Fault-Detection Managers: More May Not Be the Merrier

Ghazal Zamani · Olivia Das

Received: 27 August 2019 / Accepted: 10 December 2020 / Published online: 20 February 2021
© The Author(s), under exclusive licence to Springer Nature B.V. part of Springer Nature 2021

Abstract A fault management system contains managers that detect faults as well as initiate recovery actions. Such management systems often come in an architecture that is not only a distributed one but also decoupled from the applications. Although an arrangement like this promotes scalability, it unfortunately makes the recovery of applications dependent on the fault management system itself. This work introduces two novel equations to meet the performance objectives of applications. To this end, we first create an equation that estimates the maximum number of jobs to be handled by an application instance for meeting a given performance objective. This formula is then used by admission control mechanism to restrict the number of jobs (targeted for operational application instances) to be allowed to enter the system. Next, we create a second equation that computes the response time distribution of an application. Thereafter, we develop a simulation model that predicts the impact of the failure of four sample fault management architectures on application's performance. Exploiting our equations, we compare the architectures in terms of three distinct ways of handling affected jobs when application instances fail—allow job loss; retry jobs resulting in overload; employ admission control to mitigate the overload. Our simulation results

show that boosting the number of managers may not always be beneficial; rather, it could possibly be the interconnection topology (i.e. the layout of interconnects linking the architectural components) of the management architecture, **together** with the model parameter values that may sometimes have a bigger role to play in the application's performance.

Keywords Modeling · Fault management architecture · Job retry · Resource overload · Admission control · Model parameter values

1 Introduction

Application providers employ load-balancing replication [1–5] for their applications. Replication distributes workload across multiple application instances, each often running on its own separate virtual machine (VM). Although an application can function—possibly with degraded performance—in response to application instance failures, replication is futile if proper mechanisms are not in place to detect failures and to recover from them. Organizations often like to use independent fault management system that comes with readily available monitoring and alarm support for application components [6–8]. Such self-contained management system decoupled from application eases development cycle while promoting software reuse. Yet, this benevolent scenario unfortunately makes the recovery of applications dependent on the fault management architecture and the status of its managers—We use the term,

G. Zamani · O. Das (✉)
Department of Electrical and Computer Engineering, Ryerson
University, 350 Victoria Street, Toronto, Ontario, Canada
e-mail: odas@ee.ryerson.ca

G. Zamani
e-mail: ghazal.zamani@gmail.com

recovery, to mean isolating and removing the failed application instances.

When an application instance fails, the jobs are affected. We group the affected jobs into two separate cases—(a) the jobs that were in the queue of the application instance, and the job that was being served when the failure occurred; (b) the jobs that were sent to the failed application instance by the load balancer (LB) as long as LB *did not know* about the failure. Case (b) may occur due to following reasons [9]:

- the management components responsible for notifying the load balancer about the failure are in failed status, **or**
- the management components responsible for detecting the failure are in failed status, **or**
- the failure is currently being detected and the load balancer is yet to know about the failure.

When the failure of an application instance occurs, the affected jobs are either lost, or resubmitted by LB to the remaining operational application instances once the LB *knows* about the failure. The resubmission may cause short-term overload in the system leading to performance degradation. The overload may be handled by employing an admission control mechanism to satisfy the performance objectives. Henceforth in the paper we will use the term *retry* to imply resubmission of jobs.

The goal of our paper is to predict the impact of fault management architecture on application performance. We leverage the modeling technique of [9] to compare the effect of fault management architectures in this work. A model developed using this technique can be solved for various performance measures, and the model solution can account for the management-architecture based coverage on application performance. Such a technique would benefit the application providers to relatively compare multiple fault management architectures for their applications and select the right one that meets the stipulated performance and availability measures. Although our model solution is simulation based, and hence may be computationally expensive as compared to an analytical counterpart, it is more general in terms of service time, inter-arrival time, failure time, detection time, and restart time distributions.

Our **contribution** in this paper is as follows: We develop two novel equations to meet performance objectives. To this end, we *first* create an equation that estimates the maximum number of jobs to be handled by

an application instance for meeting a given performance objective. This formula is then used by our admission control mechanism to restrict the number of jobs (targeted for operational application instances) to be allowed to enter the system. We explain this in Section 3.1. Next, we create a *second* equation that computes the response time distribution of an application. This equation enables us to compute the response time percentiles in the context of this work. We explain this in Section 3.2. We then develop models of four fault management architectures—*centralized*, *distributed*, *hierarchical* and *one-to-one*—using a pre-existing modeling technique [9]. Thereafter, exploiting our aforementioned equations, we comparatively analyze the effect of coverage of those architectures on application performance for three different workload scenarios—(i) jobs lost due to application instance failures; (ii) jobs retried thereby causing overload; (iii) admission control employed to mitigate the overload. We do so in Section 4.

The rest of the paper is organized as follows. Section 2 elaborates a prior technique [9] to model fault management architectures. It explains the theory our current work is based upon. Section 3 introduces our novel equations that assists us in meeting performance objectives. Section 4 describes the models of four fault management architectures and presents their relative comparison. Section 5 elucidates some related work and compares them with the current work. Section 6 delves into further discussion while enumerating the limitations of the current work and suggesting some future directions. Finally, Section 7 concludes the paper.

2 Background: Way to Model Fault Management Architecture

In this section, we elaborate the modeling technique of [9] in context of our current work. In this regard, first, we describe our application model. Second, we explain the components and the connections (i.e. relationships among the components) of our fault management architecture model. We do so using an example of a centralized fault management architecture. Third, we discuss how to model the failure behavior of components. Finally, we explain how to model the propagation of failure information through a fault management architecture.

2.1 The Application Model

An application provides certain kind of service to its end users. An application is an example of a SaaS software (Software-as-a-Service) owned by an application provider who chose to deploy it in a cloud. An application deployment in a cloud is composed of m application instances ($A1, A2, \dots, Am$). We assume that each application instance runs on its own VM. For our modeling purposes, we further assume that all the VM instances are of same type in terms of their processing speed. Similar model for an application has been assumed earlier by Calheiros et al. [10].

Figure 1 shows the application model as a network of queues. Each queue corresponds to an application instance hosted on its own VM. An application instance can fail and can be restarted. To restart a failed application instance, the failure needs to be detected first. Let λ denote the arrival rate of jobs. The load balancer (LB) is an infinite server and it is assumed to be failure-free. The LB distributes the workload equally among the application instances that it *believes* are operational—“*believes*” implies that LB has an impression that the application instances are operational but in reality some instances might be in failed state. This *belief* of LB about an application instance depends on the fault management architecture (described later) and the status of the management components at the time of failure (or at the time of restart completion) of the instance. If LB *believes* that m application instances are currently operational, then the arrival rate at each of those instances will be λ/m . Let μ be the service rate of each application instance.

2.2 The Fault Management Architecture Model

A separate fault management system can be utilized to monitor the health of the application instances [6]. The management system can detect and isolate a failure and can trigger actions such as automatic restart of a failed instance. It can also notify the load balancer about the status of the application instances which in turn re-distributes the workload accordingly.

Failures of instances can be detected by mechanisms such as heartbeats, and timeouts on periodic polls. Heartbeat messages from an application instance can be generated by a special heartbeat interrupt service routine which sends a message to one or more managers, every time an interrupt occurs, as long as the

Fig. 1 The Application Model (as network of queues) where the load balancer (LB) believes that m application instances are operational

instance has not crashed. If an instance cannot initiate heartbeat messages, then it may be able to respond to messages from the manager(s); we can think of these as status polls. The responses provide the same information as heartbeat messages. Once the heartbeat information is collected, it can be propagated to other managers and finally to the load balancer.

The fault management architecture model described here has three categories of components—application instances, managers, and the load balancer. There are two categories of relationships—*detect* and *notify*. We categorize these relationships according to the information they convey. We do so in a way that supports the analysis of *belief* of the status of the application instances at different points in the management system. A *detect* relationship from component a to component b implies that component b can detect the crash failure of component a , and can trigger automatic restart of component a . There is a non-zero time to detect the failure. A *notify* relationship from component a to component b implies that component a propagates the status data about itself (when component a is an application instance) or about other application instance(s) that it has collected or received (when component a is a manager) to component b . It is assumed that the notification happens in no time.

When the failure of an application instance occurs, the occurrence is first captured by the manager(s) monitoring the instance through a *detect* relationship. As soon as the failure is detected, the failed instance is restarted by the manager who first detected it. The manager(s) then propagate the failure information to the high-level managers (if they exist in the hierarchy) through *notify* relationships. The failure information is

finally propagated to the LB through *notify* relationships by the manager(s) who are connected to the LB. The LB then initiates system reconfiguration by re-distributing the workload among the application instances that it *believes* to be currently operational.

Once the restart of the failed instance is complete and the instance is again up and running, it *notifies* its manager(s) about its working status in no time. The managers, in turn, propagate the working status to the high-level managers (if they exist in the hierarchy) through *notify* relationships. Subsequently, the working status of the instance is propagated to the LB through *notify* relationships by the manager(s) who are connected to the LB. Then LB again performs system reconfiguration.

If a hierarchy of managers exists, high-level manager(s) can monitor low-level manager(s). When a low-level manager fails, the failure is first detected by the high-level manager(s) monitoring the low-level manager through a *detect* relationship. As soon as the failure is detected, the failed manager is restarted by the monitoring manager who first detected it. Once the restart of the failed manager is complete, it *notifies* its monitoring high-level manager(s) about its working status in no time. The status information of a manager does not need to be propagated to the LB. This is because LB is responsible for system reconfiguration in response to application instance failures, not manager failures.

In usual cases, both *detect* and *notify* relationships exist between two components a and b where a is the monitored component and b is the monitoring component. We will represent this by a *detect-notify* arc (a solid line with an open ended arrow) from component a to component b . This arc implies that the failure of component a will be *detected* by component b . When component a restarts, it *notifies* component b in no time. For a *detect-notify* arc (a, b) , let $d(a, b)$ be the time to detect the failure of component a by component b .

Figure 2 shows a sample centralized fault management architecture for the application model of Fig. 1. M1 has been introduced here as the central manager that monitors all the application instances ($A1, A2, \dots, Am$). M1 notifies LB about the status of the application instances. In Fig. 2, a rectangle represents a component. In this model, since M1 is not monitored by any other manager, its failure has to be detected and it has to be restarted by a human administrator. This situation can be modeled by a *detect-notify* arc from M1 to LB with $d(M1, LB)$ equal to the time to detect the failure of M1

by a human being. Usually this time $d(M1, LB)$ will be large enough in comparison to the $d(a, b)$ values of the other arcs (where these other arcs represent automatic detection without human intervention).

When an application instance fails, the jobs get affected. Regardless of the presence or absence of a fault management architecture, the jobs that were in the queue of the instance, and the job that was being served are affected. Let n_1 be the number of such jobs. When a fault management architecture comes into play, the *knowledge* of a load balancer about the failure depends on the status of the management components at the time of failure and the connections of the management architecture. This *knowledge* in turn affects the *number* of jobs, n_2 , that the load balancer (being unaware of the failure) sends to the failed application instance. Therefore, $n_1 + n_2$ are the affected jobs.

2.3 Modeling the Failure Behavior of Components

Each component (either an application instance or a manager) except LB has three states: UP, FND (Failed-Not-Detected), FDR (Failed-Detected-and-Restarted).

Let $(M1_i, M2_i, \dots, Ms_i)$ are the *operational* managers who are monitoring the health of component i . Since the failure is usually detected using heartbeat messages or timeouts on periodic polls, the time to detect the failure of component i by manager Mk_i is deterministically distributed with value $d(i, Mk_i)$. Let δ be the minimum of the timespans taken by the managers $M1_i, M2_i, \dots$,

Fig. 2 A sample Centralized Fault Management Architecture Model for the Application Model of Fig. 1. M1 is the central manager

Ms_i to detect the failure of component i . Then, $\delta = \min \{d(i, Mk_k) \text{ where } k = 1, 2, \dots, s\}$.

The state transition diagram of a component i is shown in Fig. 3. Each component i starts in the UP state. Once it fails, it transits to the state FND (Failed-Not-Detected). Let the random variable F_i represent the time to failure of component i . The failure can be detected after time δ at which point it is restarted. This is subject to the condition that at least one of the managers monitoring the component i is operational when component i fails. Otherwise, the component i will remain in FND state until one of its managers is operational again. Once the failure of component i is detected and it is restarted, it transits to the state FDR (Failed-Detected-and-Restarted). Let the random variable R_i represent the time to restart of component i . Once the restart is complete, the component transits back to the UP state.

An application instance can process jobs only when it is in UP state. Similarly, a manager can monitor other components only when it is in UP state.

2.4 Modeling Propagation of Failure Information through Fault Management Architecture

The load-balancer LB maintains a list containing the states of all the application instances. Each of these states can be either UP or FDR. These states of the application instances are the states that LB *believes* to be true. LB distributes the workload equally among the application instances that it *believes* are in UP state.

Next, we describe two situations and their consequences where the state of A_i (that LB *believes* to be true) becomes inconsistent with the actual state of A_i .

Situation 1: Let us assume that LB contains the state of application instance A_i as UP when A_i is actually UP. In this context, suppose A_i fails. Once A_i fails, the state of A_i that LB *believes* in (which is UP) becomes inconsistent with the actual state of A_i (which is FND).

Let an *operational path* from A_i to LB be a path that contains all *operational managers* between A_i and LB. This implies that the actual state of A_i gets conveyed to

Fig. 3 State transition diagram of a component i

LB, if such a path exists. Let P_1 be the set of *operational paths* (i.e. paths containing all *operational managers*) from A_i to LB at the time of failure of A_i .

$$\text{Let, } \tau = \min_{p \in P_1} \{ \text{the value of } d(\text{initial arc of path } p) \}$$

That is, τ is the minimum time to propagate the failed state of A_i , all the way to LB.

Here, n_2 (i.e. the number of jobs the load balancer sends to the failed application instance) will depend on the duration τ since for this duration LB will continue sending the jobs to A_i although A_i has already failed.

If P_1 is null, i.e. no operational path exists from A_i to LB at the time of failure of A_i , then LB continues to *believe* that A_i is UP although A_i has already failed. Here, n_2 will depend on the time until at least one such path becomes operational, or the time until A_i becomes operational (i.e. A_i comes back to UP state again).

Thus, n_2 entirely depends on the fault management architecture, status of the managers at the time of failure, and the failure detection times.

Situation 2: Let us assume that LB contains the state of application instance A_i as FDR when A_i is actually FDR. In this context when the restart of A_i is complete, A_i goes back to UP state. Once A_i is in UP state, the state of A_i that LB *believes* in (which is FDR) becomes inconsistent with the actual state of A_i (which is UP).

Let P_2 be the set of *operational paths* (i.e. paths containing all *operational managers*) from A_i to LB at the time when A_i became operational.

If P_2 contains at least one such *operational path*, the state of A_i that LB has, becomes consistent with the actual state of A_i in no time.

If P_2 is null, then LB *believes* that A_i is failed although A_i is actually in UP state. As a result, LB does not send any job to A_i . This will result in higher response time for jobs in comparison to the case where the jobs were sent to A_i as well.

3 Equations to Meet Performance Objectives

3.1 Estimating the Maximum Number of Jobs to Be Handled by an Application Instance

The models developed in this paper handle the jobs that are affected due to failure. When the failure of an application instance occurs, the affected jobs (i.e. $n_1 + n_2$) can be handled in multiple ways. Here we discuss three possible ways:

- **Scenario-1:** The affected jobs are lost. This scenario may not be desirable for many systems.
- **Scenario-2:** The affected jobs are retried by the load balancer to the remaining operational application instances once the load balancer *knows* about the failure. This causes short-term overload which may lead to performance degradation.
- **Scenario-3:** To mitigate the overload of Scenario-2, an admission control mechanism is enforced. This will help to maintain acceptable performance of the application. We assume that the admission control mechanism is integrated in the load balancer. We model each application instance as M/M/1 queue with arrival rate λ and service rate μ . The response time (RT) distribution for such a queue is exponentially distributed with cumulative distribution function given as:

$$P(RT \leq \tau) = 1 - e^{-(\mu - \lambda)\tau}$$

Leveraging the above function, we develop an equation that estimates the maximum number of jobs (i.e. load) to be handled by an application instance to meet a given response time (i.e. performance) objective. This information is further used by our admission control mechanism to restrict the number of jobs (targeted for operational application instances) to be allowed to enter the system.

Let us take an example. If our response time objective is: “90% of jobs should be processed in less than or equal to 1 second”, then each operational application instance must abide by this objective. In this case, $\tau = 1$ and $P(RT \leq \tau) = 0.9$. Therefore, the maximum number of jobs, λ_{max} , that can be handled by an application instance can be computed as:

$$e^{-(\mu - \lambda_{max})1} = 1 - 0.9 = 0.1$$

$$\lambda_{max} = \ln(0.1) + \mu$$

When the load balancer *knows* that p application instances are operational, the admission control allows at most $(p * \lambda_{max})$ jobs into the system. Rest of the arrived jobs are not admitted to the system. The number p is adjusted based on the *knowledge* of the load balancer.

3.2 Computing the Response Time Distribution of an Application

The models developed in this work predict the response time distribution of an application as well as the probability of admitted jobs (where applicable). This is in addition to the mean throughput and the normalized throughput loss (where applicable).

Let N denote the total number of job arrivals and C denote the total number of jobs completed in one simulation run. Let T be the logical time of one such run. Then the mean throughput is C/T . During each simulation run, we record the response time, RT_i , of each job i . At the end of each simulation run, we compute the number of jobs whose response time is less than or equal to the threshold τ .

$$\text{For job } i, \text{ let } z_i = 1 \text{ if } \{RT_i \leq \tau\}$$

$$= 0 \text{ otherwise}$$

Let the random variable RT denote the response time. We estimate the response time distribution as:

$$P(RT \leq \tau) = \sum_{i=1}^N \frac{z_i}{N}$$

In Scenario-1, the jobs are lost due to application instance failures. Let, NTL denote the normalized throughput loss. It is the fraction of jobs that are lost [11]. In Scenario-2, there is no job loss. Hence NTL is 0. In Scenario-3, some jobs were not admitted by the admission control mechanism. In this case the probability of admitted jobs—fraction of jobs admitted out of the total number of arrived jobs—is a relevant measure.

4 Models of Four Sample Fault Management Architectures and their Comparative Analysis

4.1 Models of Four Fault Management Architectures

This section presents the models of four sample fault management architectures of an application shown in Fig. 1 with four application instances A1, A2, A3, A4. The fault management architectures are *centralized*, *distributed*, *hierarchical* and *one-to-one*. They vary in their number of managers from 1 to 4. We assume the same cost for every manager. The architectures thus differ in their cost. We choose these architectures since they represent a reasonable diversity in terms of the number

Table 1 Model parameters and their default values

Parameter	Parameter Value
Inter arrival time—exponentially distributed with parameter λ	$\lambda = 100$ jobs/s
Service time—exponentially distributed with parameter μ	$\mu = 40$ jobs/s
Time to failure for a manager or an application instance is Weibull distributed with scale parameter α , and shape parameter β	$\alpha = 1.04$ h $\beta = 0.85$
Time to restart a manager or an application instance—deterministically distributed	100 s
Time to detect a failure automatically by a manager—deterministically distributed	120 s
Time to detect a failure by a human administrator—deterministically distributed	600 s

of managers, the number of interconnections, and the number of points of failure. Our choice was motivated by the classification given in [12–14].

Table 1 shows the model parameters and their default values. They are motivated by the work of Poola, Ramamohanarao and Buyya [15]. We set that the inter-arrival time of jobs is exponentially distributed with parameter $\lambda = 100$ jobs/sec and the service time to be exponentially distributed with parameter $\mu = 40$ jobs/sec. Failures of computing resources in a distributed system including cloud has been observed to follow a Weibull distribution in several studies [15–17]. Being motivated by that, we adopt the time to failure for a manager or an application instance to be Weibull distributed with scale parameter $\alpha = 1.04$ hours and shape parameter $\beta = 0.85$. Similar to [15, 18], we adopt the time to restart (boot/startup) a manager or an application instance to be 100 sec (deterministically distributed). The monitoring frequency of a manager in a fault management system is typically a user-defined parameter value. Here we have assumed the time to detect a failure automatically by a manager to be deterministically distributed with a custom value of 120 sec. The time to detect a failure by a human administrator is assumed 600 sec. We adopt the response time objective as follows: “90% of jobs should be processed in less than or equal to 1 second”.

Next, we explain our sample fault management architectures through the following four cases.

- **Case-1:** This case resembles a *centralized* management architecture [19, 20]. A single manager handles all application instances, makes

Fig. 4 Case-1: *Centralized* management architecture. Manager M1 is a central manager. The failure of M1 has to be detected by a human administrator

decisions and initiates reconfiguration. Fig. 4 shows our sample *centralized* architecture. All the application instances A1, A2, A3, A4 are monitored by a single manager M1. However, M1 is not monitored by other managers. It is therefore assumed that the failure of M1 has to be detected by a human administrator. Since the manager failure is not automatically detected, the $d(M1, LB)$ corresponding to the *detect-notify* arc from M1 to LB is assumed to be 10 min, i.e. $d(M1, LB) = 600$ s. The $d(a, b)$ value for each of the remaining *detect-notify* arcs from component a to component b is assumed 120 s.

- **Case-2:** This case resembles a *distributed* management architecture [21]. It has multiple management domains with a manager for each domain. When information from another domain is needed, the manager communicates with its peer managers to retrieve it. Figure 5 shows our sample *distributed* architecture. Here the application instances A1, A2 are monitored by manager M1 in one domain whereas A3, A4 are monitored by manager M2 in another domain. The peer domain managers M1 and M2 also monitor each other. If one manager has already failed, and subsequently, the second manager fails, then this failure of the second manager has to be detected by a human administrator. $d(M1, LB)$ and $d(M2, LB)$ is 600 s. The $d(a, b)$ value for the remaining *detect-notify* arcs is assumed 120 s.

Fig. 5 Case-2: *Distributed* management architecture. M1 and M2 are two peer domain managers. M1 monitors A1 and A2, M2 monitors A3 and A4., M1 and M2 monitor each other

- Case-3:** This case resembles a *hierarchical* management architecture [22]. The hierarchical architecture also relies on multiple managers (domain managers) and introduces the concept of the manager of managers (MOM). Each domain manager is connected to the rest of the network only through the MOM. The MOM operates on a higher hierarchical level, retrieving information from and coordinating the domain managers. Unlike the distributed architecture, the domain managers do not communicate directly. Figure 6 shows our sample *hierarchical* architecture. Similar to Case-2, here also the application instances A1, A2 are monitored by manager M1 in one domain whereas A3, A4 are monitored by manager M2 in a second domain. However, unlike Case-2, the domain managers M1 and M2 do not monitor each other. They are rather monitored by yet another manager M3—a MOM. The MOM however is not monitored by any manager. The failure of MOM, M3, is therefore not automatically detected; it has to be detected by a human administrator. $d(M3, LB)$ is therefore 600 s. The $d(a, b)$ value for the remaining *detect-notify* arcs is assumed 120 s.

- Case-4:** This case resembles a *one-to-one* management architecture. In this architecture, every application instance is managed by its own manager. However, none of the managers are managed either by each other or by any separate manager. Figure 7 shows our sample *one-to-one*

architecture. Here the application instance A1 is monitored by manager M1, A2 by M2, A3 by M3, and A4 by M4. If an application instance fails, its manager will be able to detect that failure automatically in 120 s. The managers M1, M2, M3, M4 are not monitored; their failures have to be detected by a human administrator. Thus $d(M1, LB)$, $d(M2, LB)$, $d(M3, LB)$, and $d(M4, LB)$ are 600 s each.

4.2 Comparative Analysis of Four Fault Management Architectures

We simulate the application model together with each of the four aforementioned fault management architecture models, i.e. the models for Case-1 (Fig. 4), Case-2 (Fig. 5), Case-3 (Fig. 6) and Case-4 (Fig. 7). We do so in the context of each of the three aforementioned workload scenarios—jobs lost due to application instance failures (Scenario-1), jobs retried thereby causing overload (Scenario-2), and admission control employed to mitigate the overload (Scenario-3). The simulation time for a simulation run was 10000 seconds. We ran 10 simulation runs corresponding to each of the four fault management architecture models (Case-1, 2, 3 and 4) and reported the mean of each performance measure obtained from those ten runs.

4.2.1 Scenario-1: Jobs Lost Due to Application Instance Failures

In this scenario, jobs are lost when an application instance fails.

Fig. 6 Case-3: *Hierarchical* management architecture. M1 and M2 are two peer domain managers. M1 monitors A1, A2. M2 monitors A3, A4. M3 is a manager of managers (MOM). It monitors M1, M2. The failure of M3 has to be detected by a human administrator

Fig. 7 Case-4: One-to-one management architecture. Each application instance A1 to A4 is monitored by its own manager M1 to M4. The failures of the managers M1, M2, M3, M4 have to be detected by a human administrator

4.2.1.1 We analyze the effect of arrival rate λ at 90, 100 and 110 jobs/sec (keeping the other parameter values same as Table 1). Figure 8 shows the effect on (a) the mean throughput; (b) to (d) the response time distributions; and (e) the normalized throughput loss. In Fig. 8, we see that our response time objective is met by all the four architectures (Case-1 to 4). However, one architecture is better than the other depending on the arrival rate.

For $\lambda = 90$ jobs/sec

In Fig. 8(b) we see that the response time for a job is within 1 sec for about 98% of the arrived jobs for Case-2. This is better than the other three cases. In 8(e) we see that the fraction of job loss is minimum for Case-2 and consequently highest throughput (8(a)). Therefore Case-2 is the architecture of choice although it has lower number of managers (2 managers) compared to Case-3 (3 managers) and Case-4 (4 managers) owing to differences in connectivity.

For $\lambda = 100$ jobs/sec

In Fig. 8(e) we see that the fraction of job loss is about the same for both Case-2 and 3 and consequently their throughput is similar (8(a)). However, in Fig. (c) Case-2 is the best in terms of the percentage of arrived jobs each of whose response time is within 1 sec. Therefore, again Case-2 is the architecture of choice here.

For $\lambda = 110$ jobs/sec

In Fig. 8(d), the response time distribution is almost similar for Case-2 and 3. However, the fraction of job loss is higher for Case-2 than Case-3 and consequently Case-3 has higher throughput (8(a)). Therefore Case-3 is the architecture of choice.

4.2.1.2 Next we analyze the effect of scale parameter α of Weibull distribution (i.e. the failure distribution) for

every manager at 1.04, 1.54 and 2.04 hours (keeping the other parameter values same as Table 1). Figure 9 shows the effect on (a) the mean throughput; (b) to (d) the response time distributions; and (e) the normalized throughput loss. In Fig. 9, we see that the response time objective is met by all the four architectures (Case-1 to 4). However, one architecture is better than the other depending on the value of scale parameter α . Case-1 (which has a single point of failure due to centralized architecture) is the worst overall.

For $\alpha = 1.04$ hours for every manager

This is the case when managers are failing frequently compared to other α values considered here. In terms of fraction of job loss (9(e)) and hence with respect to throughput (9(a)), Case-2 and 3 are almost the same. However, in 9(b) the percentage of jobs each finishing within 1 sec is better for Case-2. Hence Case-2 is the best candidate here.

For $\alpha = 1.54$ or 2.04 hours for every manager

Here, Case-3 is the best in terms of throughput (9(a)) as well as response time percentiles (9(c) and 9(d)).

This suggests that if we buy managers which fails less often with $\alpha = 1.54$ or 2.04 hours, Case-3 hierarchical architecture with 3 managers will be the choice. However, if the managers fail more often with $\alpha = 1.04$ hours, Case-2 with less number of managers but with more inter-connectivity will be the choice.

4.2.1.3 We now analyze the effect of failure detection time by each manager at 60 sec, 120 sec and 180 sec (keeping the rest of the parameter values same as Table 1). Figure 10 shows the effect on (a) the mean throughput; (b) to (d) the response time distributions; and (e) the normalized throughput loss. Here, overall we see that Case-2 has similar or lower throughput than Case-3 (10(a)). However, Case-2 is similar or better than Case-3 in terms of response time percentiles (10(b), (c), (d)). Although all the four architectures meet the response time objective, we would choose Case-3 over Case-2 if we want higher system throughput. On the other hand if we want more percentage of jobs each finishing within 1 sec, we would prefer Case-2 architecture.

Job loss is not a desirable feature for many systems. Still, if job loss is present and if we consider the default parameter values given in Table 1 for evaluating the performance of our application, we find from Figs. 8(a), 8(c), 8(e), 9(a), 9(b), 9(e) and 10(a), 10(c), 10(e) that overall Case-2 with 2 managers is the best fault management architecture in terms of job loss (minimum or similar to other cases), average throughput (maximum or similar to other cases), and response time percentiles.

Fig. 8 Scenario-1. Jobs lost due to application instance failures. No admission control. (a) Mean throughput. (b), (c), (d) Response Time distributions. (e) Normalized throughput loss. Shown for all four architectures (Case-1 to 4) across arrival rate at 90, 100, 110 jobs/s

4.2.2 Scenario-2: Jobs Retried Thereby Causing Overload

In this scenario, jobs are not lost when an application instance fails. Rather, the unprocessed jobs are retried

when the load balancer comes to know about an application instance failure.

Since the number of retries and thereby the amount of overload is influenced by the arrival rate of jobs, we analyze the effect of arrival rate λ at 90, 100 and 110

Fig. 9 Scenario-1. Jobs lost due to application instance failures. No admission control. (a) Mean throughput. (b), (c), (d) Response Time distributions. (e) Normalized throughput loss. Shown for all

four architectures (Case-1 to 4) across α parameter of Weibull distribution (failure distribution) for every manager at 1.04, 1.54 and 2.04 h

jobs/sec (keeping the other parameter values same as Table 1). Figure 11 shows the effect on (a) the mean throughput; and (b) to (d) the response time distributions. In Fig. 11, we see that the response time objective is met by all the four architectures for $\lambda = 90$ and 100

jobs/sec (11(b), 11(c)). However, for $\lambda = 110$ jobs/sec, Case-2 is the only architecture that meets the objective (11(d)).

For $\lambda = 90$ jobs/s

Figure 11(b) suggests that Case-3 is the best with respect to response time percentile. However, Fig. 11(a) suggests that Case-2 is the best with respect to average system throughput. The architecture of choice here will depend on the priority of the application provider.

For $\lambda = 100$ jobs/sec and 110 jobs/sec Figure 11(a) suggests that Case-2 has similar or higher throughput than other cases. Figure 11(c) and (d) suggest that Case-2 will meet the 1 s response time requirement for higher number of jobs compared to other cases. So, Case-2 will be the architecture of choice.

For this scenario, if we consider the default parameter values given in Table 1, (where $\lambda = 100$ jobs/sec), Case-2 with 2 managers is the best fault management architecture.

4.2.3 Scenario-3: Admission Control Employed to Mitigate Overload Caused by Job Retrials

Similar to Scenario-2, in this scenario the unprocessed jobs are retried when the load balancer comes to *know* about an application instance failure. However, unlike Scenario-2, an admission control mechanism based on Section 3.1 is employed to mitigate overload caused by job retrials.

Again, the job arrival rate being a notable influencer on the amount of overload, we analyze the effect of arrival rate λ at 90, 100 and 110 jobs/sec (keeping the other parameter values same as Table 1) like Scenario-2. Figure 12 shows the effect on (a) the mean throughput; (b) to (d) the response time distributions; and (e) the probability of jobs admitted in presence of job retrials when an admission control mechanism is in place. Comparing 12(b), 12(c), 12(d) against 11(b), 11(c), 11(d) respectively, we find that higher number of jobs meet the response time restriction of 1 sec or less in presence of admission control. In absence of admission control for $\lambda = 110$ jobs/sec (Fig. 11(d)), Case-2 was the only architecture meeting the response time objective. On the contrary, when admission control becomes active for $\lambda = 110$ jobs/sec (Fig. 12(d)), all the four architectures now meet the response time objective. The advantage of admission control comes at a cost of not admitting a certain number of jobs. Figure 12(e) shows the fraction of jobs that were admitted into the system.

For $\lambda = 90$ jobs/sec Figure 12(e) suggests that fraction of jobs admitted is the highest for Case-3 and Fig. 12(b) suggests that Case-3 is the best in terms of

response time percentile. Case-3 architecture is therefore the preferred one.

For $\lambda = 100$ jobs/sec and 110 jobs/sec Figure 12(c), (d), (e) suggest that Case-2 is the architecture of choice since it admits higher number of jobs than others (12(e)) and has the better response time percentile (12(c), 12(d)).

For this scenario, if we consider the default parameter values given in Table 1, (where $\lambda = 100$ jobs/sec), Case-2 with 2 managers is the best fault management architecture in this situation.

5 Related Work

The performance of systems in presence of failures has traditionally been predicted using analytical models. In this regard, an exact monolithic Markov model has often been employed for prediction, for example in [23, 24]. This approach, however, suffers from largeness and stiffness problem [25]. An alternative solution has been to employ hierarchical Markov Reward models which consists of a higher-level Markov dependability model, and a bunch of lower-level performance related models—one for each state in the dependability model. The use of Markov Reward models is, however, limited to scenarios where the failure and restarts occur at much slower time scale than the processing times. This limits its use for the current work since the restart times and failure detection times may be similar—in time scales—as the application processing times [25].

Previously, several works have modeled performance in presence of failures [26–28]—Trivedi et al. [26] has modeled the normalized throughput loss of multi-processor systems; Ramani et al. [27] has modeled the performance of messaging services in distributed systems; Zimmermann et al. [28] has predicted performance using Stochastic Petri Nets. Although they modeled the failure of applications, **none** of them modeled the failure of monitoring manager components. Besides, they **never** considered modeling the **unified** effect of the **five critical** factors—fault-management architecture, time taken by a manager component to automatically detect a failure, time taken by a human administrator to detect a failure, the time to failure of a component, and the time to restart of a component—**together** on application performance.

We are further motivated by the four limitations of [9] to undertake our current work. The *first* limitation of [9] is

Fig. 10 Scenario-1. Jobs lost due to application instance failures. No admission control. (a) Mean throughput. (b), (c), (d) Response Time distributions. (e) Normalized throughput loss. Shown for all

four architectures (Case-1 to 4) across time—to detect a failure by a manager—of 60, 120, 180 s

that it assumes jobs are lost when application instances fail. In contrast, we incorporate a mechanism for *retry* of lost jobs since job completion is critical in real-time

transactions. We also include a mechanism for *admission control* to mitigate job overload caused by the retries. The *second* limitation of [9] is that it uses artificial parameter

Fig. 11 Scenario-2. Jobs retried after *knowing* application instance failures. No admission control. (a) Mean throughput. (b), (c), (d) Response Time distributions. Shown for all four architectures (Case-1 to 4) across *arrival rate* at 90, 100, 110 jobs/s

values to reach the model results. In contrast, the parameter values in our work are driven by an existing study in [15]. We leverage this study to generalize the assumption of exponential distribution—in [9]—to Weibull distribution for time to failures of components. This leads to more accurate interpretation of our results. The *third* limitation of [9] is that the number of managers stay the same in all their fault management architectures. This resulted in equal cost for all the management architectures and led to the obvious conclusion that the best configuration is the one where all managers are connected to each other and to all application instances. On the contrary, in the current work, we compare management architectures that differ in the number of managers and hence management cost. This subsequently helps us demonstrate how a certain interconnection topology (i.e. the layout of interconnects linking the architectural components) of the management architecture, **in concert with** a certain set of model parameter values, can meet system performance and reliability objectives while still maintaining lower fault management cost. The *fourth* limitation of [9] is that it restricts the analysis to average measures. We overcome this limitation by using a percentile measure for response time, given that the

response time percentile as a metric is of greater importance than average response time since it is more desirable to reduce the variability of a system's response time, rather than minimizing the average response time [29]. Another prior work on fault-management architecture [13] also motivates our work. Similar to [9], the research of [13] does not consider failure detection times, job loss, job retry or admission control; it is rather restricted to the assumption of exponentially distributed time to failures and repairs for components, and computes only average measures. Table 2 provides an overview of the comparison of a selected bunch of modeled aspects among the prior modeling endeavors, and our current work.

6 Discussion

6.1 Interconnectivity Alone Is Not the Only Deciding Factor!

Although among the four fault-management architectures, the architecture with the highest connectivity (Fig. 5, Case-2) seems to be the architecture of choice, the

Fig. 12 Scenario-3. Jobs retried after *knowing* application instance failures. Admission control employed. (a) Mean throughput. (b), (c), (d) Response Time distributions. (e) Probability of jobs admitted. Shown for all four architectures (Case-1 to 4) across *arrival rate* at 90, 100, 110 jobs/s

connectivity alone does not guide this decision. We explain it next.

Eyeballing the management components M1 to M4 in Fig. 4 (Case-1) to Fig. 7 (Case-4), we find that in Fig.

5 (Case-2), M1 and M2 are connected to each other implying that they monitor each other. Consequently, if one manager fails, the other can detect the failure and restart the failed manager automatically, without human

intervention. In contrast, if we look at Fig. 6 (Case-3), we find that although the failure of M1 and M2 can be detected and restarted by M3, the failure of M3 itself cannot be detected by either M1 or M2; rather, it can only be detected and restarted through human intervention. Similar argument holds for the management components in Fig. 4 and Fig. 7—their failures can only be detected and restarted through human intervention. Thus, given that we have selected 120 sec for automatic detection versus 600 sec for human detection (see Table 1) for analyzing the architectures of Fig. 4 to Fig. 7, it seems M1 and M2 in Fig. 5 are more available (due to lower time to automatic failure detection) as opposed to either M1 in Fig. 4, or M3 in Fig. 6, or even M1-M4 in Fig. 7 (requiring higher time to failure detection by human).

On the other hand, how the availability of managers will impact performance depends on the time to failure of the components, their restart time, as well as other model parameters; for example, as per the results in Fig. 8, when the arrival rate is lower (90 jobs/sec, Fig. 8(b)), Case-1 is better than Case-4. However, for relatively higher arrival rates (100 jobs/sec, Fig. 8(c); 110 jobs/sec, Fig. 8(d)), Case-4 is better than Case-1.

Thus, it is a difficult task to rank the management architectures based on their connectivity alone; it is rather the combined effect of the architecture **as well as** the model parameter values that impact the application's performance. Accounting for this combined effect has been an important aspect that motivated the need of our simulation models for architecture analysis.

6.2 Automating the Analysis

To automate our analysis, we have developed an architecture description layer that enables to specify the components and connections of a fault management architecture for a given application model. This layer consists of a set of entities written using the object-oriented features of the python programming language at two levels—user level and non-user level. At the user level, a user of the layer can use the given methods to specify interconnections between instances of Load Balancer, Managers, and Application components. The user can also specify the probability distributions for time to failure of Manager and Application components as well as the time to restart of these components. At the non-user level, the user is transparent to the methods that are invoked to simulate behaviors such as failure/restart behavior of managers and application

components, queueing behavior of resources, monitoring behavior of managers, job retry behavior of the load balancer once it becomes aware of the failures of application components, admission control behavior in the system, and workload generation. Our architecture description layer sits on top of a third-party open-source discrete event simulation environment SimPy [30] written in Python. We choose SimPy for its discrete event simulation functionalities that include features allowing processes and shared resource interactions. Note that the architecture description layer is helpful in modeling fault-management architectures wherever the *detect* and *notify* relationships among the components hold (see Section 2).

6.3 Limitations and Future Work

The value of including fault management architecture in the performance analysis of an application is threefold—*first*, it accounts for the failures and restarts of the application instances and the managers, *second*, it includes delays to detect the failures, and *third*, it considers the limitations of the fault management architecture itself in terms of connectivity among the managers. Our research focuses on comparing fault management architectures in terms of their *models* to obtain quick high-level insights. We refrained from a real-world empirical study in this work since our initial assessment in this regard proved to be an onerous task that was too difficult and costly to be undertaken.

Although our analysis conjectures that the distributed management architecture with two managers (Fig. 5, Case-2) is the architecture of choice in current context, it does not tell us how increasing the number of managers further—in the distributed architecture itself—would affect the application performance. This still remains a problem to be investigated.

The examples provided in this work consider only failures of application instances and managers. This is because, usually, the virtual machines and physical machines are highly available, and they are usually managed by the cloud providers. However, if the application provider also wants to monitor the virtual machines and/or physical machines allocated for their applications, then that needs to be included in the analysis as well. In this work, when an application instance fails, we assume that a new application instance (on a new VM) is not provisioned. Rather, the system transits to a degraded mode until the restart of the failed application instance is complete. We recommend future investigation to include the impact of such provisioning. For the simplicity of our

Table 2 Selected modeled aspects compared across related work and this work

Modeling work	Models monitoring manager?	Models unified effect of five critical factors together?	Models retry of lost jobs?	Models admission control of jobs to mitigate job overload?	Computes response time distribution (not average measure)?
[23, 24]	X	X	X	X	X
[26]	X	X	X	X	X
[27]	X	X	✓	X	X
[28]	X	X	X	X	X
[9]	✓	✓	X	X	X
[13]	✓	X	X	X	X
This work	✓	✓	✓	✓	✓

✓ denotes Yes

X denotes No

analysis, we assume failure-free load balancer in our sample architectures. Moreover, we do not consider the failures of network interconnects; we rather assume that a fault manager always correctly detects a failure—This is due to a constraint in the architecture description theory of [9] itself, which our work is based upon. We recommend future investigation to address these limitations.

This paper stands on the fault-management architecture modeling technique of the work in [9]. It would be an interesting endeavor to examine whether this modeling technique can be adapted in the domain of autonomic management of cloud resources [31] on one hand, as well as fog and edge computing [32, 33] on the other. One more thing to note is that our current work focuses on four traditional fault management architectures which are, centralized, distributed, hierarchical and one-to-one; In presence of big data demands, the validity of such traditional configurations and their impact on application performance remains open to be investigated.

A recent work [34] demonstrates that the learning of the front-end (i.e. user interface) of an application by its end-users can impact the performance of the application. Motivated by this work, we recommend a future investigation that explores the combined effect of the end-user learning of the front-end of applications together with the failure of fault management architectures, on applications' performance.

7 Conclusions

This work develops two novel equations to meet performance objectives of applications within the context of fault management architectures—It first introduces an

equation that estimates the maximum number of jobs to be handled by an application instance for meeting a given performance objective. This formula is then used by the admission control mechanism to restrict the number of jobs (targeted for operational application instances) to be allowed to enter the system. Next, it introduces a second equation that computes the response time distribution of an application. The two equations are then employed to comparatively analyze the effect of coverage of four fault management architectures on application performance for three different workload scenarios—(i) jobs lost due to application instance failures; (ii) jobs retried thereby causing overload; (iii) admission control employed to mitigate the overload.

The managers in all the architectures are assumed to have the same cost for our modeling purposes. We find that, given the current context, the distributed architecture with two managers (Fig. 5, Case-2) is an overall best architecture of choice (among all four architectures) in the three workload scenarios. This indicates that less number of managers with higher connectivity among them may outperform more number of managers with less connectivity for a given set of parameter values—Thus, buying a greater number of managers might not necessarily be a better decision always; the interconnection topology of the management architecture **together** with the model parameter values **may** affect the application performance as well, sometimes.

Acknowledgements We would like to thank the editors and anonymous reviewers for their valuable comments and suggestions to help and improve our research paper. We would like to thank NSERC Canada for their financial support.

References

- Cardellini, V., Colajanni, M., Philip, S.Y.: Dynamic load balancing on web-server systems. *IEEE Internet Comput.* **3**, 28–39 (1999)
- Grozev, N., Buyya, R.: Multi-cloud provisioning and load distribution for three-tier applications. *ACM Transactions Autonomous Adaptive Systems.* **9**(3), 13 (2014) 1–13:21
- Nuaimi KA, Mohamed N, Nuaimi MA and Al-Jaroodi J (2012) A survey of load balancing in cloud computing: challenges and algorithms. In: 2nd Symposium on Network Cloud Computing and Applications (NCCA), pp 137–142
- Vaquero, L.M., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review.* **41**(1), 45–52 (2011)
- Tu, M., Ma, H., Xiao, L., Yen, I.L., Bastani, F., Xu, D.: Data placement in P2P data grids considering the availability, security, access performance and load balancing. *J Grid Computing.* **11**(1), 103–127 (2013)
- da Rosa, R.R., Lehmann, M., Gomes, M.M., Nobre, J.C., da Costa, C.A., Rigo, S.J., Lena, M., Mohr, R.F., de Oliveira, L.R.: A survey on global management view: toward combining system monitoring, resource management, and load prediction. *J Grid Computing.* **17**(3), 473–502 (2019)
- Roblitz, T., Schintke, F., Reinefeld, A., Barring, O., Lopez, M.B., Cancio, G., Chapeland, S., Chouikh, K., Cons, L., Poznanski, P., Defert, P.: Autonomic management of large clusters and their integration into the grid. *J Grid Computing.* **2**(3), 247–260 (2004)
- Birje, M.N., Manvi, S.S.: Wigrimma: a wireless grid monitoring model using agents. *J Grid Computing.* **9**(4), 549–572 (2011)
- Zamani G, and O. Das. (2017) Impact of a Fault Management Architecture on the Performance of a Component-based System. 13th European Dependable Computing Conference (EDCC 2017), Geneva, Switzerland, September, 87–94
- Calheiros RN, Ranjan R, and Buyya R (2011) Virtual machine provisioning based on analytical performance and QoS in cloud computing environments. In: proceedings of ICPP'11. Pp 295–304
- Trivedi, K.S., Muppala, J., Woollet, S.P., Haverkort, B.R.: Composite performance and dependability analysis. *Perform. Eval.* **14**, 197–215 (1992)
- Stamatelopoulos F, Roussopoulos N, and Maglaris B (1995) Using a DBMS for hierarchical network management, in: Proceedings of the Engineer conference, NETWORKD + INTEROP'95
- Das, O., Woodside, C.M.: Analyzing the effectiveness of fault-management architectures in layered distributed systems. *Performance Evaluation, Elsevier.* **56**(2004), 93–120 (2004)
- Das, O., Woodside, C.M.: Modeling the coverage and effectiveness of fault-management architectures in layered distributed systems. In: Proceedings International Conference on Dependable Systems and Networks, pp. 745–754. Washington, DC (2002). <https://doi.org/10.1109/DSN.2002.1029020>
- Poola, D., Ramamohanarao, K., Buyya, R.: Enhancing reliability of workflow execution using task replication and spot instances. *ACM Transactions Autonomous Adaptive Systems.* **10**(4), 30 (2016) 1–30:21
- Javadi, B., Abawajy, J., Buyya, R.: Failure-aware resource provisioning for hybrid cloud infrastructure. *J Parallel Distributed Computing.* **72**(10), 1318–1331 (2012)
- Tang, X., Li, K., Liao, G.: An effective reliability-driven technique of allocating tasks on heterogeneous cluster systems. *Cluster Computing, Springer.* **17**(4), 1413–1425 (2014)
- Ming M, Humphrey M (2012) A performance study on the VM startup time in the cloud. IEEE 5th International Conference on Cloud Computing (CLOUD 2012), Honolulu, USA, June, 423–430
- Cassel, L.N., Patridge, G., Westcott, J.: Network management architectures and protocols: problems and approaches. *IEEE J. Select. Areas Comm.* **7**(7), 1104–1114 (1989)
- Marshall, R.: The Simple Book: an Introduction to Internet Management, 2nd edn. Prentice-Hall, Englewood Cliffs (1994)
- Leinwand, A., Fang, K.: Network Management: a Practical Perspective. Addison-Wesley, Reading (1993)
- Dupuy, A., Sengupta, S., Wolfson, O., Yemini, Y.: Design of the Netmate network management system. In: Krishnan, I., Zimmer, W. (eds.) *Integrated Network Management.* Elsevier, Amsterdam (1991)
- Ciardo, G., Blakemore, A., Chimento, P.F., Muppala, J.K., Trivedi, K.S.: Automated generation and analysis of Markov reward models using stochastic reward nets. In: *Linear Algebra Markov Chains, and Queueing Models, Ima Volumes In Mathematics and Its Applications*, vol. 48, pp. 145–191. Springer, New York (1993)
- Trivedi, K. S., Malhotra, M., & Fricks, R. M. (1994). Markov reward approach to performability and reliability analysis, in: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Durham, NC, 1994, pp. 7–11
- Bolch, G., Greiner, S., de Meer, H., and Trivedi, K.S. (2006) Queueing networks and Markov chains - modelling and performance evaluation with computer science applications, 2nd Ed. Wiley, 2006
- Trivedi, K.S., Andrade, E.C., Machida, F.: Combining performance and availability analysis in practice. *Adv. Comput.* **84**, 1–38 (2012)
- Ramani, S., Goseva-Popstojanova, K., Trivedi, K.S.: A framework for performability modeling of messaging services in distributed systems. In: Proc. of 8th IEEE Intl. Conference on engineering of complex computer systems (ICECCS 02), pp. 25–34 (2002)
- Zimmermann, A. et al. (2000). Petri net modelling and performability evaluation with TimeNET 3.0. International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Springer Berlin Heidelberg, pp. 188–202
- Broadwell PM (2004) Response time as a performability metric for online services. Report No. UCB//CSD-04-1324. Computer Science Division (EECS), University of California, Berkeley

30. SimPy (2017) Discrete event simulation library in python. Website: simpy.readthedocs.io (Accessed June 7, 2017)
31. Singh, S., Chana, I., Buyya, R.: STAR: SLA-aware automatic management of cloud resources. *IEEE Transactions on Cloud Computing*, **8**(4), 1–14 (2020)
32. Mahmud, R., Ramamohanarao, K., Buyya, R.: Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions. *ACM Computing Survey*, **53**(4), 88 (2020) 1–88:43
33. Aslanpour, M. S., Gill, S. S., & Toosi, A. N. (2020). Performance evaluation metrics for cloud, fog and edge computing: a review, taxonomy, benchmarks and standards for future research. *Internet of Things*, 100273
34. Das, O., & Das, A. (2020). CogQN: a Queueing model that captures human learning of the user interfaces of session-based systems. 17th international conference on quantitative evaluation of SysTems (QEST 2020), short paper (springer, LNCS series), august 2020, Vienna (to be held online due to COVID-19)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.