Chauvel *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2015) 4:18
DOI s13677-015-0043-7

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Evaluating robustness of cloud-based systems

Franck Chauvel*, Hui Song, Nicolas Ferry and Franck Fleurey

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 parts 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.

Keywords: Systems architectures; Cloud-based systems; Robustness indicators; Robustness metric; Software deployment; Sensitive components; Failure sequences

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, Cloud Sigma). In this dynamic environment, cloud-based systems are continuously refined to make the most of new opportunities.

High availability is one of the main selling argument of Cloud solutions and is therefore included in many service level agreements (SLA). Yet, systems—even highly available ones—eventually fail [1]. Engineers must then understand and quantify the impact of possible failures, that is to say to measure the *robustness*. Robustness complements reliability and availability: *reliability* quantifies the frequency of failures and *availability* reflects the fraction of the time when the system is up and running (implying repair). Measuring robustness is critical to understand and estimate the consequences that architectural decisions may have on the robustness of the system. Understanding robustness thereby helps avoid expensive rollbacks of inappropriate decisions.

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 judgments, are always available but are extremely time consuming to consolidate. By contrast quantitative indicators of such as ATAM [2] result from complex procedures 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, which require minimal knowledge of the systems and do not involve expert judgments. 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

*Correspondence: franck.chauvel@sintef.no
SINTEF ICT, P.O. Box 124 Blindern, N-0314 Oslo, Norway

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. We also compare our measure with existing topological metrics, including density, distance, and centrality.

The remainder is organised as follows. Section Motivating Example introduces a running example of Cloud-based system, where various deployment architectures may be envisioned. Section Robustness of Ecosystems first introduces how robustness is understood and estimated in Ecology. Section Overview summarises our robustness indicators and discusses the minimal required inputs. Section Cloud Robustness Indicators details their calculation and explain how the models used in Biology are tailored to Cloud-based systems. Section Robustness Operationalisation explains how our measurement can be obtained on a real systems, by using cloud management tools. Section Running Examples presents our prototype implementation and how we used it to select robust deployments of SensApp. In Section Comparisons, we contrast our robustness metric with existing one from graph theory. Section Threats to Validity reviews the main weakness of our approach before Section Related Work discusses a selection of complementary publications and Section Conclusion presents our future research directions.

Motivating example

SensApp [3] 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. It gives sensors a place to continuously push data and provides Cloud services for notification and query.

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

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.

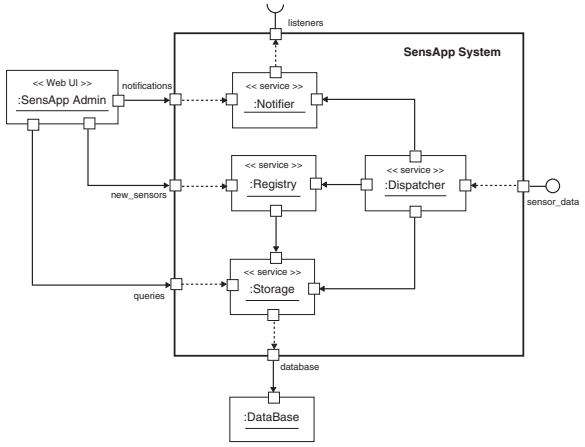

Fig. 1 Decomposition of SensApp into high-level functional components, depicted as a UML 2 component diagram

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 platforms are provided as services (e.g., Heroku, GoogleApp Engine, Cloudbees) but they 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 virtual machine (VM).

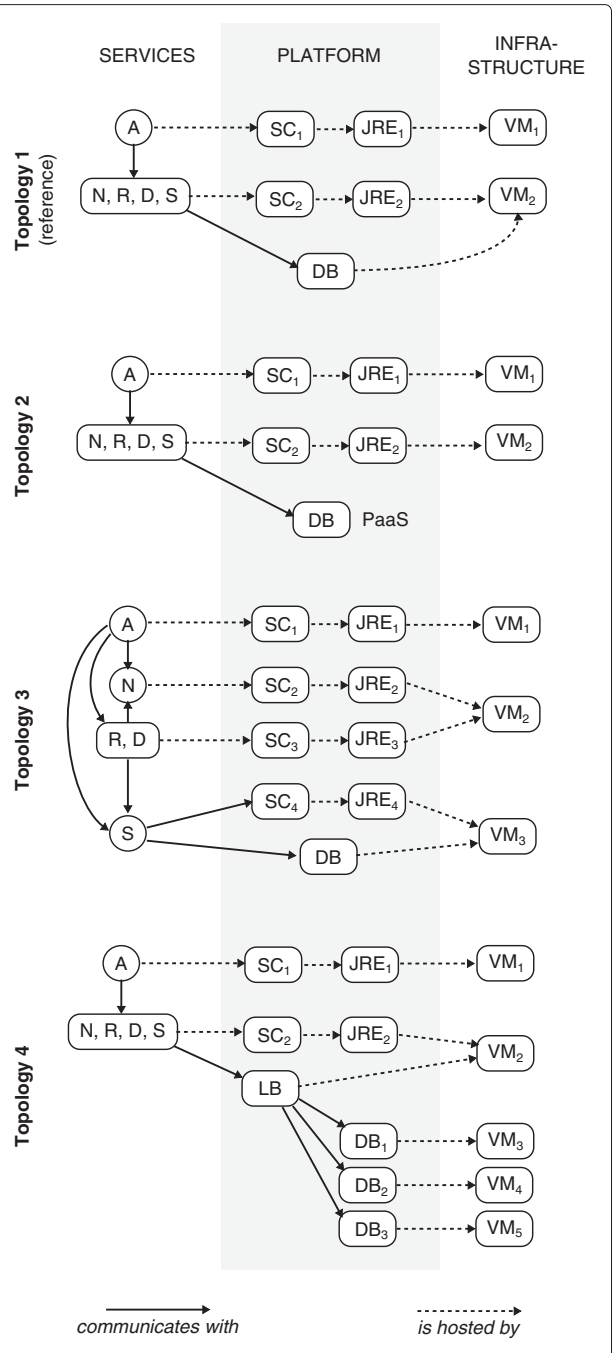

Fig. 2 Four alternatives deployment topologies of the SensApp system

Figure 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 [4] or ENVIROFY [5].

Getting insight regarding the robustness of the SensApp services in such topologies is critical and difficult, regardless of the approach chosen: graph theory metrics or full-fledge architectural evaluation.

Existing graph-theory metrics such as density, average node degree, distance or betweenness, fail to capture trade-offs between isolation and replication. These metrics are often used to quantify various forms robustness at the architecture level and are readily available in most of scientific computation software packages such as Matlab, or R.

Yet, interpreting such metrics requires to choose either a isolation or a replication oriented view. With isolation in mind, a dense topology (one with more edges) implies broader propagation of failures and, in turn, a more brittle architecture. By contrast, with replication in mind, a dense topology has alternative and redundant paths, and is therefore synonym of robustness.

Alternatively, comprehensive analysis methods, such as ATAM [2] or ABC/DD [6], remain quite time and resource consuming. These are comprehensive processes including analysis and testing, and could be used for audit for instance, but are of little help to quickly sort out architectural ideas, especially when the systems does not yet exist. In our SensApp scenario, although we could go through such heavy procedure, we do not have all the pieces of knowledge needed to apply such methods onto the speculative topologies.

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.

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

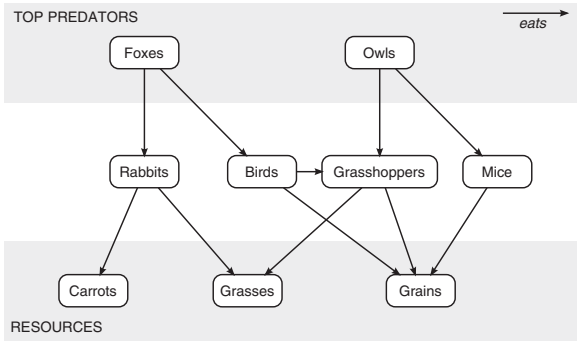

Fig. 3 Illustrative ecosystem capturing the prey-predator relationships between a small set of species

trophic levels which gather species with similar properties, such as the "resources" or the "top predators" in Fig. 3.

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* [7]. 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.

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.

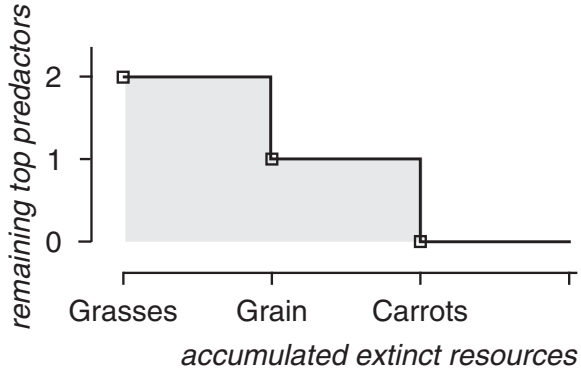

Fig. 4 Response of the "top predators" group to the extinction sequence "grasses, grains and carrots" (cf. Fig. 3)

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

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.

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.

These three indicators respond to different questions, which motivates robustness analysis in our experience.

- Which topology is the most robust? When considering alternative designs or deployments, engineers have to select as objectively as possible, a robust one. The key selling factor of Cloud solutions (public, private, PaaS, SaaS) is availability, reliability or both.

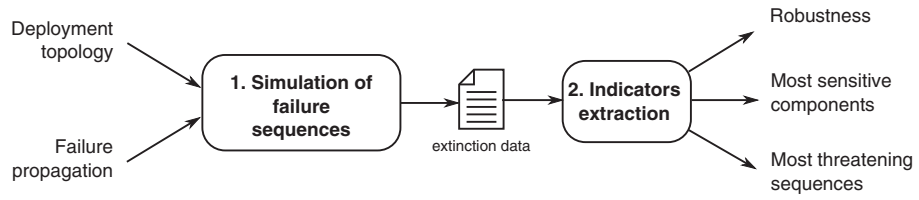

Fig. 5 Overview of the robustness indicators computation process

The robustness indicator is way to capture and rank alternative topologies with minimal effort.

- Which component compromises the topology? When the system is already deployed, and understanding what causes an outage is very difficult. Knowing most sensitive components helps prioritise the investigations.
- How a malicious agent can take down the system? When a system goes in production, unforeseen events will happen, especially unwanted requests. Knowing the major threats raises awareness of weakness of the topology.

Various factors contribute to failures: resource contention, high work-load, unexpected inputs, network configuration, and so on. Understanding how failure propagated is a complex endeavour (requiring information of concurrent processes, memory snapshots, etc.) which goes beyond the use of these indicators. We focused our contribution on supporting key architectural decisions.

Cloud robustness indicators

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.

Figure 2 in Sec. 1 denotes these two types of , 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 $\mathbf{s}' = \mathbf{s} \wedge \mathbf{f}$.

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

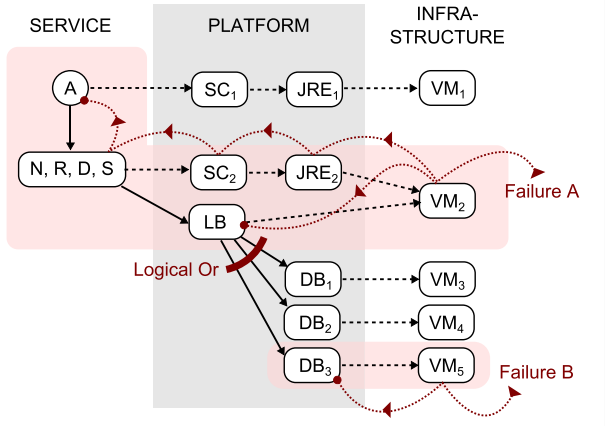

Fig. 6 Propagation of two separate failures throughout Topology no. 4

the Admin component A and the load-balancer LB are:

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

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 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:

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

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

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

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 1.

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

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

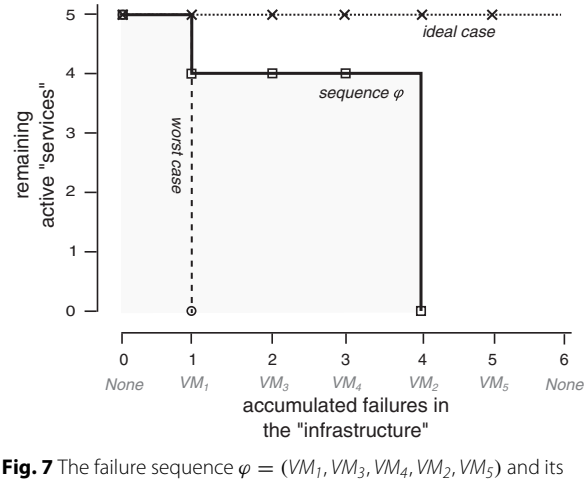

Fig. 7 The failure sequence $\varphi = (VM_1, VM_3, VM_4, VM_2, VM_5)$ and its impact on the service layer in Topology 4

in the layers of interest (i.e., services and infrastructure), we normalise it and obtain $r(\mathbf{s}, \varphi) = 0.48$.

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:

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

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

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

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.

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.

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:

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

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:

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

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:

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

In the following, we shall use these three indicators to identify a more robust deployment for the SensApp application.

Accounting for reliabilities

Failure sequences only captures the dependencies between components, but fail to account for individual reliabilities.

As opposed to robustness, which measures the impact of failures, *reliability* measures the capacity to function properly over a period of time. Engineers quantify it using *failure rates* and *mean time to failure* (MTTF), but other representations such as ranges or probability distributions may better describe the inherent uncertainty in predicting failures.

Intuitively, failure sequences are not equally probable. Universal third-party components, such as standard libraries or application servers, are more reliable and therefore less likely to fail than new in-house components. A failure sequence where reliable components fail first is unlikely. The generation of failure sequences must therefore sample components according to their individual reliability.

A proportional sampling algorithm such as the roulette-wheel or the tournament selections accounts for reliabilities expressed as scalar values, and results in a set of failures sequences globally reflecting individual reliabilities.

We may go further and account as well for the time when failures are triggered. The horizontal axis is no longer only ordinal, but stands for time, and the robustness value thereby reflects the overall lifetime of the architecture. To our experience, this brings little added value as the robustness values are no more comparable because their normalisation depends on time.

Robustness operationalisation

One valuable aspect of this robustness measurement, is that it can be operationalised: it is possible to quantify the robustness of an existing system, without relying on simulation. This requires the capacity to inject failures and the capacity to detect how they propagate. We explain below each of these two activities.

Figure 8 shows the key steps of the process. The test driver first provisions the system in a test environment using the *set up* procedure (including provisioning virtual machines, installing and configuring components). The test driver then iterates through the given failure sequence: for each component, it triggers its failure using the associated *fault injection script* and then run a complete *test suite* to check the status of all components, as any could have failed consequently. Eventually, the test-driver releases the cloud resources used by the systems using the *tear down* procedure and outputs the response (so called trace) associated with the failure sequence.

If enough failure sequences are run, all three robustness indicators can be extracted from the set of traces produced by the test driver. By contrast with the simulation techniques, measuring robustness does not require to know the dependencies between components nor the rules that govern failure propagation. We rely on test suites to detect cascading failures.

Set-up & tear down

The robustness testing process described above follows the general structure of a test. It includes a *set up* phase, which provisions a new instance of the system in a test environment. Its counterpart is the *tear down*, which frees all the resources needed by this test instances. These two steps are critical to prevent interference between failure sequences and to ensure that each failure sequence is run against a system whose state is fully known.

Recent Cloud management tools such as Cloudify, Chef, Puppet or CloudML provide an effective means to fully automate the deployment of complex cloud based architecture. In a nutshell, these tools can provision virtual machines and connect to them to install the needed software packages.

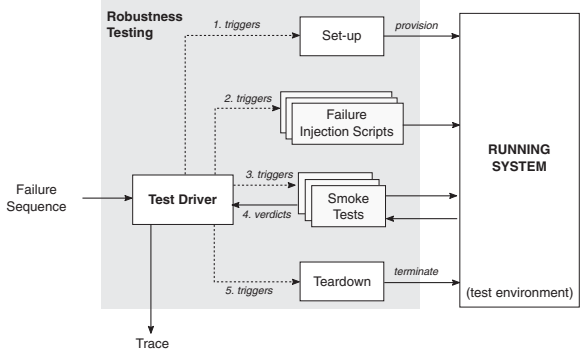

Fig. 8 Using fault-injection and testing to measure robustness to specific failure sequences

Failure injection

In a Cloud setting, injecting failures (in a test environment) can be done in two main ways: by simply shutting down the components of interest, or by changing the network configuration so that calls to not reach the expected end point. This sort of remote control of architecture are now supported by Cloud management tools such Chef, Puppet or CloudML to name a few, but can also be encoded in shell scripts.

Note that shutting down a remote component properly (i.e., using the devoted API calls) may give the component enough time to notify its partners of its termination, so that they may take any possible relevant action, such as switching to a degraded mode for instance. In this respect, network configuration changes, or process interruption may better emulate the sudden failure of a software component. In addition, virtualisation techniques now permit to inject failure in the infrastructure level by terminating for instance the virtual machine hosting the components.

In our SensApp example (see Sec. 1), injecting a failure of a War container, such as Jetty for instance, can be done by just killing the associated process at the system level. With the proper access rights, it is possible to open an SSH connection to the host system, and to kill the Jetty server process.

Failure detection

Detecting failure of components can be done using tests, that exercises a service/component and reports deviation from the expected behaviour. To detect the state of the system once a failure has been injected, one therefore needs to run a single test for each component in the system.

A test that verifies whether a component is up and running do not necessarily need to exercise edge-cases of this component specification. Single invocation with a set of minimum assertions are enough in practise to detect issues caused by a problem at the architecture level, such as a missing dependency. Such tests, referred as *smoke tests* by engineers are often used to check the validity of fresh deployments. They differ from integration tests, which are specifically designed to exercise specific transaction in the system. Regarding web components, tools such as Selenium for instance, enable QA engineers to quickly automate these simple testing tasks at the GUI level. Such tools are good solutions for the development of the needed smoke tests.

In practise, the use of smoke tests to evaluate the impact of failure injected at the architecture level may be challenged by the existence of optimisation techniques such as caching for instance, which may prevent to properly exercise the dependency of the component under test. Persistence frameworks such as Hibernate for instance

may cache objects that are read from the database and avoid subsequent connection to the database.

Running examples

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.

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

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 [8, 9] 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 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.

Overall robustness

The first indicator is the overall robustness, which reflects to which extent the system is able to stand failure sequences. Table 1 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.

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 failures 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.

Relative robustness

Robustness indicators can also be calculated with respect to specific subsystems. Columns 3 and 4 of Table 1 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

Table 1 Robustness indicators computed for the four topologies of Fig. 2

	Robustness of X to failure in Y		
Topology	overall	service/infra	service/platform
Topology 1	0.1364	0.2064	0.1394
Topology 2	0.1492	0.2112	0.1421
Topology 3	0.1518	0.1890	0.0955
Topology 4	0.1928	0.3057	0.1478
SensApp (Fig. 1)	0.1488	n/a	n/a

failures in the infrastructure ($0.1890 < 0.2112$) and to failure in the platform ($0.0955 < 0.1421$).

In Topology 2, two components might fail at the infrastructure level: VM_1 and VM_2. Without additional information, both failures are equally probable, 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 that 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.

Sensitive components

The second indicator provided by Trio is the identification of the most sensitive components, that is to 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.

Table 2 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.

Table 2 Most sensitive components of the four SensApp topologies

	Most sensitive components		
Topology	overall	service/infra	service/platform
Topology 1	VM_2	VM_2	*DB*
Topology 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

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 3 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.

Effect of individual reliability

By changing the way we sample failure sequences, robustness can account for the reliability of individual components. We present here how changes in the reliability of a single component affects the robustness of the whole architecture. We used Topology 3, where we varied the MTTF associated to VM_2, its most sensitive component (as shown in Table 2). We assigned a MTTF of 1 to all other components. Figure 9 shows the response of the robustness of the service layer, with respect to failure in the infrastructure.

As shown by the vertical dashed line, when all components are equally reliable (MTTF = 1), the robustness of the service layer is about 0.18 (cf. Table 1). Increasing the MTTF of VM_2 does increase the robustness, but it follows a law of diminishing return. Assuming that the cost of increasing reliability is constant, there is a limit above which, increasing reliability does not worth the associated gain in robustness.

These results emphasise that a sensitive component can be mitigated by either increasing its reliability or by modifying its role in the topology (using replication for instance).

Table 3 Most harmful sequences at various level for the four topologies of Fig. 2

	Most harmful failure sequences		
Topology	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

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.

Performance benchmark

To better understand the factors that influence the time we spend simulating failure sequences, we measured the duration spent simulating 500 failure sequences of random architecture models (including random propositional formulas driving local failure propagation). We generated 600 Trio topologies, representative of three commonly used families of graphs, namely random graphs [10], small world graphs similar to social networks [11], and scale free graphs, similar to human-made architectures (including software architectures [12, 13]). Appendix 1 details the methods we used to generate these random Trio topologies. The size of these synthetic topologies (their number of components) is uniformly distributed over the interval $[0, 2500]$. According to our experience, such models represent architectures that are larger than existing cloud architectures.

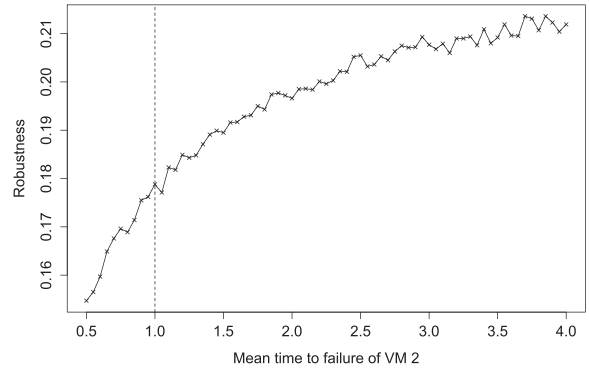

Fig. 9 Effect of individual reliability on robustness

This performance benchmark was run on a laptop equipped with a processor Intel Core i7-4810MQ 2.80 GHz, with 16 GB of RAM and running Windows 7 Enterprise (64 bits). The system was equipped with a 64 bits Java virtual machine (Java 8 update 25) running with default settings. To minimize variation during the experiment, 25 models were initially simulated as a "warm-up" of the JVM, to let it cache enough compiled byte code, before we actually measure speed.

As one can see on Fig. 10, the time spent simulating 500 failure sequences is strongly influenced by the size of the architecture model and by the family of topology. To quantify the influence of these factors on the failure sequence simulation duration, we carried out a regression analysis. We assume a quadratic model based on the relationship:

$$\begin{aligned}\sqrt{\text{duration}} &= B_4 \times \text{size} \\ &+ B_3 \times \text{density} \\ &+ B_2 \times \text{scale-free} \\ &+ B_1 \times \text{small-world} \\ &+ \varepsilon\end{aligned}$$

Table 4 presents the key statistics of the resulting prediction model. Note that the size, density and graph family factors explain approximately about 80 % of the variance of the duration of failure sequence simulation ($R^2 \approx 0.79$). Note that the density of the graph has a limited impact on the overall duration.

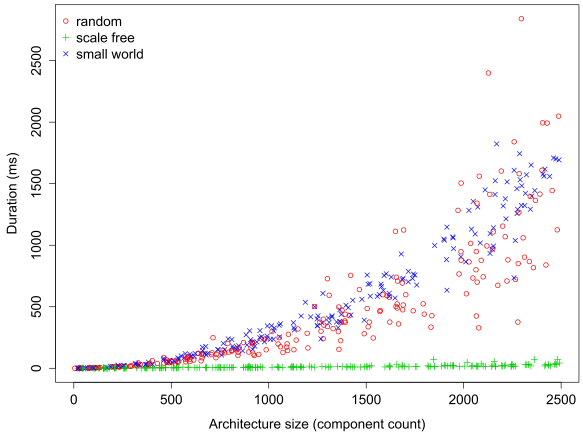

Fig. 10 Duration of failure sequences simulation for 600 topologies including up to 2 500 components

Finally, note that the algorithm is significantly faster for scale free topologies, which are typical of human-made architecture, and especially software architecture.

Comparisons

We discuss here the validity of our robustness metrics, that is to say the extent to which it does reflect a sort of robustness. To investigate this question, we studied the correlations between our robustness metric and existing metrics used in graph/network theory. To do so, we generated 600 trio topologies, covering three graphs families (random (R), small worlds (SW) and scale-free (SF) graphs), for which we computed selected robustness metrics, and we computed their correlation with our metric. Appendix 1 details the generation process we used.

We selected the following topological metrics, used in the literature to characterize robustness. We refer the reader to [14] for a comprehensive treatment of such topological metrics.

Density The density of the graph is the ratio of the number of edges in the graph over the number of possible edges.

Average node degree The degree of a node is the number of edges connected to that node. By extension, the average node degree is the mean degree of all nodes.

Diameter The diameter of the graph is the longest of all shortest path between any pair of vertexes.

Mean distance The mean distance is mean length of all shortest path between any pair of vertexes.

Centrality The centrality reflects the importance of a node in a graph, and in turn the "speed" at which random "messages" can navigate from one node to the other. There are various formulas to compute centrality, based on node degree, node distance, eigenvalue of the adjacency matrix.

For the record, the correlation coefficient is a value within $[-1, 1]$, where both extrema show a strong negative or positive correlation, whereas 0 stands for no correlation. As one can see in Table 5, in general, our robustness metrics does not correlate with existing topological metrics (all correlation coefficients are below ± 0.5). Yet, for scale-free graphs, which are more representative of software architectures, the correlations is much more significant: only one correlation with the average node degree is below ± 0.5. This confirms that some of the robustness characterized by topological metrics is also reflected in our metric.

Threats to validity

We discuss in this section the validity, the reliability and usefulness of our robustness measurement.

Evaluating *validity* is questioning the extent to which we do measure a form of robustness. Recalling the IEEE glossary of Software Engineering [15] where *robustness* is defined as *the degree to which, a system or a component*

Table 4 Regression analysis characterizing the impact that architecture size, density, and graph family have on failure sequence duration

Parameters	B_i	Std. err.	t-value	p-value
size	1.07e-02	3.06e-04	34.980	< 2e-16
density	-7.40e-01	1.28e+00	-0.577	0.564
scale free	-1.45e+01	8.51e-01	-17.122	< 2e-16
small world	1.77e+00	7.60e-01	2.332	0.020
(intercept)	5.21e+00	8.30e-01	6.277	6.65e-10
			residual	5.486
			F statistic	594.7
			adjusted R^2	0.7986

Table 5 Pearson correlation coefficients between our robustness metric and existing topological metrics

Metric	Overall	Graph family		
		R	SF	SW
density	0.17	−0.05	0.82	−0.26
diameter	−0.24	−0.16	−0.59	0.68
avg. distance	−0.01	−0.13	−0.62	0.63
avg. node degree	0.30	0.83	0.34	0.76
cent. degree	0.42	0.23	0.84	0.38
cent. betweenness	0.44	−0.07	0.52	0.61
cent. eigenvalue	−0.34	0.50	−0.70	0.75
cent. closeness	0.42	0.10	0.81	0.22

can function correctly in the presence of invalid inputs or stressful environmental conditions, we see that our measurement does not account for invalid inputs, but only for stressful environmental conditions. In our approach the notion of *environment* only reflects other architectural entities on which the system depends, but does not account for network conditions such as work load for instance.

Evaluating *reliability* is questioning, whether our approach, applied on the same system by different people, would eventually yield similar robustness values. Although our measure of robustness is an *expected value* and its computation is subject to some variance, the main threat to reliability is the model on which robustness is computed. We are convinced that system modeling at the architectural level (components and dependencies) leaves little room for interpretation, as such components often exist as deployable artifacts. The logical formulae, which are used to simulate failure propagation, are a source of variation but note that if the approach is used to generate tests, such formulae are not needed.

Regarding the *usefulness* of our measurement, one could argue that Cloud services often exhibit high availability (i.e., 99 % is often found in SLA). Yet, as pointed by M. Nygard [1], "*despite our best laid plans, bad things will happen*". Major cloud consumers follows this advice and anticipate possible failures. At Netflix for instance[1], one of the main consumer of Cloud resources, engineers do fail significant parts of their infrastructure to evaluate the impact on their services, using a program so-called "Chaos monkey". The measurement we propose does not solve the issue of robustness, but it helps anticipate what bad things can happen at the architecture level, and in turn, helps to make sure that the system can recover.

Related work

This work is the continuation of a line of research about robustness, whose preliminary results were published in the UCC 2014 conference [16].

Making reasonable design decision, especially in the early stages of the development process is recognised as a major 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., ATAM [2], CBAM [17], ABC/DD [6]). 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 judgments.

Qualitative indicators are one building block of risk analysis methods such as Predict [18], CORAS [19]. The CORAS method for instance helps identify major threats and the related mitigations based on subjective probabilities. Consolidating expert judgment 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, betweenness, distance or reliability polynomials [20, 21], 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. [22] 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. [23] 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 needs historical data of software patches, and therefore only works on well-supported software.

Alternatively, *percolation theory* studies how graphs react to addition (resp. removal) or nodes or links. Although this more of a theoretical framework, the method remains similar: removing elements and measuring the evolution of some key properties. Endurance [24] and elasticity [25] are other attempt to understand robustness in terms of failures and how the propagate through a graph. By contrast with these work, our approach permits to finely tuned the propagation of failure for each component.

Fault Tree Analysis (FTA) [26, 27] 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.

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.

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. We shown that our metrics permits to simultaneously account for both isolation and replication strategies, and remains computable on a reasonable amount of time.

Yet, our metrics remain subject to the understanding engineers have of the system. Automating the extraction of both the architecture as well as the failure propagation models would secure our robustness metrics and contribute to better understand such systems that have evolved for many years.

Endnote

[1] See http://techblog.netflix.com/2011/07/netflix-simian-army.html

Appendix A: Random TRIO topologies

We detail below, the process we used to generate randomized Trio models. A Trio topology is a directed graph where each node is decorated with a propositional formula, where each variable refer to the other nodes in the graph (i.e., the successors node). The procedure we followed can be summarized as follows:

1. Generate a random directed graph $G = (V, E)$;
2. For each vertex $v \in V$, generate a random propositional formula involving the vertex $succ(v)$.

A1. Random directed graphs

We considered three main classes of graphs, namely *random graphs*, *small worlds* and *scale free graphs*. A graph is a very versatile model that can describe structures ranging from social networks to molecules. Depending on the domain of interest, graphs may have very

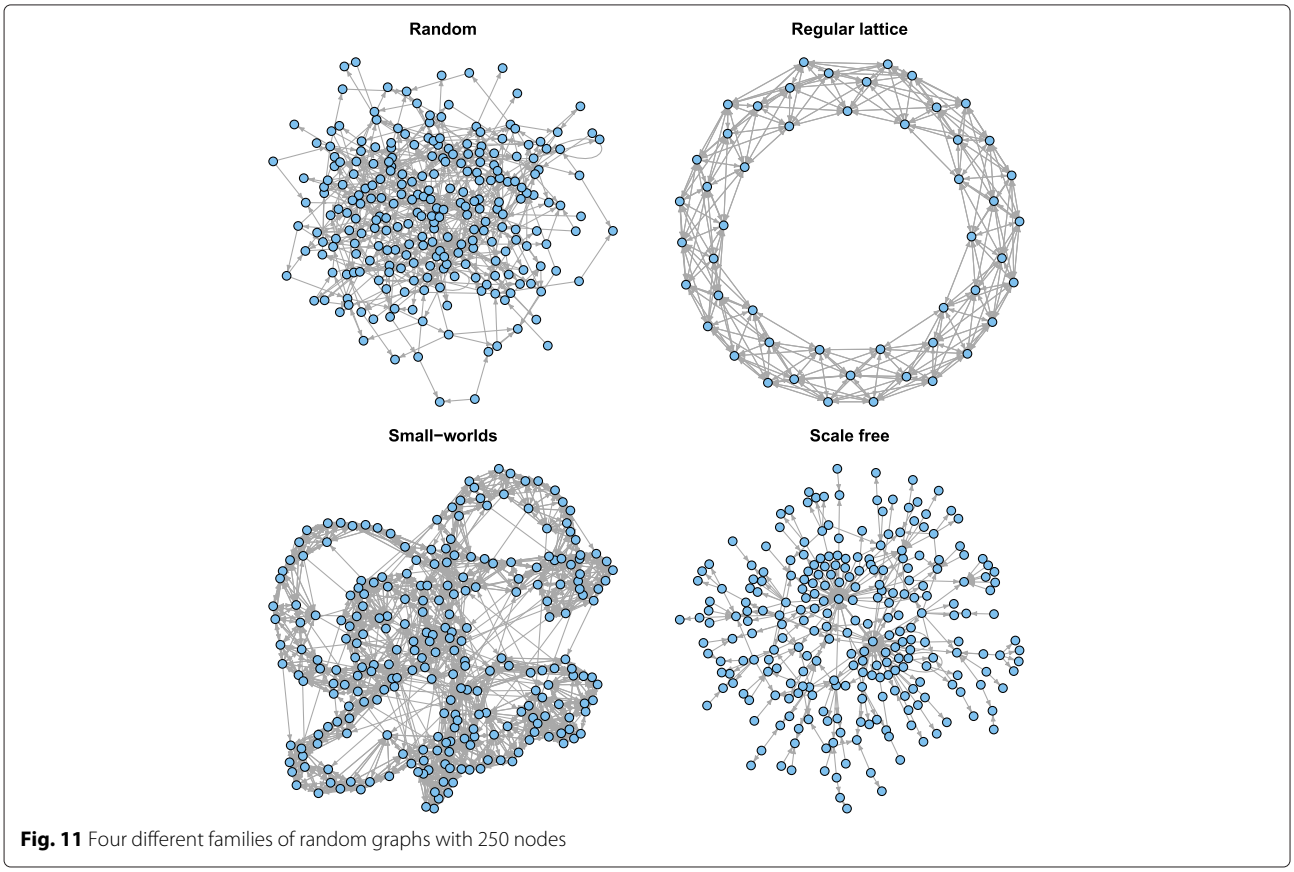

Fig. 11 Four different families of random graphs with 250 nodes

different structures, which are captured by their statistical properties, and especially by their node degree distribution. Figure 11 illustrates the structure of such graphs families.

Random graphs are graphs whose node-degree distribution follows a Poison law. Erdös and Renyi [10] proposed a model to generate random graphs with a given number of vertexes, where each possible egdes has a fixed probability to exist. Although such graphs are easy to generate, they are not representative of structures observed in either nature or engineering.

Small-worlds graph In a small-worlds graph, each vertex can be reached from every other vertex in a small number of steps. Small worlds are often characterized by interconnected aggregations of vertexes. Such graphs are typical of human collaborations such as social networks for instance. We used the model proposed by Watts and Strogatz [11] to generate such small world graphs. In a nutshell, the idea is to first build a regular ring lattice and then to randomly replace edges with a fixed probability.

Scale-free graphs In scale-free graph some vertex have significantly more edges than the others: the degree distribution follows a power law. Such graphs have been observed in human made structure, such as network infrastructure of the Worldwide Web. We generate such graphs using the Barabási and Albert model [28], where graphs are built incrementally: New vertexes are connected to existing vertexes chosen with a probability that depends on their number of edges.

A.2 Random propositional formula

Once a random graph is available, we convert it into a TRIO topology. A vertex in the graph is mapped onto a component in the TRIO topology, and for each vertex, its outgoing edges capture the dependencies of the associated TRIO component. For each component, we then generate a random propositional formula referring to its very dependencies, where each successor vertex in the graph, become a variable in the generated propositional formula.

The generation of random propositional formula is based on a Boltzmann sampler [29]. In a nutshell, Boltzmann samplers provide a means to generate combinatorial structures of a fixed size (e.g., lists, trees, and the like) that are drawn following a uniform distribution.

Appendix B: Convergence of failure propagation

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*.

Table 6 Truth table of Eq. 12

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

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:

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

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:

$$\mathcal{E}_{\mathbf{s}}[\![(p_1,\ldots,p_k)]\!] = (\mathcal{E}_{\mathbf{s}}[\![p_1]\!],\ldots,\mathcal{E}_{\mathbf{s}}[\![p_k]\!]) \tag{11}$$

$$\mathcal{E}_{\mathbf{s}}[\![s_i \wedge e]\!] = \mathcal{E}_{\mathbf{s}}[\![s_i]\!] \wedge \mathcal{E}_{\mathbf{s}}[\![e]\!] \tag{12}$$

$$\begin{aligned} \mathcal{E}_{\mathbf{s}}[\![s_i]\!] &= \mathbf{s}[\,i] \\ \mathcal{E}_{\mathbf{s}}[\![e_1 \wedge e_2]\!] &= \mathcal{E}_{\mathbf{s}}[\![e_1]\!] \times \mathcal{E}_{\mathbf{s}}[\![e_2]\!] \\ \mathcal{E}_{\mathbf{s}}[\![e_1 \vee e_2]\!] &= \mathcal{E}_{\mathbf{s}}[\![e_1]\!] + \mathcal{E}_{\mathbf{s}}[\![e_2]\!] \\ \mathcal{E}_{\mathbf{s}}[\![\neg e]\!] &= 1 - \mathcal{E}_{\mathbf{s}}[\![e]\!] \end{aligned}$$

Equation 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.

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:

$$\forall i \leq k, \quad \mathcal{E}_{\mathbf{s}}[\![p_i]\!] \leq s_i \tag{16}$$

$$\forall i \leq k, \quad \mathcal{E}_{\mathbf{s}}[\![s_i \wedge e_i]\!] \leq s_i$$

$$\forall i \leq k, \quad \mathcal{E}_{\mathbf{s}}[\![s_i]\!] \times \mathcal{E}_{\mathbf{s}}[\![e_i]\!] \leq s_i \tag{17}$$

Equation 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 6, once a component has failed, it remains failed forever.

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*.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors contributed equally to this work.

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).

Received: 21 May 2015 Accepted: 11 August 2015
Published online: 28 August 2015

References

1. Nygard MT (2007) Release it! Design and deploy production-ready software. Pragmatic Bookshelf, Dallas, Texas - Raleigh, North Carolina
2. Kazman R, Klein M, Barbacci M, Longstaff T, Lipson H, Carriere J (1998) The architecture tradeoff analysis method. In: Engineering of complex computer systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on. IEEE, Washington DC, USA. pp 68–78
3. Mosser S, Fleurey F, Morin B, Chauvel F, Solberg A, Goutier I (2012) 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, Washington, DC, USA. pp 400–406
4. Roman D, Gao X, Berre AJ (2011) Demonstration: SensApp — An Application Development Platform for OGC-based Sensor Services. In: Taylor K, Ayyagari A, Roure DD (eds). Proceedings of the 4th international workshop on semantic sensor networks, ssn11, Bonn, Germany, October 23, 2011. CEUR-WS.org, Aachen, Germany Vol. 839. pp 107–110. CEUR Workshop Proceedings
5. Havlik F, Havlik D, Egly M, Berre A, Grønmo R, van der Shaaf H, Modafferi S, Middleton S, Sabeur Z, Granell C, Esbrí MA, Lorenzo J, Schleidt K, Pielorzoo J (2013) Final recommendations for environmental enablers D4.4. http://cordis.europa.eu/fp7/ict/netinnovation/deliverables/envirofi/envirofi-d44.pdf, Deliverable, ENVIROFY Consortium
6. Cui X, Sun Y, Mei H (2008) Towards automated solution synthesis and rationale capture in decision-centric architecture design. In: Software architecture, 2008. WICSA 2008. Seventh working IEEE/IFIP conference on. IEEE Computer Society, Washington DC, USA. pp 221–230
7. Memmott J, Waser NM, Price MV (2004) Tolerance of pollination networks to species extinctions. Proc R Soc Lond Ser B Biol Sci 271(1557):2605–2611
8. Ferry N, Rossini A, Chauvel F, Morin B, Solberg A (2013) Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: O'Conner L (ed). CLOUD 2013: IEEE 6th International Conference on Cloud Computing. IEEE Computer Society, Washington DC, USA. pp 887–894
9. Ferry N, Song H, Rossini A, Chauvel F, Solberg A (2014) CloudMF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications. In: Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on. IEEE Computer Society, Washington DC, USA. pp 269–277
10. Erdös P, Rényi A (1959) On random graphs, I. Publ Math (Debrecen) 6:290–297
11. Watts DJ, Strogatz SH (1998) Collective dynamics of "small-world" networks. Nature 393(6684):440–442. ISSN 0028-0836
12. Concas G, Marchesi M, Pinna S, Serra N (2007) Power-laws in a large object-oriented software system. IEEE Trans Softw Eng 33(10):687–708. ISSN 0098-5589
13. Louridas P, Spinellis D, Vlachos V (2008) Power laws in software. ACM Trans Softw Eng Methodol 18(1):2:1–2:26. ISSN 1049-331X
14. Mahadevan P, Krioukov D, Fomenkov M, Dimitropoulos Xenofontas, Claffy KC, Vahdat Amin (2006) The internet as-level topology: Three data sources and one definitive metric. Comput Commun Rev (SIGCOMM) 36(1):17–26. ISSN 0146-4833
15. ISO/IEC/IEEE (2010) Systems and software engineering – vocabulary 24765. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5733833
16. Chauvel F, Song H, Ferry N, Fleurey F (2014) Robustness Indicators for Cloud-Based Systems Topologies. In: Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on. IEEE Computer Society, Washington DC, USA. pp 307–316
17. Kazman R, Asundi J, Klein M (2001) Quantifying the costs and benefits of architectural decisions. In: Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on. pp 297–306
18. Omerovic A, Solhaug B, Stølen K (2012) Assessing practical usefulness and performance of the prediqt method: An industrial case study. Inf Softw Technol 54(12):1377–1395
19. Lund M, Solhaug B, Stølen K (2011) A Guided Tour of the CORAS Method. In: Model-driven risk analysis: The CORAS approach. Springer, Berlin Heidelberg. pp 23–43. http://dx.doi.org/10.1007/978-3-642-12323-8_3
20. Wilkov R (1972) Analysis and design of reliable computer networks. IEEE Trans Commun 20(3):660–678. ISSN 0090-6778
21. Bigdeli A, Tizghadam A, Leon-Garcia A (2009) Comparison of network criticality, algebraic connectivity, and other graph metrics. In: Proceedings of the 1st Annual Workshop on Simplifying Complex Network for Practitioners, SIMPLEX '09. ACM, New York, NY, USA. pp 4:1–4:6
22. Caballero J, Kampouris T, Song D, Wang J (2008) Would diversity really increase the robustness of the routing infrastructure against software defects? In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008. The Internet Society, Reston, USA. http://www.isoc.org/isoc/conferences/ndss/08/papers/11_would_diversity_really.pdf
23. Gorbenko A, Kharchenko V, Tarasyuk O, Romanovsky A (2011) Using diversity in cloud-based deployment environment to avoid intrusions. In: Troubitsyna E (ed). Software engineering for resilient systems - third international workshop, SERENE 2011, Geneva, Switzerland, September 29-30, 2011. Proceedings. Springer, Berlin, Heidelberg Vol. 6968. pp 145–155. http://dx.doi.org/10.1007/978-3-642-24124-6_14, Lecture Notes in Computer Science
24. Manzano M, Calle E, Torres-Padrosa V, Segovia J, Harle D (2013) Endurance: A new robustness measure for complex networks under multiple failure scenarios. Comput Netw 57(17):3641 –3653. http://www.sciencedirect.com/science/article/pii/S1389128613002740, ISSN 1389-1286
25. Sydney A, Scoglio CM, Schumm P, Kooij RE (2008) ELASTICITY: topological characterization of robustness in complex networks. In: Murata M, Akan Ö (eds). 3rd International ICST Conference on Bio-Inspired Models of Network, Information, and Computing Systems, BIONETICS 2008, Hyogo, Japan, November 25-28, 2008. ICST / ACM. p 19. http://dx.doi.org/10.4108/ICST.BIONETICS2008.4713
26. Xing L, Amari S (2008) Fault tree analysis. In: Misra KB (ed). Handbook of performability engineering. Springer, London. pp 595–620
27. Zhou J, Stalhaane T (2004) Using FMEA for early robustness analysis of Web-based systems. In: Proceedings of the 28th Annual International Conference on Computer Software and Applications (COMPSAC 2004). IEEE Computer Society, Washington DC, USA Vol. 2. pp 28–29

28. Barabási A, Albert R (1999) Emergence of scaling in random networks. Science 286(5439):509–512
29. Duchon P, Flajolet P, Louchard G, Schaeffer G (2004) Boltzmann samplers for the random generation of combinatorial structures. Comb Probab Comput 13:577–625. ISSN 1469-2163