2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing

Robustness Indicators for Cloud-based Systems Topologies

Franck Chauvel, Hui Song, Nicolas Ferry, Franck Fleurey
SINTEF ICT
Oslo, Norway
{name.surname}@sintef.no

***Abstract*—Various services are now available in the Cloud, ranging from turnkey databases and application servers to high-level services such as continuous integration or source version control. To stand out of this diversity, *robustness* of service compositions is an important selling argument, but which remains difficult to understand and estimate as it does not only depend on services but also on the underlying platform and infrastructure. Yet, choosing a specific service composition may fail to deliver the expected robustness, but reverting early choices may jeopardise the success of any Cloud project.**

Inspired by existing models used in Biology to quantify the robustness of ecosystems, we show how to tailor them to obtain early indicators of robustness for cloud-based deployments. This technique helps identify weakest services in the overall architecture and in turn mitigates the risk of having to revert key architectural choices. We illustrate our approach by comparing the robustness of four alternative deployments of the SensApp application, which includes a MongoDB database, four REST services and a graphical web-front end.

I. Introduction

Cloud is very dynamic: new services are continuously made available, whereas less useful ones get rapidly dismissed. Services include end-user applications (e.g., weather forecast), platform services (e.g., Heroku, Google App Engine), and infrastructure services (e.g., Amazon EC2, Flexiant). In this dynamic environment, cloud-based systems are continuously refined to make the most of new opportunities.

Among others, *robustness* is a key selling argument for many cloud services and applications, and is included in many service-layer agreements (SLA). Yet, since cloud-applications compose and re-compose services from all cloud levels (i.e., applications, platform and infrastructure) it is critical to understand and estimate the impact that alternative architecture decisions may have on robustness. This is critical during the whole life of the system: as little data about newly discovered services may be available to accurately estimate robustness of possible designs. The risk is that any decision that later turns out irrelevant, be either very expensive to revert, or become a heavy *technical debt* [1] for the developers.

Robustness indicators are commonly accepted as a means to mitigate this risk through the development and differentiate as soon as possible candidate architectures. Qualitative indicators, based on subjective expert judgements, are always available but are extremely time consuming to consolidate. By contrast quantitative indicators such as Mean-time-to-failure (MTTF) result from complex simulations that require a detailed technical knowledge which may not be available. The fast pace of the Cloud calls for rapid feedback regarding robustness of the system, despite the complexity of robustness evaluation.

Our contribution is a set of three robustness indicators, tailored for cloud-based systems together, which require minimal knowledge of the systems and do not involve expert judgements. In addition, we provide Trio, an experimental tool to compute them on cloud-topologies.

Our indicators are inspired by the robustness metrics used for ecosystems in Biology. They are based on an analogy between species extinctions in ecosystems and component failures in software systems. Based on this, one can not only compare the overall robustness (1) of alternative architectures, but also spot the weakest components (2) as well as the most threatening failure sequences (3). We illustrate how our indicators permit to rank alternative architectures of the SensApp application, which includes a database, four REST services and a graphical front-end.

The remainder is organised as follows. Section II introduces a running example of Cloud-based system, where various deployment architectures may be envisioned. Section III first introduces how robustness is understood and estimated in Ecology. Section IV gives an overview of the robustness indicators that we propose and discusses the minimal required inputs. Section V details their calculation and explain how the models used in Biology are tailored to Cloud-based systems. Section VI presents our prototype implementation and how we used it to select robust deployments of SensApp. Finally, Section VII discusses a selection of related work before Section VIII concludes and presents our future research directions.

II. Motivating Example

SensApp [2] is the typical cloud-based application that we use as a running example hereafter. SensApp acts as a buffer between sensor networks and cloud-based systems. On one hand, it gives sensors a place to continuously push data while, on the other hand, it provides higher level services with notification and query facilities.

As a service-oriented architecture (SOA), SensApp is made of four REST services that intercept raw data coming from sensors, store it in a database (e.g., MongoDB) and notify interested users eventually. In addition, SensApp provides a

978-1-4799-7881-6/14 $31.00 © 2014 IEEE
DOI

graphical web interface called *SensApp Admin*, which helps administrators manage the underlying database. Fig. 1 summarises these high-level components and their interactions.

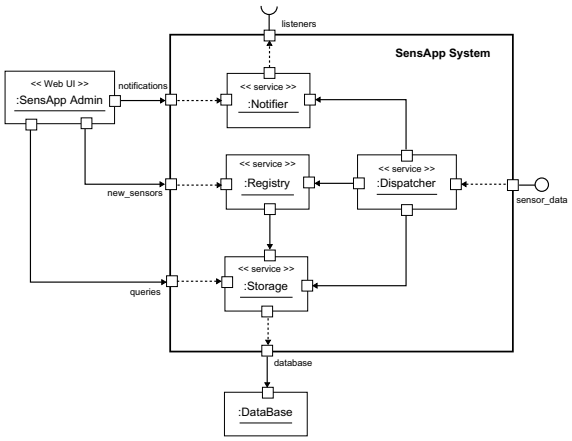

Figure 1. Decomposition of SensApp into high-level functional components, depicted as a UML 2 component diagram

We are interested in the robustness of SensApp, which is commonly understood as its *ability to cope with failures*, and which can intuitively be improved using two main strategies: *replication* and *isolation*. In case of failures, replication leverages backup components whereas isolation minimises failure propagation. In practise, good isolation enables efficient replication, as it permits to only replicate the most critical parts. Breaking down SensApp into six separate services improves isolation by restricting how failures can propagate (which is not the case in a monolithic implementation) and makes possible to replicate the database if needed.

However, robustness in Cloud-based system must also take into account the execution environment, as environment failures eventually propagate to the services. Identifying the environment for any Cloud-application requires answering the three following questions to gain insight regarding platform, infrastructure and allocation.

- What **platform**? Web services are mainly SOAP-based or REST-based and both can be built on many technologies such as Java Enterprise, .NET, Ruby, to name a few. These environments directly impact the eventual robustness of the system.
- What **infrastructure**? Most environments are provided as services (e.g., Heroku, GoogleApp Engine, Cloudbees) but can also be manually installed and configured on public or private virtual machines (e.g., AWS EC2, OpenStack). The size, type and features (e.g., replication, scaling) of the infrastructure further impact the eventual robustness.
- Which **allocation scheme**? Finally, the way the application components are deployed on the platform, and the way the platform is deployed on the infrastructure reflect the use of isolation and replication, and in turn, affect the eventual robustness.

For SensApp, the development team selected Java as an execution environment. Services thus have to run on a servlet container (SC) such as Jetty or Tomcat, which requires a Java Runtime Environment (JRE) to run properly on any given vitrual machine (VM).

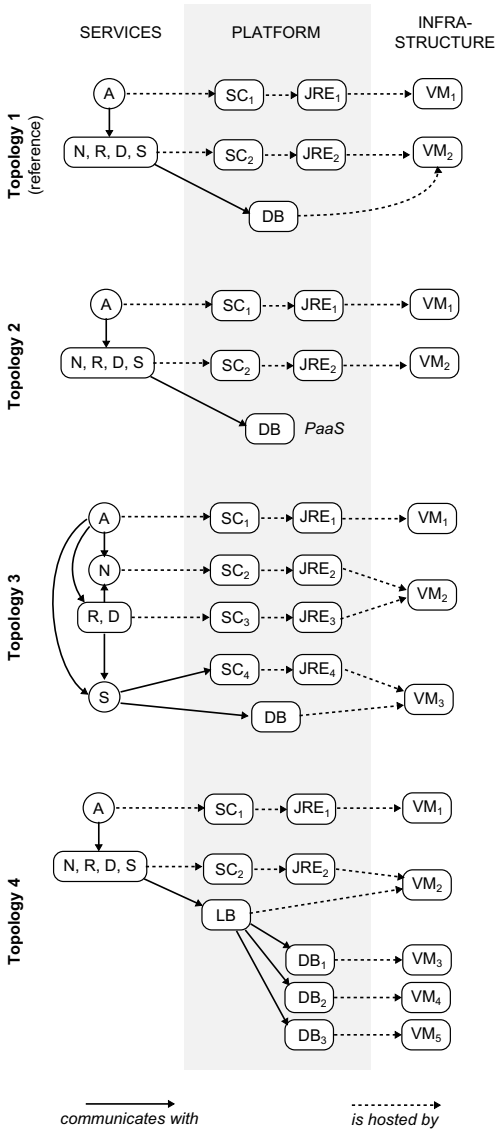

Figure 2. Four alternatives deployment topologies of the SensApp system

Fig. 2 illustrates four possible deployment topologies for SensApp, representing alternative answers to the above three questions. In the reference topology, we envisioned to isolate SensApp Admin (A) on a separate machine, running within its own servlet container. In the second topology, the database (DB) is outsourced to a platform-as-a-service provider (PaaS) to increase isolation at the platform level. In the third topology, we further isolated each SensApp service in a separate servlet

container. However, at the infrastructure level, the container of the dispatcher and the registry are still running on the same virtual machine (VM). Finally, in the fourth topology, we included an additional load balancer (LB) in front a replicated database (DB). Other topologies are possible, but these four ones reflect our experience in deploying SensApp in projects such as ENVISION [3] or ENVIROFY [4].

Getting insight regarding the robustness of the SensApp services in such topologies is critical and difficult, especially in continuous development. Development teams have to rely on *indicators* to discard the less relevant one. However, accurate qualitative analysis, using realistic performance models is not practical due to the lack of detailed knowledge about the implementation of the system. On the other hand, qualitative indicators relying on subjective expert judgement is extremely time consuming to gather and consolidate in practise. Last but not least, existing indicators provides the robustness of the whole system, while architects are looking for the robustness of the services running in alternative environments.

In the following, we present three robustness indicators, inspired by an approach used in Biology for evaluating the robustness of ecosystems, which does not require any subjective judgement nor any deep technical knowledge.

III. Robustness of Ecosystems

Ecology is the branch of Biology that focuses on the interactions between organisms or groups of organisms and their environment. Ecology portrays these interactions as networks of species, called *ecosystems*, where dependencies between species generally capture prey-predator relationships. The intuition is that a species whose food or resources are not anymore available will starve and die consequently. In the ecosystem depicted in Fig. 3 below, if both grasses and carrots disappear, rabbits will starve and disappear as well. Ecosystems are often partitioned into *trophic levels* which gather species with similar properties, such as the "resources" or the "top predators" in Fig. 3.

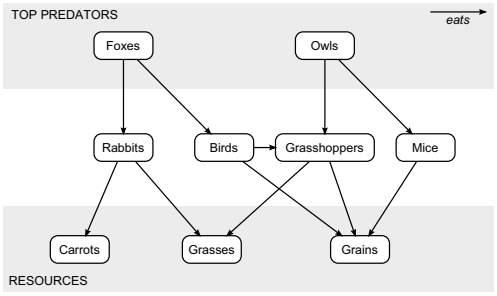

Figure 3. Illustrative ecosystem capturing the prey-predator relationships between a small set of species

Ecologists distinguish between the robustness of the whole ecosystem and the robustness of a particular trophic level. In both cases, robustness is understood as the capacity of the ecosystem (or part of it) to survive to the extinction of species and is measured by simulating *extinction sequences* [5]. During one extinction sequence, one extinguishes the species one after the other while measuring how many survive at each step. For instance, Fig. 4 shows the response of the top predator group to the extinction sequence "grasses, grains and carrots". The robustness of the top predators to this particular sequence is given by the grey area. The overall robustness of the top predators is the average robustness over all possible extinction sequences.

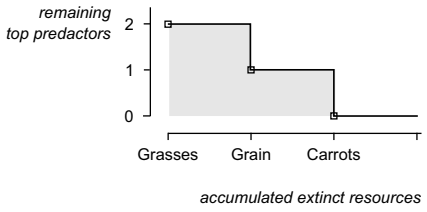

Figure 4. Response of the "top predators" group to the extinction sequence "grasses, grains and carrots" (cf. Fig. 3)

It is worth to note that a high robustness at the ecosystem-level does not necessarily imply the absence of brittle species or groups of species. In Fig. 3, mice is a sensitive species, which goes extinct as soon as grains disappear.

We show in the following how the notion of *extinction sequences* can be applied on Cloud topology to reflect their robustness.

IV. Overview

As shown by Fig. 5, the calculation of our robustness indicators requires two inputs, which can be obtained in the earliest stages of the development process.

- **A deployment topology** which describes the various components of the system of interest. In a Cloud setting, the key point is to categorise the deployment by identifying the underlying infrastructure including application servers, middleware components and virtual machines.
- **A failure propagation model** which describes how components' failures propagate through the system. This failure propagation model is a set of propositional logic formulae specifying the necessary conditions for each component to fail (with respect to its dependencies). If the topology distinguishes communication between components from hosting (as we did on Fig. 2), default formulae can be generated, as we assume that a component fails as soon as it misses any of its dependencies or its host.

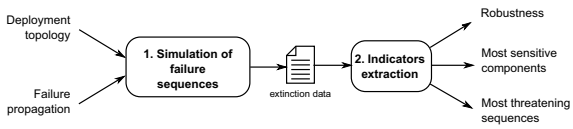

Figure 5. Overview of the robustness indicators computation process

From the data generated during the simulation of failure sequences, three main indicators are derived:

- a *robustness indicator* as a value in [0, 1] reflecting the impact that failures in a subsystem X have on another subsystem Y. A robustness of 0.2 means that a failure of 20 % of the subsystem X takes down, in average, 80 % of the subsystem Y. Similarly, a robustness of 0 means that any single failure in X, annihilates the subsystem Y. Conversely, a robustness of 1 means that the two subsystems are completely isolated and that failures cannot propagate from one to the other.
- the *most sensitive components*, regardless of their role in the architecture (application, platform or infrastructure) are the components whose local failure brings down the most significant part of the topology.
- the most *threatening failures sequences*, describing the most probable ordering of failures with a strong impact.

We present below how we calculate these three robustness indicators, and the minimal inputs required to compute them.

V. Cloud Robustness Indicators

A. Modelling Cloud Topologies

Our model of cloud topologies is built on an analogy between species extinctions and software failures. Species extinctions, which might trigger other species extinctions, can be seen as failures of the software components, which might similarly propagate to other components. Indeed, our model relies on two main concepts: components and the relationships between them.

As for species in ecosystems, the internals of a software component are abstracted: only the relationships between components matter. We focus on two kinds of relationships, especially:

- **Communication** between components represents the fact that a given component requests or triggers a computation by another one. In practise, such communication is either a local invocation or a remote procedure call (RPC).
- **Execution** represents the fact that one component is the execution platform of another one. Example of such relationships are application servers such as Tomcat or Jetty, which are required to *execute* any Java servlet components.

Fig. 2 in Section II denotes these two types of relationships, as plain and dashed lines, respectively. The core SensApp services communicate among each others, but are executed (i.e., hosted) by other components at the platform level. For instance in Topology 4, the Admin component A communicates with the other services N, R, D, S, and is executed by a servlet container SC_1.

Formally, we define a topology as a set of components which can be either active or failed. We model the state of a topology including k components, as a state vector such as:

$$\mathbf{s} = (s_1, s_2, \ldots, s_k) \text{ where } s_i \in \{0, 1\} \quad (1)$$

Here 1 represents the activity of component s_i, whereas 0 indicates it has failed. The overall activity level of the topology, is denoted by α and given by $\alpha(\mathbf{s}) = \sum_{i=1}^{k} s_i$. Similarly, we model any failure in the system $\mathbf{s}$ as a failure vector :

$$\mathbf{f} = (f_1, f_2, \ldots, f_k) \text{ where } \begin{cases} f_i = 0 & \text{if } s_i \text{ fails} \\ f_i = 1 & \text{otherwise} \end{cases} \quad (2)$$

The local effect of a failure vector $\mathbf{f}$ (without propagation) is a new state vector $\mathbf{s}' = \mathbf{s} \wedge \mathbf{f}$.

B. Failure Propagation

The limit of our analogy with Ecology lies in the fact that whereas a species can theoretically survive as long as there is at least one species on which it can feed, the propagation of failure varies from software to software. Some components fail as soon as one of their dependencies failed, other may be more resilient and stand the failure of some of their dependencies.

To this end, we include in each component, a *propositional logic* formula specifying the conditions under which a component fails due to missing dependencies. In Topology 4 depicted below on Fig. 6, the formulae used for the Admin component A and the load-balancer LB are:

$$\begin{array}{lll} A & \texttt{requires} & SC_1 \wedge N \wedge R \wedge D \wedge S \\ LB & \texttt{requires} & VM_2 \wedge (DB_1 \vee DB_2 \vee DB_3) \end{array}$$

The Admin component thus fails as soon as it misses any of its dependencies (logical conjunction). By contrast, the load-balancer (LB) fails if the underlying platform fails (VM_2) and if the all of the back-end database (DB_1, DB_2, and DB_3) failed.

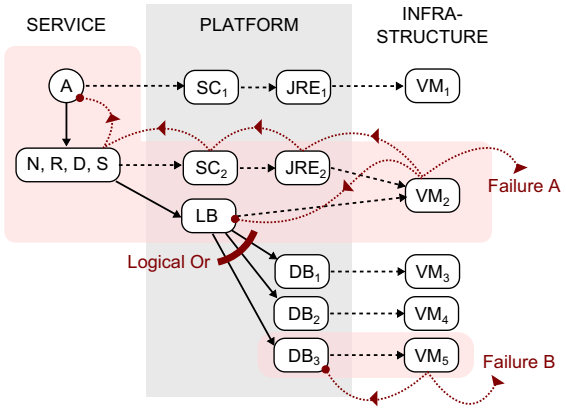

Figure 6. Propagation of two separate failures throughout Topology no. 4

Fig. 6 illustrates the propagation of two failures on Topology 4 (see also Fig. 2). Failure A, on the upper part, first hits VM_2, which indirectly supports the execution of the core services. Failure A thus propagates in turn to JRE_2, to SC_2,

to the group N, R, D, S and eventually reaches the Admin component A. This propagation scheme is due to the fact, that the components requires the other. By contrast, Failure B, on the lower part, does not propagate beyond the load balancer (LB), as it only requires that one of its dependencies be running. Failure B does takes down DB_3, but DB_1 and DB_2 cover for it.

Formally, we represent the set of rules that govern the propagation of failures from one component to its direct neighbours, as a propagation vector $\mathbf{p}$, where each function p_i represents the conditions under which the component s_i fails, such as:

$$\mathbf{p}(\mathbf{s}) = (p_1, p_2, \ldots, p_k) \text{ where } p_i : \{0,1\}^k \to \{0,1\} \quad (3)$$

The propagation of failures throughout the whole system is therefore defined by the iterated function $\mathbf{p}$.

$$\mathbf{p}_\infty(\mathbf{s}) = \lim_{n\to\infty} \overbrace{\mathbf{p}(\mathbf{p}(\ldots\mathbf{p}(\mathbf{s})))}^{\text{n times}} \quad (4)$$

Note that the propagation of failures converges, under the assumption that each local formula is the conjunction of the current state of the system and of constraints on the other components. In other words, each function p_i is of the form $p_i(\mathbf{s}) = s_i \wedge h(\mathbf{s})$, where $h(\mathbf{s})$ is the necessary environment for s_i to be active. The interested reader can found a detailed proof of the convergence in appendix.

The number of components indirectly taken down by a failure $\mathbf{f}$, so called impact and denoted I can thus be obtained by: $I(\mathbf{s}, \mathbf{f}) = \alpha(\mathbf{s}) - \alpha(\mathbf{p}_\infty(\mathbf{s} \wedge \mathbf{f}))$.

C. Robustness to Specific Failure Sequence

As ecologists, who study the impact of extinction sequences, we propose to study the impact of failure sequences.

In Topology 4 (see Fig. 6), we are interested in the impact that failures occurring in the infrastructure have on the service layer. We thus gradually fail components from the infrastructure (i.e., VM_1, VM_2, VM_3 or VM_4), and we observe how the components at the service level survive this sequence of failure. Let us consider the sequence $\varphi = (VM_1, VM_3, VM_4, VM_2, VM_5)$. The failure of VM_1 propagates until the Admin component A. By contrast, the two subsequent failures of VM_3 and VM_4 do not impact the service layer as they are contained behind the load-balancer LB. It is the failure of the VM_2, which hosts the core services of SensApp, that eventually annihilates the service layer completely. The impact of this sequence of failures is illustrated by Fig. 7 below.

The robustness of Topology 4 to this failure sequence $r(\mathbf{s}, \varphi)$ is $5 + 4 + 4 + 4 = 17$. In the ideal case (see Fig. 7), no service would have been impacted at all and the robustness would have been $5 \times 6 = 30$. By contrast, in the worst case (see Fig. 7), all services would have failed with VM_1 and the robustness would have been 5. As this robustness indicator varies according to the number of components in the layers of interest (i.e., services and infrastructure), we normalise it and obtain $r(\mathbf{s}, \varphi) = 0.48$.

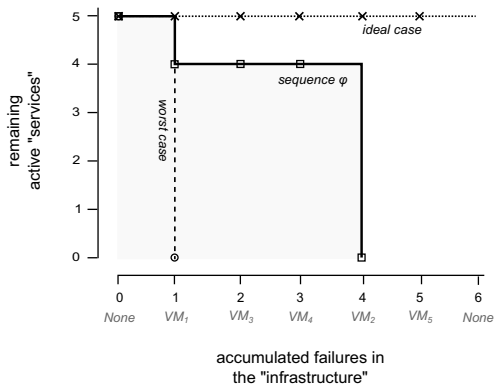

Figure 7. The failure sequence $\varphi = (VM_1, VM_3, VM_4, VM_2, VM_5)$ and its impact on the service layer in Topology 4

Given an initial state vector $\mathbf{s}$ and a sequence of failure vectors $\varphi = (\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n)$, we recursively define the associated sequence of state vectors σ as:

$$\sigma_{n+1} = \mathbf{p}_\infty(\sigma_n \wedge \varphi_n) \qquad \text{with } \sigma_0 = \mathbf{s} \quad (5)$$

Finally the robustness r of a topology in state $\mathbf{s}$ to a particular failure sequence φ is given by the formula below.

$$r(\mathbf{s}, \varphi) = \sum_{j=1}^{n} \alpha(\sigma_j) \quad (6)$$

For the sake of comparison, the robustness values we will show hereafter are all scaled on the unit interval $[0, 1]$, with respect to the maximum and minimum robustness.

Note that all failure sequences are not equally probable, despite the fact that we do not make any assumption on the individual failure probability of each component. In Fig. 7 for instance, assuming that each VM is selected randomly from the infrastructure level, the probability to select the sequence φ is $\mathbb{P}[\varphi] = \frac{1}{5} \cdot \frac{1}{4} \cdot \frac{1}{3} \cdot \frac{1}{2} = 8.33 \times 10^{-3}$. In this particular case, all possible failure sequences at the infrastructure level are equally probable, as there is no dependencies between the components at this level. Note that this is not the case at the platform and services levels.

D. Indicators of Interest

Given a topology, the concept of failure sequence permits to generate three indicators, namely the overall robustness, its most sensitive components and the most threatening sequences.

a) Overall robustness: We define the overall robustness as the impact that all possible failure sequences might have, in average, on the system. Let us consider the *random experiment* of observing any random failure sequence ϕ in a given topology. The set of possible outcomes, denoted Φ, is thus the space of all possible sequences. We define the random variable $R(\phi)$ that associates its robustness to each possible sequences such as $R(\phi) = r(\mathbf{s}, \phi)$. We then define the *overall*

robustness of the topology in state $\mathbf{s}$, denoted as $\mathscr{R}(\mathbf{s})$ as the expected value of the random variable R, such as:

$$\mathscr{R}(\mathbf{s}) = \mathbb{E}[R] = \sum_{\varphi \in \Phi} R(\varphi) \cdot \mathbb{P}[\phi = \varphi] \tag{7}$$

b) Sensitive Components: We define the most sensitive component as the one whose failure $\mathbf{f}_\top$ has the highest average impact regardless of the state in which it occurs as shown below:

$$\mathbf{f}_\top = \max_{\mathbf{f}} \left(\sum_{\mathbf{s} \in S} I(\mathbf{s}, \mathbf{f}) \right) \tag{8}$$

c) Threatening Sequences: By combining the robustness of each sequence with its probability to occur, we identify the associated threat level, as the product of these two values. The most threatening sequence $\varphi_\top$ is thus the sequence with the highest threat level, as formalised below:

$$\varphi_\top = \max_{\varphi} (1 - R(\varphi) \cdot \mathbb{P}[\phi = \varphi]) \tag{9}$$

In the following, we show how these three indicators help identifying a more robust deployment topology for the SensApp application.

VI. Running Examples

A. Prototype Implementation

We developed the *topology robustness indicator* (Trio), a prototype that simulates failure sequences. Trio provides a domain- specific language to evaluate the robustness of various topologies. The Trio description of the topology presented by Fig. 1 is given by Listing 1.

```
system: 'SensApp'

components:
  - Admin requires Notifier and Registry and Storage
  - Notifier
  - Registry requires Storage
  - Dispatcher requires Notifier and Registry and Storage
  - Storage requires DB
  - DB

tags:
  - 'platform' on DB
  - 'service' on Admin, Registry, Storage, Dispatcher,
      Notifier
```

Listing 1. SensApp described in Trio (see in Fig. 1)

Our prototype is an open source Java application, whose source code is available at https://github.com/fchauvel/trio. The robustness evaluator as well as the samples topologies studied in the following can be downloaded at https://github.com/fchauvel/trio/releases.

Trio topologies can automatically be derived from deployment models and SensApp was initially deployed using CloudML [6], [7] for instance. CloudML is a domain-specific language to capture the deployment of cloud-based applications including "communication" and "execution" relationship between components. The strength of CloudML is that the description of the deployment is causally connected to the real running system: any change on the description can be enacted on demand on the system, and, conversely, any change occurring in the system is automatically reflected in the model. Although our prototype is is derived from the CloudML concepts, our indicators can be computed for any deployment models capturing communication and execution.

In the experiment results presented below, we ran Trio against our SensApp example. For each topology, we evaluated the service layer as a separate system, the service layer w.r.t., failures in the infrastructure, and w.r.t., failures in the platform layer. Each time, Trio simulated 10 000 random failure sequences to gain statistical evidence.

B. Overall Robustness

The first indicator is the overall robustness, which reflects to which extent the system is able to stand failure sequences. Table I summarises several robustness, depending on the layers where failures are observed and and where they are injected respectively. Column 2, denoted as "overall" contains the robustness for the whole system. By contrast, the other two columns display indicators measured when failures occur either at the infrastructure or platform levels and that their effect is *only* observed at the service level.

	Robustness of X to failure in Y		
Topology	overall	service/infra	service/platform
Topology 1	0.1427	0.0135	0.0239
Topology 2	0.1507	0.0131	0.0248
Topology 3	0.1559	0.0090	0.0148
Topology 4	0.1899	0.0246	0.0182
SensApp (Fig. 1)	0.1488	n/a	n/a

Table I
Robustness indicators computed for the four topologies of Fig. 2

As one can see in Column *overall*, Topologies 2, 3 and 4 introduced in Fig, 2 all increase the overall robustness (w.r.t., Topology 1). The robustness indicator reflects both the dependencies and the number of components in the system.

Topology 2, illustrates the benefits of removing dependencies. Outsourcing the DB to a PaaS service, improved the robustness because it protects against failures of VM_2, and in turn, reduced its impact.

Topology 3 illustrates the addition of components. Using separate platforms inherently increases the overall robustness, as more failure are needed to get the system completely down. For instance, though only 8 components survive the failure of VM_1 in Topology 1, 13 survive in Topology 3.

Topology 4 illustrates the benefits of components which are inherently more robust to failure. Here the load-balancer fails only if all its back-end DB have failed already.

C. Relative Robustness

Robustness indicators can also be calculated with respect to specific subsystems. Column 3 and 4 of Table I denotes the

robustness of the service layer only, when failures occur at the infrastructure or at the platform level, respectively.

Here, whereas all topologies improved the robustness of the overall system, not all topologies increase the robustness of the service layer. Compared to Topology 2, Topology 3, for instance decreases the services robustness to failures in the infrastructure ($0.0090 < 0.01310$) and to failure in the platform ($0.0148 < 0.0248$).

In Topology 2, two components might fail at the infrastructure level: VM_1 and VM_2. Without additional information, both failures are equally probable, but their impacts are significantly different. Four services survive a failure of VM_1 whereas no service survive a failure VM_2. Statistically, there is 50 % chance that a large part (i.e., 80 %) of the system survives.

By contrast, in Topology 3, three components might fail at the infrastructure level. Four components will survive a failure of VM_1, one will survive a failure of VM_2 and one will survive a failure of VM_3. Statistically, there is now only 33 % chances that a large part (i.e., 80 %) of the system survives. The service layer of Topology 3 is thus less robust to failure in the infrastructure than the one of Topology 2.

This is due to the fact the robustness indicator is an *expected value* that combines the impact of each failure sequence with its probability. By contrast, the general intuition only reflects how significant are the impact of a failure, and overlooks the fact that they might be extremely rare.

D. Sensitive Components

The second indicator provided by Trio is the identification of the most sensitive components, that is two say, the component whose failure brings down the largest part of the system. As for the robustness indicator, this information is relative to the subparts of the system where the failure are injected and where their impact is observed.

	Most sensitive components		
Topology	overall	service/infra	service/platform
Topology 1	VM_2	VM_2	DB
Topology 2	VM_2	VM_2	DB
Topology 3	VM_2	VM_2	JRE_4
Topology 4	VM_2	VM_2	SC_2
Services (Fig. 1)	DB	n/a	n/a

Table II
MOST SENSITIVE COMPONENTS OF THE FOUR SENSAPP TOPOLOGIES

Table II only presents the most sensitive components in each situation, although the tool does provide a complete ranking.

Regarding the whole topologies, the most sensitive components are at the infrastructure layer, as they support the execution of the service layer and its underlying platform. The same is observed when failures are injected into the infrastructure level (see column *service/infra*). In this case the complete rankings reveal that the sensitivity of VM_2 and VM_3 are very close. Indeed, in Topology 3, only the Storage (S) survives a failure of VM_2, whereas only the Notifier (N) survives a failure VM_3. Thus, seen from the service layer, both VM_3 and VM_2 have a similar impact.

This shows that the structure imposed by the service layer significantly hinders the benefits of alternative deployment schemes. In SensApp, with the exception of the Notifier (N), all other services depend on the database, and this remains regardless of the selected infrastructure, software stack and allocation scheme.

E. Threatening Failure Sequences

The third indicator offered by Trio is the identification of the most threatening failure sequences, or in other words, the failures that brings down the largest part of the system and which are very likely to happen. Table III only presents the top sequences identified in each of the situations.

These results confirm the intuition that sequences hitting sensitive components are the most harmful. As shown on Fig. 1, the most two sensitive components of SensApp are the database (DB) and the Notifier (N), and the most harmful sequence is thus to bring them down in this very order. This order reflects the fact that failing the DB impacts more significantly the rest of the application that failing the Notifier. Only the notifier survives a failure of the DB, whereas no other service is impacted by a failure of the notifier.

It is worth to note that harmful failure sequences computed for the alternative deployments reflect as well the sensitive components of SensApp. For instance, regarding Topology 3, failing VM_2 and VM_3 results in failure of the Notifier and the DB, respectively.

F. Discussion

Isolation and *replication* are two common strategies to improve robustness. Yet, as shown in the SensApp case, if they do increase the robustness of the overall system, they do not necessarily increase the robustness of the services. In Topology 3, the usage of isolated platforms supporting the core services, does not actually enhance their robustness.

Getting insight about the benefits of such strategies is possible in early stages of development, but requires indicators that go beyond the mere structure of the graph and finely account for isolation and replication.

It is worth to note that the model is flexible enough to accomodate cases where the knowledge about the deployment is only partial. In Topology 2 for instance, we do not know the execution environment of the DB, which is provided by a PaaS provider. Yet we can assume that it is isolated from the other execution environments, and thus carry on with the calculation of the indicators.

We believe that coupling these robustness indicators to the runtime model offered by CloudML can help manage cloud-based systems, by detecting, in real-time, changes which significantly hinder the robustness of the service layer.

VII. RELATED WORK

Making reasonable design decision, especially in the early stages of the development process is recognised as a major

Topology	Most harmful failure sequences		
	overall	service/infra	service/platform
Topology 1	(VM_2, JRE_1, VM_1)	(VM_3, VM_1)	(JRE_2, DB)
Topology 2	(VM_2, VM_1, DB)	(VM_2)	(JRE_2, DB)
Topology 3	(VM_2, VM_3, VM_1)	(VM_3, VM_2)	(JRE_4, SC_4)
Topology 4	$(VM_2, VM_5, VM_3, VM_4, VM_1)$	(VM_2)	(JRE_2)
Services (Fig. 1)	(DB, N)	n/a	n/a

Table III
MOST HARMFUL SEQUENCES AT VARIOUS LEVEL FOR THE FOUR TOPOLOGIES OF FIG. 2

factor of quality and fast return-on-investment (ROI). Various methods have been proposed to help elicit candidate designs or architecture fragments that meet functional and extra-functional requirements (e.g., CBAM [8], ABC/DD [9]). Reconciling conflicting requirements requires the calculation of indicators for the quality-dimensions of interest. Our approach is an attempt to provide robustness indicators, which can be used in such tools.

Existing indicators are either quantitative when they reflect quantities, that can be actually measured on the final running system, or *qualitative* if they results from subjective expert judgements.

Qualitative indicators are one building block of risk analysis methods such as Predict [10], CORAS [11]. The CORAS method for instance helps identify major threats and the related mitigations based on subjective probabilities. Consolidating expert judgement is an expensive and time consuming activity that our approach avoid as it only focuses on the network topologies and the associated failure propagation model.

Various quantitative indicators have been identified in the past. Graph Theory provides a large body of metrics such as connectivity, betweeness, distance or reliability polynomials [12], [13], which are correlated to some forms of robustness. Yet, they do not simultaneously accommodate for both *isolation* and *replication*. Connectivity for instance correlates robustness with a high number of alternative paths, reflecting replication. Yet, depending on the failure propagation model, a highly connected graph may be very brittle due to the extensive propagation of failures. In Caballero et al. [14] for instance, robustness is measured as the connectivity of coloured networks after taking off of nodes from the graph which have particular colours. This work assumes that failures of networks are caused by software bugs, and therefore when a failure happens, all the nodes hosting the same software will be down. The work simplified the software on network nodes, without considering the dependencies and software stacks. Gorbenko et al. [15] consider the software stacks for measuring the security of networked systems, and measure the security of a whole stack by the time it requires to recover from an attack (by switching to other alternative services or waiting for a patch of the attacked software). The measure need historical data of software patches, and therefore only works on well-supported software.

Fault Tree Analysis (FTA) [16] is another general tool used to study failures. A fault tree represents the logical combination of events which lead to a particular failure, and can be used for robustness evaluation and diagnosis. By contrast with FTA, failure sequences do not focus on a single particular failure but account for sequences of failures and how their accumulation impacts the system or subsystem of interest.

Other application-specific indicators can be derived by detailed system simulation. In Palladio [17] for instance, one can model the components of the system, their dependencies, as well as the underlying platform. The model, which includes detailed behavioural description is then translated into queueing networks, from which one can obtain various metrics such as mean-time to failure (MTTF). By contrast with our robustness metric, these metrics are accurate. Yet, the large amount of detailed needed in input can be prohibitive at the early stages of development.

To the best of our knowledge, this research work is the first attempt to adapt the notion extinction sequences, well accepted in Ecology, to the problem of deployment topology robustness.

VIII. CONCLUSION

With the ever growing number of opportunities provided in the Cloud, it becomes critical to make the right choices regarding the architecture of the system in the early stages of development. To this end, we provide three robustness indicators: the overall robustness, the most sensitives components and the most threatening failure sequences. These indicators can be derived from any deployment topology, provided a fault propagation model. Our solution, inspired by Ecology, is built upon an analogy between species extinction and components' failures, which both propagate into the (eco-) system. Through SensApp, our running example, we showed how these three indicators help sort out architectural decisions regarding robustness. One limitation of our approach is that it does not take into account the probability that a given component fails. Such probabilities, which can be found for some Cloud providers, could help calibrate the indicators with realistic data and ease the use of our indicators.

ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number: 318484 (MODAClouds) and 600654 (DIVERSIFY).

APPENDIX

We detail below a proof of the convergence of the failure propagation. As we shall see, failures propagation is monotonic and bounded by the zero-state vector (i.e., all components are failed): it therefore converges as states the *monotone convergence theorem*.

A. Lower bound of failure propagation

Recall that the propagation of a failure to the direct neighbours is given by the function-vector $\mathbf{p}(\mathbf{s})$, conforming to the following grammar:

$$\begin{array}{rcl} \mathbf{p} & ::= & (p_1,\, p_2,\, \ldots,\, p_k) \\ p_i & ::= & s_i \;\wedge\; e \\ e & ::= & s_j \;\mid\; e_1 \wedge e_2 \;\mid\; e_1 \vee e_2 \;\mid\; \neg\, e \end{array} \tag{10}$$

Given a state vector $\mathbf{s}$, we denote the evaluation of a propagation function-vector $\mathbf{p}$ by the function $\mathcal{E}$ as shown below:

$$\begin{array}{rcll} \mathcal{E}_{\mathbf{s}}[\![(p_1, \ldots, p_k)]\!] & = & (\mathcal{E}_{\mathbf{s}}[\![p_1]\!], \ldots, \mathcal{E}_{\mathbf{s}}[\![p_k]\!]) & (11) \\ \mathcal{E}_{\mathbf{s}}[\![s_i \wedge e]\!] & = & \mathcal{E}_{\mathbf{s}}[\![s_i]\!] \wedge \mathcal{E}_{\mathbf{s}}[\![e]\!] & (12) \\ \mathcal{E}_{\mathbf{s}}[\![s_i]\!] & = & \mathbf{s}[i] & \\ \mathcal{E}_{\mathbf{s}}[\![e_1 \wedge e_2]\!] & = & \mathcal{E}_{\mathbf{s}}[\![e_1]\!] \times \mathcal{E}_{\mathbf{s}}[\![e_2]\!] & \\ \mathcal{E}_{\mathbf{s}}[\![e_1 \vee e_2]\!] & = & \mathcal{E}_{\mathbf{s}}[\![e_1]\!] + \mathcal{E}_{\mathbf{s}}[\![e_2]\!] & \\ \mathcal{E}_{\mathbf{s}}[\![\neg e]\!] & = & 1 - \mathcal{E}_{\mathbf{s}}[\![e]\!] & \end{array}$$

Eq. 12 intuitively implies that failures propagate throughout the topology until every single component is failed. The zero vector $\mathbf{s}_0 = (0, 0, \ldots, 0)$ acts as the absorbing element of the propagation and is therefore its lower bound.

B. Monotony of failure propagation

We demonstrate below that the associated level of activity always decreases (or remains constant) while failures propagate. In other words:

$$\alpha\left(\mathcal{E}_{\mathbf{s}}[\![\mathbf{p}]\!]\right) \leq \alpha(\mathbf{s}) \tag{13}$$

Using Eq. 11, we can rewrite Eq. 13 as follows:

$$\alpha\left(\left(\mathcal{E}_{\mathbf{s}}[\![p_1]\!], \ldots, \mathcal{E}_{\mathbf{s}}[\![p_k]\!]\right)\right) \leq \alpha(\mathbf{s}) \tag{14}$$

Given the definition of the activity level α, as the number of active component in the system, Eq. 14 yields:

$$\sum_{i=1}^{k} \mathcal{E}_{\mathbf{s}}[\![p_i]\!] \leq \sum_{i=1}^{k} s_i \tag{15}$$

To show that the number of active components is less or equal after the propagation, we show that each component can only be inactivated if it is still active but not activated again. In other words:

$$\begin{array}{rcll} \forall i \leq k, \quad \mathcal{E}_{\mathbf{s}}[\![p_i]\!] & \leq & s_i & (16) \\ \forall i \leq k, \quad \mathcal{E}_{\mathbf{s}}[\![s_i \;\wedge\; e_i]\!] & \leq & s_i & \\ \forall i \leq k, \quad \mathcal{E}_{\mathbf{s}}[\![s_i]\!] \;\times\; \mathcal{E}_{\mathbf{s}}[\![e_i]\!] & \leq & s_i & (17) \end{array}$$

Eq. 17 holds due to the fact that the future state of a component is the conjunction between the its current state s_i and an logical expression e_i over its direct environment. As shown in Table IV, once a component has failed, it remains failed forever.

s_i	$\mathcal{E}_{\mathbf{s}}[\![e_i]\!]$	$s_i \wedge \mathcal{E}_{\mathbf{s}}[\![e_i]\!]$	$s_i \times \mathcal{E}_{\mathbf{s}}[\![e_i]\!] \leq s_i$
Active	Active	Active	yes
Inactive	Active	Inactive	yes
Active	Inactive	Inactive	yes
Inactive	Inactive	Inactive	yes

Table IV
TRUTH TABLE OF EQ. 12

Given the fact the failure propagation forms a monotone decreasing sequence bounded by the zero vector, it converges toward its very minimum, as stated by the *monotone convergence theorem*.

REFERENCES

[1] F. Buschmann, "To pay or not to pay technical debt," *Software, IEEE*, vol. 28, no. 6, pp. 29–31, Nov 2011.

[2] S. Mosser, F. Fleurey, B. Morin, F. Chauvel, A. Solberg, and I. Goutier, "SENSAPP as a Reference Platform to Support Cloud Experiments: From the Internet of Things to the Internet of Services," in *SYNASC 2012: 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society, 2012, pp. 400–406.

[3] D. Roman, X. Gao, and A. J. Berre, "Demonstration: Sensapp - an application development platform for ogc-based sensor services," in *Proceedings of the 4th International Workshop on Semantic Sensor Networks, SSN11, Bonn, Germany, October 23, 2011*, ser. CEUR Workshop Proceedings, K. Taylor, A. Ayyagari, and D. D. Roure, Eds., vol. 839. CEUR-WS.org, 2011, pp. 107–110. [Online]. Available: http://ceur-ws.org/Vol-839/roman.pdf

[4] F. Havlik, D. Havlik, M. Egly, A.-J. Berre, R. Grønmo, H. van der Shaaf, S. Modafferi, S. Middleton, Z. Sabeur, C. Granell, M. A. Esbrí, J. Lorenzo, K. Schleidt, and J. Pielorzoo, "Final recommendations for environmental enablers," ENVIROFY Consortium, Deliverable D4.4, 2013.

[5] J. Memmott, N. M. Waser, and M. V. Price, "Tolerance of pollination networks to species extinctions," *Proceedings of the Royal Society of London. Series B: Biological Sciences*, vol. 271, no. 1557, pp. 2605–2611, 2004. [Online]. Available: http://rspb.royalsocietypublishing.org/content/271/1557/2605.abstract

[6] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *CLOUD 2013: IEEE 6th International Conference on Cloud Computing*, L. O'Conner, Ed. IEEE Computer Society, 2013, pp. 887–894.

[7] N. Ferry, G. Brataas, A. Rossini, F. Chauvel, and A. Solberg, "Towards Bridging the Gap Between Scalability and Elasticity," in *CLOSER 2014: 4th International Conference on Cloud Computing and Services Science – Special Session on Multi-Clouds*. SCITEPRESS, 2014, pp. 746–751.

[8] R. Kazman, J. Asundi, and M. Klein, "Quantifying the costs and benefits of architectural decisions," in *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, May 2001, pp. 297–306.

[9] X. Cui, Y. Sun, and H. Mei, "Towards automated solution synthesis and rationale capture in decision-centric architecture design," in *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, Feb 2008, pp. 221–230.

[10] A. Omerovic, B. Solhaug, and K. Stølen, "Assessing practical usefulness and performance of the prediqt method: An industrial case study," *Information & Software Technology*, vol. 54, no. 12, pp. 1377–1395, 2012.

[11] M. S. Lund, B. Solhaug, and K. Stølen, *Model-driven risk analysis: the CORAS approach*. Springer, 2010.

[12] R. Wilkov, "Analysis and design of reliable computer networks," *Communications, IEEE Transactions on*, vol. 20, no. 3, pp. 660–678, Jun 1972.

[13] A. Bigdeli, A. Tizghadam, and A. Leon-Garcia, "Comparison of network criticality, algebraic connectivity, and other graph metrics," in *Proceedings of the 1st Annual Workshop on Simplifying Complex Network for Practitioners*, ser. SIMPLEX '09. New York, NY, USA: ACM, 2009, pp. 4:1–4:6. [Online]. Available: http://doi.acm.org/10.1145/1610304.1610308

[14] J. Caballero, T. Kampouris, D. Song, and J. Wang, "Would diversity really increase the robustness of the routing infrastructure against software defects?" *Department of Electrical and Computing Engineering*, p. 40, 2008.

[15] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and A. Romanovsky, *Using diversity in cloud-based deployment environment to avoid intrusions*. Springer, 2011.

[16] L. Xing and S. Amari, "Fault tree analysis," in *Handbook of Performability Engineering*, K. Misra, Ed. Springer London, 2008, pp. 595–620. [Online]. Available: http://dx.doi.org/10.1007/978-1-84800-131-2_38

[17] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.