i.e. this is a time-critical scenario);
- Predict the potential increase of load on the warning system when public users (customers) increase;
- Operate reliably and robustly throughout its life time;
- Be scalable when the number of sensors increases.

The technological advances using SWITCH consist in a solution that integrates the sensors with a cloud platform that can offer real time information and decision in case of disaster. These sensors can be used for measuring several parameters like water level, water flow, temperature, pressure, but also some parameters that define the environment quality such as: water, air and soil quality.

Sensors in the field transmit information to IP Gateway via GSM / GPRS / Radio. The gateway transmits the data collected from the sensors to the database server. The notification server periodically checks the data from the database, and if they exceed certain values set on different communications channels, notifications are sent to the Alarm Trigger. We also could deploy sensor networks for real-time monitoring including seismic activity, radioactivity, tsunamis, marine / maritime activities, and floods.

A. *Disaster early warning use case diagram*

The use case diagram for disaster early warning can be seen in Fig. 2.

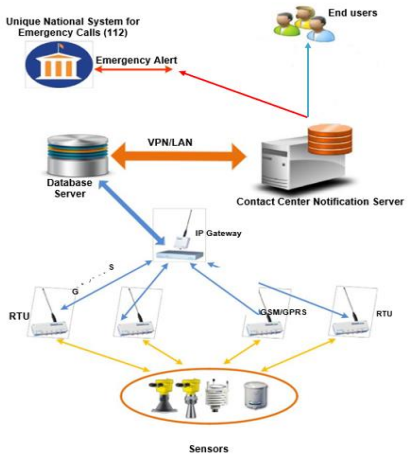

Fig. 2. Use case diagram for disaster early warning

The main actors are:

- End-users: Citizens, Service Providers, etc. (who can seek help much easier through the proposed platform).
- Emergency stakeholders, Government Emergency Agencies, Environment Agencies, Cloud Providers, etc. (which may provide help much faster by using the proposed platform).
- Platform System Operator: supervise server status, traffic trespassing firewall cluster, etc.

The system architecture must fulfill the following requirements:

- Sensors that transmit the field data (temperature, humidity, sensors that measure water level, etc.)
- Subsystem for transmitting data from the sensors to the database from the Remote Telemetry Unit (RTU) and IP Gateway.
- The database where data is stored, and reports/statistics are created based on information received from the sensors.
- Notification Server checks DB (database) data and statistics in real time and sends notifications to operators if there are values outside of predetermined unbroken.

The success end conditions specify that the architecture should detect, process and transmits sensor data and notify the relevant stakeholders/end-users for different emergency situations: floods, water and air pollution, drought.

Also, failure end conditions should take into account:

- Workflow breaks;
- Security attacks;
- Lack of resources;
- VM crash.

Furthermore, in an emergency situation, the infrastructure should offer minimal services and QoS guarantees as presented in Fig. 1 for at least text notification.

B. *Scenarios in the use case*

Possible issues that must be verified during the implementation process:

- System level performance requirements;
- Verifiability;
- Integration complexity;
- Use of virtualized resources;
- Configuration of the infrastructure;
- Data intensive communication;
- Adaptability for quality-on-demand;
- Adaptability to changing infrastructure;
- SLA negotiation.

IV. TECHNICAL DESCRIPTION

In this section the main components of the sub-systems are described, such as telemetry system, monitoring adapter and notification system.

A. *Telemetry System*

The telemetry system consists of elements like computers, devices or processes that compose the entire system. The presented solution has three major components. RTU (Remote Telemetry Unit) is the device which takes data from the environment and stores it into a cloud database platform. The dashboard represents an interface in which data can be stored and visualized, as separated time-series metrics. Last but not least, there is the Alert component, which reads data from the data bases and sends an alarm if a threshold is exceeded.

The RTU is an electronic remote device which monitors and reports events that happen at a remote site, allowing the network operator to visualize the data distances from where it is implanted. In our development, we used the A753 GPRS RTU, because it is flexible, and it can be deployed in a large variety of applications, from agriculture to hydrographics to professional meteorology, from water quality to flood warning, from AMR (Automatic Meter Reading) to leak detection, from the monitoring of solar power to wind energy. The central elements of the telemonitoring system, are the Data Concentrator (Gateway) and the Data Presentation Server. The gateway performs communication with the RTUs, and also allows the configuration and management of all RTUs and sensors. The second element is hosted on a computer with strong server features (such as safe unattended running 24/24 and 7/7). The Data Presentation Server is based on the software package that is focused on the presentation of data in various formats (e.g. tables and diagrams).

The metrics are extracted from telemetry and then added to dashboards such as Graphite, Grafana or Prometheus in order to visualize them.

1. Graphite

Graphite [16] is an enterprise-ready monitoring tool that runs equally well on cheap hardware or Cloud infrastructure. Teams utilize Graphite to track the execution of their sites, applications, business benefits, or networked servers. It denoted the begin of another age of monitoring tools, making it simpler to store, recover, share, or view time-series metrics.

Graphite does two things:

- Stores numeric time-series data;
- Renders diagrams of this data on request.

Graphite is not a collection agent, but it offers the simplest path for getting your measurements into a time-series database. Also, Graphite has one of the largest ecosystems of data integrations and third-party tools, so one can easily use a collection agent or language bindings, for instance. Furthermore, Whisper is an on-disk database which maintain the long-term reservation for metrics. Whereas carbon plays the role of the receiver for the metrics, WSGI webapp is responsible for REST API for deriving the data out from Whisper for analyzing and demonstration. The webapp collects all the data from a local Whisper backend and can be configured for querying the other webapps too.

Client APIs used for the proposed use case are as follows:

- Graphitejs represents jQuery plugin that displays, makes and updates graphs easily utilizing the Graphite URL api;
- Cubism.js is a D3 plugin for visualizing time series data in real time, and can pull data from Graphite;
- txCarbonClient represents a Twisted API that reports metrics to Carbon;
- structured_metrics is a lightweight python library that uses plugins to read in Graphite’s list of metric names and convert it into a multi-dimensional tag space of clear, sanitized targets.

Graphite is used to create diagrams with numeric values which change after some time. Essentially, a program is composed to gather these numeric data which are then send to Graphite's backend, Carbon.

2. Grafana

In order to query, visualize, alert on and understand the metrics we used Grafana [17], one of the leading open source platform for time series analytics. With Grafana it is possible to design, explore, and share dashboards with the team, because of its data driven characteristics.

Grafana can be used with a wide range of depository backends for the time series data. Every Data Source has a particular Query Editor that is custom-build for the characteristics and abilities that the specific Data Source exposes. The query language and capacities of every Data Source are clearly altogether different. It is possible to join information from numerous Data Sources onto a solitary Dashboard, yet each Panel is attached to a particular Data Source that has a place within a specific Organization. The supported Data Sources are Graphite, Elasticsearch, CloudWatch, InfluxDB, OpenTSDB, Prometheus.

The visualizing feature enable to build quick and adaptable customer side diagrams with a wide number of choices for modules to visualize measurements and logs. In Fig. 3 we present telemetry metrics from Graphite Carbon.

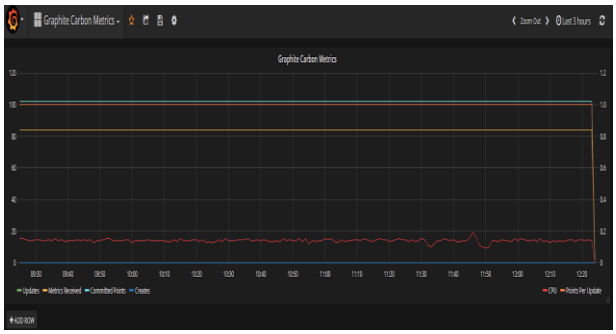

Fig. 3. Visualization of telemetry data using Graphite

Alerting is set up by characterizing alert principles for most critical measurements using the platform, while Grafana will ceaselessly assess them and then, send notices.

At the point when an alert changes state it conveys notifications to email or sending them from Slack, PagerDuty, VictorOps, email, or through webhook.

Also, we used Graphite to make dynamic and reusable dashboards with layout variables which show up as dropdowns at the highest point of the dashboard.

Furthermore, as we used mixed data sources from environmental sensors and performance metrics of the system, Graphite enabled blending distinctive data sources within the same chart. It is possible to indicate a data source on each inquiry premise, which can be used even for custom data sources.

Annotations can be used to Explain charts with rich events from various data sources and float over events displying the full event metadata and labels. Ad-hoc filters allow the setting up of new key/value filters quickly, even while doing something else, that are automatically applied to all inquiries that utilize that data source.

3. Prometheus

Prometheus [18] is an open source solution for monitoring and alerting. Prometheus is quite unique, because it allows the view of metrics information when placing the cursor on a respective line on the graph.

Moreover, Prometheus enables an increased dimensional data pattern, as time series are determined by a metric name and a set of key-value pairs. Segmenting, viewing and understanding of gathered time series data in a database is done by using an adaptable query language, so that tables, ad-hoc diagrams and alerts can be generated. Furthermore, data can be visualized in Prometheus using a multitude of modes: an integrated expression browser, a console template language, and a combination with Grafana.

For efficient storage, time series are saved in memory and on local disk in an effective custom configuration, while functional sharding and federation accomplishes scaling. Every server is relying only on local depository, being independent for reliability and easy orientation. Also, all binaries are statistically connected and simple to deploy.

For alerting, information is maintained related to the dimensions of the data and are described using Prometheus's flexible query language, while notifications and silencing are done by an alert manager.

Multiple client libraries allow easy implantation and simple handling of services, as more than ten languages are already supported. Prometheus accepts bridging of third-party data by exporters and integration with system statistics, Docker, HAProxy, Statsd, and JMX metrics.

B. Monitoring adapter

The monitoring adapter is based on JCatascopia [19], which is written in Java and has no client SDKs for other languages. It contains a server, a database which is currently based on Apache Cassandra and a monitoring agent which loads several probes by default and some extra ones which can be defined by the user - in this case by each organization. The probes are basically small java classes which are loaded by the agent and are used to pull data from the containers.

The current solution, JCatascopia with Java agent and probes, has two properties which makes hard, if not impossible to be integrated with the containers used in SWITCH. First it is implemented in Java and has no client SDKs for other languages. The second is the fact that the agent container needs to be configured with the address of the container it monitors.

Because the probes pull data from containers, they require the address of the container they are supposed to monitor. Also, because the design of the probe is quite static it is very hard to reconfigure the probe or the agent under which it runs to allow multiple containers being monitored by the same probe/agent. This requires instantiating one agent for each container. This wouldn't be a problem if agents could be run in-process or have a small footprint. Unfortunately, the container, as it was designed, requires a lot of memory and a lot of packages to be able to run a JVM.

Because the agent and probe are written in Java, and the probe needs to be run by the agent it is impossible to integrate inside an application written in another language or in a container which runs a software not controlled by users - web server, database server, etc.

To solve the above problems, the following design presented in Fig. 4, was proposed.

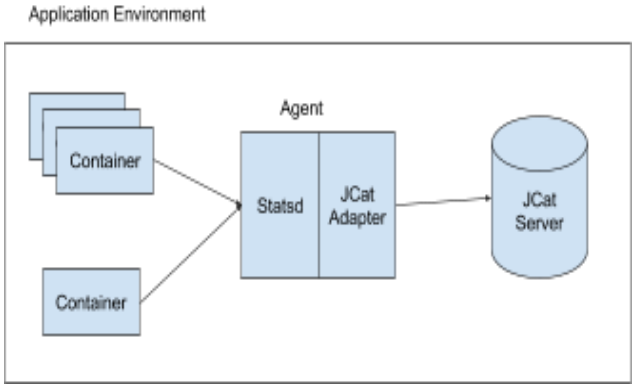

Fig. 4. JCatascopia Agent Adapter design

The main idea of this design is to use an agent that can receive metric data from containers using a better-known protocol with client SDKs available for many programming languages. Statsd is that agent.

This architecture was done with the following criteria in mind:

- Only one agent per environment - a single agent is needed in the environment and can receive data from a large number of containers. Statsd is written in node.js and supports large concurrency.
- Easy to configure containers - because there is only one agent, the container only needs to receive the address of the agent as a parameter and no other configuration needs to happen. The same applies to the agent container which only needs to receive the address of the JCatascopia Server.
- Low footprint, easy to integrate - because there are client SDKs available and plugins for software like Nginx, MySQL, etc., the footprint of including monitoring in any container is very low (usually the order of tens of kilobytes). The clients will run in-process so containers remain simple and there is no need for complicated process management.

JCatascopia requires certain information to be passed for each metric. Because of that a metric key name needs to follow a very strict format: `eu.switch.<application-environment>.<container-id>.<container-ip>.<metric-group-name>.<metric-name>.<units>`

The fields are as follows:

- eu.switch.<application-environment> - this needs to be configured in the Statsd container, as MONITORING_PREFIX. It helps the adapter to filter metrics which are not supposed to be sent to JCatascopia.

- <container-id> - this represents the sending server/VM/container ID. It is suggested to use the hostname generated by docker as a hexadecimal random string. Other options are using a randomly generated string with enough entropy to prevent duplication (eg. UUID)

- <container-ip> - this represents the container IP. Note: Because StatsD keys use dots to separate each part, the IP will be written with dashes replacing of dots. Eg. 127.0.0.1 -> 127-0-0-1

- <metric-group-name> - this represents the metric group or probe name in JCatascopia terms. Note: there are some "reserved" group/probe names and it is advised to use those for the assigned values - StatsInfoProbe, CPUProbe, DiskStatsProbe, NetworkProbe, MemoryProbe.

<metric-name> - this represents the metric name in JCatascopia terms. Note: there are some "reserved" metric names for each probe and it is advised to use those for the assigned values.

- <units> - (optional) this represents the metric units to display in JCatascopia. It can be skipped. In case it is skipped, the dot preceding it needs to be skipped as well.

The StatsD [20] protocol used for collecting telemetry data requires all metric values to be integers. For real numbers, it is suggested to use a power of 10 multiplied by that metric value, up to the required precision.

Moreover, JCatascopia accepts other types than integers, for STRINGs, there is a special protocol that allows sending that value. Instead of the last part of the key - <units> - the string which represents the value for the key will be sent. The string should be cleaned up of non-alphanumeric characters, except dash ("-") and underscore ("_"), which are accepted, before being passed to StatsD. The value will be ignored, so it can be anything.

Because of the way the metrics are processed by StatsD, clients should only need to send the string value once - if they don't need to change it. Of course, sending the value multiple times is also supported, but not required.

Implementing the StatsD protocol should refer to the documentation of the particular client SDK used for information on how to send actual data to StatsD. The only particularity of the implementation for the JCatascopia Adapter is that it processes only counters. Timers, counters, sets and other types of metrics are not supported.

Also, StatsD resets gauges to 0 at each flush interval (i.e. each time they are sent to the backend - JCatascopia). A container sending metrics to JCatascopia through Statsd needs to implement a loop for sending metric data to StatsD.

The monitoring agent is uploaded in Docker Hub at beia/monitoring_adapter. It has two parameters, in the form of environment variables, which can be set:

- MONITORING_SERVER - it should be initialized to the address of the monitoring server (JCatascopia)

- MONITORING_PREFIX - it should be initialized to the prefix of all the metric keys that are to be processed and sent forward by the agent. For example: "eu.switch.beia"

- GRAPHITE_SERVER - (optional) it should be initialized to the address of a Graphite instance which will receive the metrics along with JCatascopia.

- LOGGING_LEVEL - (optional) it should be initialized to one of these values error, warn(default), info, debug, trace. It gives debug information.

The container will fail to start if the two environment variables are not set, as during development, these variables should be set by the developer. In production, in combination with the platform components of SWITCH, these variables should be set and deployed along with the containers specific to ASAP.

To test the adapter and developing own metrics in conjunction with JCatascopia, we used the following steps.

- Step #1: start the container with JCatascopia server using the command:

```
docker run -d --rm -p 8080:8080 -p 4242:4242 -p 4245:4245 salmant/ul_monitoring_server_container_image
```

The host on which the container was started will be called $JCATA_HOST.

For verification it is possible to access http://$JCATA_HOST:8080/JCatascopia-Web/home.jsp and see an UI with some graphics.

- Step #2: start the monitoring_agent with the command:

```
docker run -e MONITORING_SERVER="$JCATA_HOST" -e MONITORING_PREFIX="eu.switch.<beia|mog|wt|anything>" -p 8125:8125/udp beia/monitoring_adapter
```

The MONITORING_SERVER environment variable has to be set in the container to start, as well as the MONITORING_PREFIX variable, which enables communication with the JCatascopia server. The MONITORING_PREFIX should match the value used when sending the metrics, in order for JCatascopia to operate properly.

- Step #3: write code to start sending metrics to JCatascopia using the proper format for metric keys.

If the correct format for metrics are not passed, the agent will just ignore the metrics and the monitoring server will not receive any warning or notification. As StatsD supports many types of data - counters, gauges, timers, etc. - in this adapter only the gauge is implemented.

To validate the successful deployment of the monitoring adapter, we are able to see the agent and its associated metrics, in the above interface.

C. *Notification System*

The notification system is an important subsystem responsible to deliver reliable, scalable, real-time voice notifications through multiple communication means (including VoIP/SIP and landlines) and offers a simple REST API for other modules to trigger and monitor that kind of actions.

The architecture is composed of 3 submodules:

- Asterisk – a software PBX that assures the communication between the Notification System and SIP provides/landlines;
- Notification Workers – the software that implements the reliable interface for triggering and control the processes;
- Redis database – a in-memory database that is used to keep the consistency of call statuses between different notification works.

In our architecture, each Notification Worker consists of 2 modules:

- Flask API which exposes the Web API for communicating with other components;
- Asterisk Worker which handles the asynchronous events related to a call and realizes the communication to the Asterisk PBX.

The number of Notification Workers can be scaled up/down by simply adding a new container with this image. The architecture of Notification System is described as several communicating modules that can be visualized in Fig. 5.

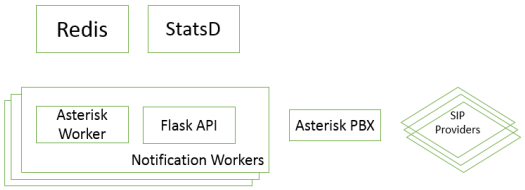

Fig. 5. Notification System Architecture

As we developed our software, each module is a Docker container. For our scaling proposes, the number of Notification Workers can be scaled-up by sample deploying a new container with the given image and different Asterisk PBX containers can get connected to multiple SIP providers.

Having multiple Notification Workers is not in the concern of the user-agent. This aspect is handled by the DNS-based load-balancer in Docker.

The interaction between the user-agent and the Notification System is realized through the REST API exposed by the Flask API module inside the Notification Worker module.

In Table 1, we present the API for originating a voice notification with a prerecorded message and checking the status for this type of call.

TABLE 1 NOTIFICATION SYSTEM API

Originate playback call	
URL	http://<notification_system_host>:<port>/originate
HTTP method	POST
Request Format	JSON
Request parameters	extension – the SIP extension/phone number to be called; file – the name of file to be play-backed; sip_provider – SIP provider to be used.
Example of valid request	{"extension": "0723456789", "file": "tt-goodbye", "sip_provider": "clickphone"}

Example of response	{
	"file": "tt-goodbye",
	"id": "58f477dd-332e-4088-835a-fe6055d032d1",
	"status": "initiated",
	"status_code": 1,
	"timestamp": 1512660726.229323
	}

Status check	
URL	http://<notification_system_host>:<port>/originate
HTTP method	POST
Request Format	No-body
Response format	JSON object with the same format as the one for originate.

For a successful execution, any request sent to this API will response with a 200 OK HTTP status. If any other exception raises, a different status code is provided.

For the most of those exception, additional details are included in the response. We present the format of this kind of response in Table 2.

TABLE 2 NOTIFICATION SYSTEM ERROR RESPONSE FORMAT

Response format	JSON
Example of error response	{"error":{"short_description": "Call not found", "http_code": 404}}

V. INTEGRATION WITH SWITCH SUBSYSTEMS

This section presents the integration of the use case for telemetry in case of emergency situations with SWITCH subsystems SIDE, DRIP and ASAP [21]. In Fig. 6 we present how the application is adapted to dynamic conditions by SWITCH.

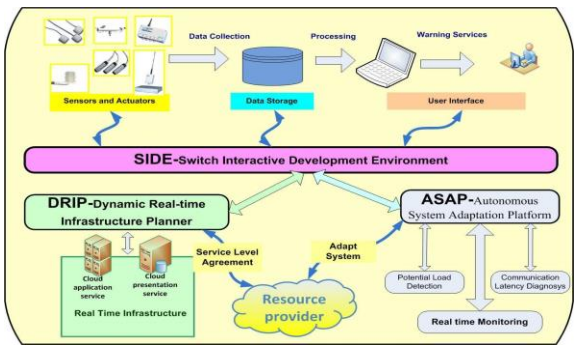

Fig. 6. Adaptability model of disaster early warning use case to SWITCH

A. SIDE

The SIDE (SWITCH Interactive Development Environment) subsystem provides the developer of early warning systems with an intuitive interface to create an application composed out of several services in what is usually considered a micro service architecture. When creating an application each component must first be described and the additional functionalities it requires or contains i.e. Volumes, monitoring, Hardware requirements... This assignment is done by adding nodes to the component that describe the parameters that can be changed or checked. There are several nodes defined at this time, but more can be added in the future.

The components can be connected in application view. At this time the application developer can specify the desired QoS parameters he deems define his components or application. This enables the system to be more flexible, reassigning variables to suit the required application. Based on the nodes added to the components of the application SIDE can add the required SWITCH components, such as Monitoring Server, Monitoring Adaptor or Alarm Tigger.

As this information is collected by SIDE it is transformed into TOSCA description that can be used by other systems, such as DRIP.

B. DRIP

The DRIP (Dynamic Real-Time Infrastructure Planner) subsystem has several capabilities for planning, provisioning and deployment. The first stage, is the Infrastructure planer which takes the information that has been collated by SIDE, specifically the Hardware requirements and the QoS metrics and creates a plan that should satisfy the systems performance.

The second stage is Infrastructure provisioning. It takes the plan as an input and provisions the machines based on that plan and the available credentials. It re turns the information needed to use the VMs, and starts them on a given cloud provider

The last stage, deployment, can be broken down into two parts. First a SWARM cluster is created on the available infrastructure. Then the deployment takes a TOSCA file as an input and transforms it into a Docker deploy file that can be run on the cluster.

C. ASAP

The ASAP (Autonomous System Adaptation Platform) sub-system is split into several parts. The monitoring, described in above, starts working after the application has been deployed. And the results can be seen in SIDE as a graph of available system or application metrics. Noting this the developer can change the configuration of the system or the infrastructure he is using and redeploy the application to suit his needs.

Additionally, the developer can specify the parameters of the Alarm trigger for each application component. This information is stored in SIDE and send to the Alarm Trigger component. The Alarm Trigger parses this information, compares it to the current state of the system based on monitoring and can, if the state does not meet the requirements, trigger an alarm or a direct action from the system via the Self Adaptor component, thus safeguarding the system form failure.

VI. RESULTS OF SWITCH

All the code and configuration are currently contained in a git repository. The repository contains all the software and instructions (Docker files, Docker Compose configurations, config files) necessary to build the whole infrastructure for the project.

Currently all the components in both SWITCH and BEIA's use-case are contained as Docker images and we started the integrating our components with SWITCH platform as following:

- For the integration with SIDE component we experimented the creation of a TOSCA file for our software components and validated the possibility of describing the requirements of our use-case using SIDE;
- For the integration with DRIP we assured that each of our components is individually contained as a Docker image and tested that the environment offered by the deployment platform allows our components to run properly;
- For the integration with ASAP we included the Monitoring Server in our deployment and successfully collected some relevant for scaling metrics from some of our components.

A. Current status

We have a stack that is usually called LEMP (Linux + Nginx + MariaDB/MySQL + PHP-FPM), but in the context of our architecture could be called LESP (Linux + Nginx + Supervisor + PHP-FPM) + LM (Linux + MariaDB/MySQL). It is composed of two containers. The first container maps to the Notification server in the architecture and is running Nginx and php5-fpm under Supervisord. Nginx and php5-fpm communicate through a Unix socket, for better performance, which forces them to run under the same container. Supervisord is a process manager that is required because Docker allows only one entry point per container. Supervisord has the added benefit of being able to restart the processes in case they die and being able to manage each process independently. The second container maps to the Database server in the architecture and is running MySQL with a persistent volume for the database.

All the components of the telemetry system represent one module in the architecture. The LEMP stack has smaller memory footprint and better performance. Because of this, PHP-FPM + Nginx is the recommended way of running PHP.

B. Existing components

The telemetry system consists of a database and gateway component for interfacing with proprietary sensors and communication protocols (GPRS, UHF). The features implemented in the current telemetry software do not provide the level of flexibility and detail required, so a simple dashboard implemented in PHP was developed. To be able to produce alerts on sensor data and specific abnormal behaviors, a rule engine that takes the data from sensors and alerts other components down the communication path is included.

Communication with sensors is done over an XML protocol, called addUPI, that allows applications to iterate sensors and nodes - platforms containing multiple sensors, and also gathering sensor data. The telemetry component is polling the IP gateway and sensors, using a cron script, and deposits the data into a MySQL database. The same database contains definitions for the dashboards, user logins and alert definitions.

The telemetry component is served through a Nginx HTTP server which uses PHP-FPM as an application container. PHP-FPM has multiple features which are designed to optimize for speed and memory usage. Most importantly, PHP-FPM contains an opcode cache which allows PHP scripts to be interpreted just once and executed multiple times. Also, PHP-FPM is using a worker pool to optimize client access and prevent spawning many temporary PHP processes. Nginx also has features that allow serving many clients without great memory consumption and CPU usage.

The PHP application architecture is shown in Fig. 7.

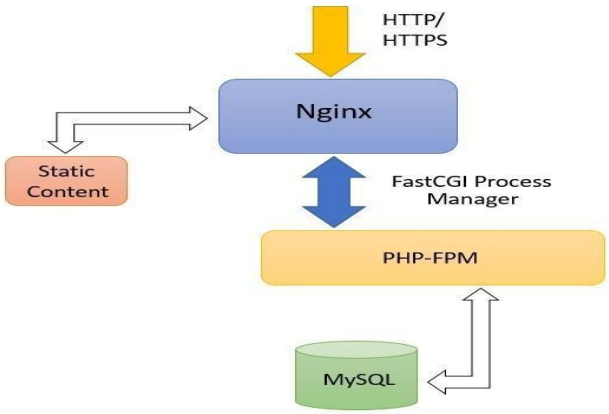

Fig. 7. NGINX + PHP-FPM architecture

The two containers are connected using Docker Compose, which allows setting a separate network for the communication between the two containers, adding volumes and creating configurations that include environment variables.

VII. Conclusion

In summary, the paper analyzed related work for telemetry system used in emergency situations and the functional description of the pilot case was created, associated with usage scenarios (players, actions, equipment) and overall requirements. Also, an architecture indicating the main components of the solution, the associated services and interfaces and the technologies that are used in the implementation phase together with evaluation metrics are presented. An early prototype / proof of concept for the use case with all the main components implemented (remote telemetry units, IP gateway, database server, notification server) and with some known issues (portability of Windows images, adaptation of applications running in virtual machines to containers) has been designed. As future work we envision to measure performance metrics for scaling of different components of the proposed use case for disaster early warning.

Acknowledgment

This project has been funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 643963 (SWITCH project).

References

[1] G. Buribayeva, T. Miyachi, A. Yeshmukhametov, and Y. Mikami, "An Autonomous Emergency Warning System based on Cloud Servers and SNS," Procedia Computer Science, no. 60, pp. 722 – 729, 2015.
[2] D. K. Krishnappa, E. Lyons, D. Irwin, and M. Zink, "Compute cloud based weather detection and warning system," Geoscience and Remote Sensing Symposium (IGARSS), IEEE International, University of Massachusetts Amherst, pp. 2430-2433, 2012.
[3] Jie Li, M. Humphrey, D. Agarwal, K. Jackson, C. van Ingen, and Youngryel Ryu, "eScience in the Cloud: A MODIS Satellite Data Reprojection and Reduction Pipeline in the Windows Azure Platform," In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1-10, 2010.
[4] J. J. Rehr, J. P. Gardner, M. Prange, L. Svec, and F. Vila, "Scientific Computing in the Cloud," In Computing in science & Engineering, vol. 12, no. 3, pp. 34-43, 2010.
[5] K. S. S. Rani, and N. Indhumathi, "An Efficient Modern Irrigation and Plant Growth Monitoring System using Sensor Network," Asian Journal of Research in Social Sciences and Humanities, no. 8, vol. 6, pp. 350-359, 2016.
[6] LIBELIUM: Early warning system to prevent floods and allow disaster management in Colombian rivers http://www.libelium.com/early-warning-system-to-prevent-floods-and-allow-disaster-management-in-colombian-rivers/.
[7] J. Zschau, and A.N. Küppers, "Early warning systems for natural disaster reduction," Springer Science & Business Media, November 2013.
[8] T. Glade, and F. Nadim, "Early warning systems for natural hazards and risks," Natural Hazards, vol. 70, no. 3, pp. 1669- 1671, 2014.
[9] J. W. de Groot, and D. M. Flannigan, "Climate change and early warning systems for wildland fire," Reducing Disaster: Early Warning Systems for Climate Change, Springer, pp. 127-151, 2014.
[10] F. E. Horita, J. P. de Albuquerque, V. Marchezini, and E. M. Mendiondo. "A qualitative analysis of the early warning process in disaster management," In Proceedings of the 13th International Conference on Information Systems for Crisis Response and Management (ISCRAM), pp. 1-9, 2016.
[11] J. Cools, D. Innocenti, and S. O'Brien. "Lessons from flood early warning systems," Environmental Science & Policy, vol. 58, pp. 117-122, 2016.
[12] M. Takemoto, K. Koizumi, Y. Fujiwara, H. Morishita, and K. Oda, "Improvement of a slope disaster warning system for practical use," Japanese Geotechnical Society Special Publication 2, no. 3, pp. 196-200, 2016.
[13] A. Alhmoudi, and Z. U. H. Aziz. "Integrated framework for early warning system in UAE," International Journal of Disaster Resilience in the Built Environment vol. 7, no. 4, pp. 361-373, 2016.
[14] M. Arcorace, F. Silvestro, R. Rudari, G. Boni, L. Dell'Oro, and E. Bjorgo, "Forecast-based Integrated Flood Detection System for Emergency Response and Disaster Risk Reduction (Flood-FINDER)," EGU General Assembly Conference Abstracts, vol. 18, pp. 8770-8774, 2016.
[15] J. Udo, and N. Jungermann, "Early Warning System Ghana: how to successfully implement a disaster early warning system in a data scarce region," EGU General Assembly Conference Abstracts, vol. 18, pp. 12819-12823, 2016.
[16] GRAPHITE Documentation https://graphite.readthedocs.io/en/latest/
[17] GRAFANA: The open platform for beautiful analytics and monitoring https://grafana.com/
[18] PROMETHEUS: From metrics to insight https://prometheus.io/
[19] S. Taherizadeh, and V. Stankovski. "Incremental Learning from Multi-level Monitoring Data and Its Application to Component Based Software Engineering." In Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual, vol. 2, pp. 378-383, 2017
[20] M. Miglierina, and D. A. Tamburri. "Towards Omnia: A Monitoring Factory for Quality-Aware DevOps." In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, pp. 145-150. ACM, 2017.
[21] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses, and C. de Laat, "Developing and operating time critical applications in clouds: the state of the art and the SWITCH approach". Procedia Computer Science, vol. 68, 2015, pp.17-28.