PCRAFT: Capacity Planning for Dependable Stateless Services

Rasha Faqeh[‡], André Martin[‡], Valerio Schiavoni[*], Pramod Bhatotia[†], Pascal Felber[*] and Christof Fetzer[‡]
[*]University of Neuchâtel, Switzerland. E-mail: first.last@unine.ch
[†]Technical University of Munich, Germany. E-mail: first.last@in.tum.de
[‡]Technical University of Dresden, Germany. E-mail: first.last@tu-dresden.de

Abstract—**Fault-tolerance techniques depend on replication to enhance availability, albeit at the cost of increased infrastructure costs. This results in a fundamental trade-off: Fault-tolerant services must satisfy given availability and performance constraints while minimising the number of replicated resources. These constraints pose *capacity planning* challenges for the service operators to *minimise replication costs without negatively impacting availability*.**

To this end, we present PCRAFT,[1] a system to enable capacity planning of dependable services. PCRAFT's capacity planning is based on a hybrid approach that combines empirical performance measurements with probabilistic modelling of availability based on fault injection. In particular, we integrate traditional service-level availability mechanisms (active route anywhere and passive failover) and deployment schemes (cloud and on-premises) to quantify the number of nodes needed to satisfy the given availability and performance constraints.Our evaluation based on real-world applications shows that cloud deployment requires fewer nodes than on-premises deployments. Additionally, when considering on-premises deployments, we show how passive failover requires fewer nodes than active route anywhere. Furthermore, our evaluation quantify the quality enhancement given by additional integrity mechanisms and how this affects the number of nodes needed.

I. Introduction

Dependability is a must-have requirement for modern Internet-based services. These services must be *highly available, integrity protected, secure* and offer high assurance of *performance* to users, in terms of response time and throughput. Service providers can use replication to achieve high availability and more reliable services, and deploy them either using an on-premises cluster or on a public cloud. However, such techniques regardless of the chosen deployment *(i)* increase the complexity and the costs of the supporting infrastructure, and *(ii)* degrade the observed performance of the applications. To address the problem of performance assurance, the service providers provision extra physical resources. However, over-provisioning increases the operational costs without improving availability. This represents an important problem: *how to achieve the required level of availability and performance while maintain affordable costs?* We tackle this problem directly by providing a capacity planning process.

We consider a scenario of a service that uses a cluster of server nodes that run stateless [1] or soft-state [2] applications. The service has a target throughput and availability requirements. To fulfil these requirements, applications are replicated on different nodes. However, nodes crash and are replaced with new ones, mainly to enhance or guarantee availability of the services. This can be implemented using simple replication techniques such as *active route anywhere* (ARA) or *passive failover* (PF). *Active route anywhere* provisions more nodes to process users requests than required to meet the target performance. The extra nodes fulfil this objective as long as sufficiently many stay available. *Passive failover* also requires extra nodes to be provisioned but they are passively waiting in a standby pool: when an active node fails, it is replaced by one from the pool. After repairing the node, it is returned to the pool to handle new failures.

We consider two physical deployment schemes: *on-premises cluster* and *public cloud*. The scheme has direct consequences on the fault-tolerance properties as well as the associated costs. Cloud providers largely rely on the passive failover approach, restarting virtual machines (or lately containers) of a failed node on a functional one. Doing so, they can meet service level agreements (SLAs) under several classes of *nines* [3]. Note that the service provider only has to pay for the nodes actively participating in the cluster without the need to pay for the nodes that passively exist in the pool. Conversely, when compared to an on-premises deployment, resource sharing in the cloud (*e.g.*, co-located virtual machines) impacts the observed performance, leading to a degraded response time and unexpected variations [4]. This degradation is perceived by clients as service unavailability if it exceeds a certain (negative) threshold. On the other hand, on-premises solutions must provide and pay for both the active nodes in the cluster and the passive nodes in the pool.

Even when the service is available and serves requests, because of transient hardware faults these requests may be incorrectly processed and produce incorrect results. Therefore, mechanisms such as instruction level redundancy [5] are often used to protect the integrity of the execution by detecting which executions are incorrectly processed, yet at the price of decreased performance and corresponding capacity.

Our system, PCRAFT, proposes a capacity planning process that is based on a combination of empirical performance measurements with availability and integrity probabilistic modelling. We consider the costs of deploying dependable applications in terms of both the number of nodes used and

[1]**P**erformant, **C**heap, **R**eliable and **A**vailable **F**ault **T**olerance.

the integrity of the service provided. This simpler cost model allows us to easily reason about the total cost of ownership of a cluster and to consider the number of nodes as a building block for more complex schemes.

PCRAFT integrates two availability techniques in its process: *passive failover* and *active route anywhere*. PF consists of loosely coupled and independent servers with failover capabilities. To tolerate f failures, at least f extra nodes should be available in a standby pool. By default, these nodes are in *cold* mode, *i.e.*, they are only started when needed. ARA uses active replication to deploy f additional, fully functional nodes behind a load-balancer that dispatches requests to all of them. The service performance is ensured only if at most f nodes crash. In addition, PCRAFT uses *instruction level redundancy* (ILR) [5] to protect node integrity from transient faults. ILR replicates data flow instructions and executes two instruction streams in parallel, leveraging the instruction-level parallelism of modern CPUs. To do so, PCRAFT relies on the HAFT [6] framework, which additionally exploits hardware transactional memory (HTM) [7] to recover from faults.

Previous studies on capacity planning mainly modelled the system performance with varying workloads and resource conditions [8], [9], [10], [11] assuming an always available reliable infrastructure. Studies that quantify the availability in the event of different types of failures ignored the effect of failures on the performance [12], [13]. Similarly, the combined effect of faults on availability and performance [14], [15], [16], [17] did not quantify the number of nodes needed to ensure both the performance and availability levels.

In this paper, we propose the following contributions. *(i)* We introduce PCRAFT, a capacity planning process that is based on a combination of empirical experimentation and modelling to quantify the number of nodes needed to assure availability and performance levels. *(ii)* We develop a collection of probabilistic models to measure the availability of services when incorporated with various failures and recovery behaviour via different fault-tolerance approaches and physical deployment schemes. *(iii)* We measure the integrity of the service by combining integrity models with fault-injection.

The remainder of the paper is structured as follows. §II provides additional information on dependability schemes and related work on capacity planning processes. We describe our fault model and present our approach in §III. Probabilistic models are discussed in §IV. We show in §V how one can use PCRAFT's capacity planning process to generate the associated costs, before concluding in §VI.

II. Related work

We review dependability approaches as well as studies related to capacity planning.

A. Dependability approaches

Service dependability encompasses various measures. We focus on integrity and availability. Crash failures degrade the service availability. Integrity failures degrade both the service reliability and availability.

Availability. Availability enhancing approaches attempt to harden against crash failures. Crash failures implement fail-stop behaviour, where the faulty hardware component stops executing the algorithm. A standard approach to achieve high availability is the use of state machine replication [18]. The server application is replicated actively on $2f + 1$ nodes to tolerate f nodes crashes. If a minority of replicas crashes, the surviving replicas keep serving requests. Due to the high runtime and infrastructure costs, this approach is usually restricted to protect stateful applications.

In the context of stateless applications, highly available systems [19] replicate server applications either using passive failover or active replication schemes. *Passive failover* consists of loosely coupled and independent servers with failover capabilities which are connected to a monitoring service (*e.g.*, heartbeats). To tolerate f failures, at least f extra nodes should be available in a standby pool. The monitoring service detects server crashes and initiates a failover to restart/migrate the service using a node from the pool. The failover time, *i.e.*, the time needed to replace the failed node and to reconfigure the new node, directly affects the availability of the service. The failover can be implemented using a fast switchover to a *hot* node from the pool, already started with the application configured and its state periodically updated (*e.g.*, VMware fault tolerance). Alternatively, a delayed switchover to a *cold* node not yet started is also possible (*e.g.*, VMware high availability).

In contrast, *active replication* schemes deploy for the service f additional, fully functional nodes behind a load-balancer that dispatches requests to all of them. The service performance is ensured only if at most f nodes crash. The load balancer distributes requests in two ways. With *sticky* sessions, user requests are routed to a predetermined physical node for processing (user session data are cached locally), but possibly leading to load imbalances. *Active route anywhere* does not restrict requests to specific nodes, and load is equally balanced. However, user session data is not cached: locality information is lost, as the probability for subsequent requests to route to the same physical node is low. PCRAFT integrates both PF in cold mode and ARA in its process.

Transient faults. Hardware failures are not restricted to crashes. Transient faults could manifest as arbitrary state corruptions [20], data loss [21] and, possibly, outages [22]. Surprisingly, transient faults occur at high rates and reappear frequently after the first occurrence [23], [24]. Servers can protect the memory and caches using error correcting codes (ECC). However, protecting the CPU computation and registers is hard. Byzantine fault tolerance (BFT) [25] protects against silent data corruptions, transient faults and malicious attacks. However, BFT is not broadly adopted since it introduces considerable costs due to performance and management overheads [26].

Other hardening approaches attempt to reduce the replication and runtime overheads. These approaches add redundancy locally at the level of processes, threads, and instructions.

Process level redundancy creates one slave redundant process for each application master process in the program, and synchronises state on system calls [27]. Redundant multi-threading [28] executes code in redundant threads instead of processes. However, both approaches require deterministic multi-threading. *Instruction level redundancy* (ILR) [5] replicates data flow instructions to create a shadow stream that shares the same memory state with the master stream but uses an exclusive copy of CPU registers. The two streams execute in parallel, leveraging the instruction-level parallelism of modern CPUs. The master stream updates the memory after comparing the memory update results with the shadows. ILR does not restrict threads interleaving and hence allows for non-determinism in applications. However, ILR replicates at instruction level: fault detection is limited to faults in the CPU registers or execution unit. HAFT [6] is a state-of-art framework that hardens server applications against data corruptions using instruction level redundancy [5] to detect faults and hardware transactional memory (HTM) [7] to recover from them. It extends the LLVM compiler framework [29] to create self-checking applications by replicating the application data instructions and inserts periodic integrity checks wrapped inside HTM-based transactions. Any mismatch between the two execution streams is revealed by the inserted integrity checks and the execution of the application is then flagged as corrupted. PCRAFT uses the HAFT approach to protect node integrity.

B. *Dependability-costs studies*

Capacity planning requires estimating the adequate size of different resources to ensure the dependability of the service. Modelling techniques are well suited to address this problem, as we survey next.

Performance. Several studies use queuing theories and networks to predict the workload demands on resources such as CPU, memory and network bandwidth to support future computational needs [9], [11]. Similarly, [8], [10] focus on workload characterisation of the arrival and service rates to estimate the throughput and response time. These studies use white-box analytical models to represent the server hardware/software and take into account the different resources of the system (CPU, memory, disk, threads, etc). As analytical models require domain expert knowledge and a large number of factors make the models complex, only those factors with the highest impact on performance, *i.e.*, which can tolerate minor errors, are considered in the performance predication.

Machine-learning based approaches [30], [31] adapt a black-box approach (*i.e.*, zero-knowledge about the system) using empirical models. While these models provide more accurate performance predictions, they are not portable across different hardware and deployments. Similarly, PCRAFT uses empirical measurements to predicate the service performance using specific software and hardware node configurations.

Availability. Several studies focused on estimating the effect of failures using stochastic models. Failures happen at different granularities: *(i)* CPU, memory [13], *(ii)* operating system, power supply, hypervisor and VM [32], [33], or *(iii)* cluster, data-centre and cloud provider [34], [12]. State and non-state space modelling are two main approaches. State space models, such as continuous-time Markov models (CTMC), stochastic Petri nets and stochastic networks [12], [13], [32], [33] depict the system failure and repair behaviour using the states and rates to represent occurrence of events. Non-state models [32], [34] capture the conditions leading to failures considering the structure of components. Computationally-wise, these are usually cheaper. In PCRAFT, we focus on service unavailability regardless of which component failure caused the node to crash. We use CTMC to represent PCRAFT's availability models and capture the dependency between components, such as automated repair for components and failover in the cloud deployment.

Performability. Performability studies model the effect of (un)availability and performance on a single metric. They can take the form of a monolithic model, or a hierarchical model to reduce the possibly generated state space. PCRAFT uses a hybrid approach for performability, which allows us to construct availability models that can be fed with performance results from the empirical measurements instead of using performance modelling. In [14], authors quantify the effect of workload changes (*e.g.*, arrival and job service rate), fault-load (physical machine failure rates) and system capacity on service quality when deployed in the cloud. They consider the unavailability and the service response time as two different quality-of-service (QoS) metrics. They use CTMC models and include both warm [35] or cold [12] standby nodes. To quantify the QoS metrics, they assume the same number of nodes in each pool (warm/cold), scaling both pools by the same factor. PCRAFT can scale each pool independently to match certain performance and availability criteria. As opposed to our proposal, they do not support active replication. In [15], authors propose a hierarchical analytical model for both availability and latency, including correlated failures. Their model can be leveraged to quantify the service response time and the success probability of client requests without using queueing models to predict the latency. Additionally, they quantify the effect of adding more VMs to the service on the response time and service unavailability, with an increasing arrival request rate. PCRAFT tackles this problem by adding more nodes to cope with both performance and availability requirements.

III. System design

We discuss in this section our assumptions about the fault model and we present the overall architecture and operation mode of PCRAFT.

A. *Fault model*

We assume a dependable service built using a cluster of server nodes with three sources of failures: (1) Availability failures at the node level regardless of the cause. We assume fail-stop failures that either crash nodes or make them non-responsive, *e.g.*, failing hardware, software, disks, memory. We also assume that server nodes fail independently but do not exclude that many nodes can fail simultaneously. We exclude failure types that cause multiple nodes to fail due to a single failure, *e.g.*, network failures or software bugs that deterministically crashes applications. (2) Performance failures: requests cannot be served due to limited capacity, *i.e.*, the number of available nodes is too low. (3) Integrity failures: nodes are available but requests are served incorrectly without any error notification, in particular silent data corruption (SDC) [5] which leads to non-malicious Byzantine behaviour. SDCs are transient hardware faults caused by single event upsets, *e.g.*, one bit-flip in a CPU register or miscomputation in a CPU execution unit. Note that memory and caches are protected against bit-flips due to the use of ECC which is assumed to exist in all modern servers.

A service is *dependable* only if *(i)* the availability is above a predetermined threshold, *(ii)* whenever the service is available, there is sufficient capacity to ensure performance, and *(iii)* the service data integrity is protected.

B. Architecture

We consider a distributed n-tier service, with a front-end load balancer, a middle-tier cluster of server nodes and a connected back-end persistent storage. We focus on the middle-tier (*i.e.*, web, application, caching servers) to be available and well-performing. We assume the load balancer to be always available (*e.g.*, using replication [19]) and the back-end resources to be able to scale accordingly.

The load balancer intercepts the client requests and distributes them to the homogeneous cluster of servers nodes. A node becomes saturated when serving the maximum throughput (requests per unit time) with an acceptable latency (application-dependent response time threshold). Adding more requests to the node after saturation will cause the latency perceived by users to be monotonically increasing as more requests are added to the node's local queue (sharing model [15]). Alternatively, the server prematurely rejects the requests if the server is saturated—assuming no queues (constant bit rate model [15])—with users subsequently re-sending the rejected requests. In PCRAFT, we adapt the constant bit rate model for server nodes to control the upper bound of the latency perceived by users.

C. Dependable services

The service consists of a cluster of nodes. Its throughput is the sum of the throughput of each node, while the latency is the average latency of all nodes that constitute the cluster. The service is available if at least a single node in the cluster is available, but with a degraded performance. To consider the service dependable, it must fulfils both the availability constraints (*e.g.*, three 9's) and the performance constraints (*e.g.*, target throughput and response time threshold). Additionally, the integrity mechanisms must be implemented at the node level to enhance the integrity of the served requests.

To meet our performance requirements, we can predict—based on single node performance—the number of nodes needed. Typically, we first assume that all nodes are always available. Since nodes can actually fail, in a second step we increase the availability by over-provisioning the number of nodes in the cluster. Over-provisioning might take the form of ARA in which fully functional extra nodes are added to the service cluster, or simply by replacing a failed node from a standby pool of nodes with PF. The number of nodes needed for the over-provisioning varies based on the type of deployment—on-premises or in the cloud. This is due to the availability mechanisms which are implemented in the cloud to assure the availability level for a single node that must meet the SLA. However, single node provisioning is typically insufficient to meet the availability constraints of a dependable service. Finding the adequate number of nodes in each case requires solving the capacity planning problem for this service.

D. Capacity planning process

The process consists of two-phases. First, we empirically benchmark the performance of a single server node. Second, we model the availability behaviour and parameterise the models using the values from the first phase to identify how many server nodes we need to ensure the dependability requirements.

Experimentation phase. Assuming an always available node, we profile the service that consists of a single node along two dimensions: performance and integrity. First, we benchmark the performance of both the native application ($native$) and the integrity protected version of the application (ft). We refer to the throughput of the single node in both cases as NdT_{native} and $NodT_{ft}$, respectively. Note that usually $NodT_{ft} < NodT_{native}$. The service target throughput $SerT$ is defined by the service provider as ($Num_{native} \times NodT_{native}$) or ($Num_{ft} \times NodT_{ft}$). For different node types ($native$ and ft), we can calculate Num_{native} and Num_{ft} as an estimation for the number of nodes needed in the cluster to fulfil the service target performance assuming nodes do not fail and without any over-provisioning of nodes. Note that usually $Num_{ft} > Num_{native}$ to fulfil the same $SerT$. Second, we inject integrity faults with a service that uses $native$ and ft nodes to benchmark the integrity behaviour in each case. The results of the fault injection is subsequently used to estimate the integrity behaviour of applications in the presence of integrity faults.

Modelling phase. We build availability models for a cluster of nodes that implements over-provisioning using ARA or PF mechanisms with a cloud or on-premises deployment. The model can be used to first estimate the availability level achieved by the basic number of nodes Num calculated in the previous phase. Then, by comparing the availability achieved to the target availability defined by the service provider, we over-provision iteratively the number of nodes in the cluster until the availability constraint is met. In addition to availability models, we build integrity models for $native$ and *ft* to estimate the integrity behaviour of nodes in the cluster when deployed in the cloud.

IV. Implementation

In the second phase of the process, we build continuous-time Markov chain (CTMC) models a probabilistic model checker tool called PRISM [36]. The models, which capture the availability and the integrity behaviour, consist of possible states and possible transitions between them with their respective rates (failure and repair rates). We assume all rates have inter-event times that are exponentially distributed [37]. CTMC suffers from well-known state space explosion problems. To reduce the generated number of states, we follow two approaches. First, we attempt to aggregate states together. Specifically, we use homogeneous nodes in the cluster and so they exhibit similar failure and repair behaviour. It is thus not necessary to distinguish between which node in the cluster failed. Rather, we can simply keep track of up nodes ($UpNodes$). Second, performability analysis generally uses a hierarchy of models instead of monolithic ones in order to combine both availability and performance and hence reduce the number of generated states. In this paper, instead of solving performance models, we use the empirical measurements to feed the availability models, which further reduces the number of states needed.

For our models, we explore the following server node types: *(i) native* nodes without any software/hardware hardening mechanisms implemented at the node level, and *(ii) ft* nodes which are integrity protected using the HAFT [6] approach. Additionally, we consider two deployments, in the *cloud* and *on-premise*, as well as two over-provisioning techniques, *active route anywhere* and *passive failover*.

Availability models. In the availability models (Figure 1), we inject hardware crash faults at different rates and consider the different over-provisioning techniques and deployments schemes. The two node types do not differ in their availability models at the node level because they use the same hardware. However, the performance achieved by each type is different and we need to use a higher number of *ft* nodes compared to $native$ nodes to fulfil the same performance constraints. This results in different availability levels achieved in a cluster by each type of node (see later in Figure 6).

Passive failover. Figure 1(a) presents the state machine diagram for the availability models of PF deployed in the cloud or on-premises. The PF technique assumes the existence of a pool that has a number of *cold nodes* that are turned off and hence do not fail. With a cloud deployment, the number of nodes passively waiting in the pool is unlimited ($Pool = \infty$) compared with concrete value ($Pool \geq 0$) for the on-premises case. The cluster initially consists of $UpNodes$. We start with a number of nodes that fulfil the performance constraint assuming always available nodes, *i.e.*, $UpNodes = Num$ (❶). A fault $\lambda_{HWCrash}$ crashes a node in the cluster (❷), hence reducing the number of $UpNodes$. Because $UpNodes$ nodes can crash independently, this rate is multiplied by the number

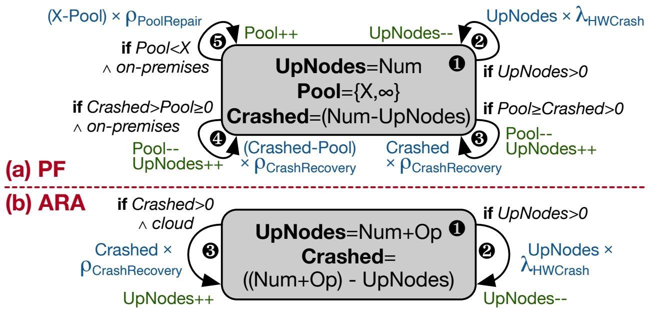

Fig. 1: State machine for a cluster of nodes using passive failover (top) or active route anywhere (bottom) as over-provisioning in the availability models for the cloud and on-premises deployments. Rectangles represent the states while arrows are transitions with associated rates. The system fails at rates of λ's and recovers at rates of ρ's.

of nodes that can be affected by such a fault. A crashed node can be recovered by replacing it with a node from the pool with a rate of $\rho_{CrashRecovery}$ (❸). This rate is affected by the number of available nodes in the pool (❸ and ❹). This would increase the $UpNodes$ while, additionally, decreasing the $Pool$ size in an on-premises deployment. The crashed nodes from the pool may be repaired (❺) and returned back to the pool to use if needed at rate of $\rho_{PoolRepair}$, which would increase the number of available nodes inside the pool. Note that ❹ and ❺ are special cases for the on-premises deployment.

To calculate the availability achieved by such a model, we use "rewards" to represent the time spent in each state in a one year time period. The cluster traverses different states according to the failures and recovery rates, and the time spent in a state where $UpNodes = Num$ represents the cluster availability. If the time spent outside this state does not exceed the downtime specified by the availability constraints (*e.g.*, three 9's availability = 8.77 hours downtime per year), the cluster is considered available and well-performing.

Active route anywhere. Figure 1(b) presents the state machine diagram for the availability models of ARA deployed in the cloud or on-premises. In ARA, in addition to the initial number of nodes needed to fulfil performance constraints (Num), we additionally use over-provisioned nodes (OP) which are actively participating in the cluster. Therefore, the number of active nodes in the cluster is $UpNodes = Num + OP$ (❶). A fault $\lambda_{HWCrash}$ crashes a node in the cluster (❷), hence reducing the number of $UpNodes$. Unlike PF, this rate is multiplied by all active nodes including the over-provisioned nodes ($UpNodes$). In a cloud deployment (❸), the failed node is replaced automatically by another node at rate of $\rho_{CrashRecovery}$, which increments $UpNodes$.

We calculate the cluster availability by considering the time spent in a state where $UpNodes \geq Num$. Consequently, at most OP nodes can fail simultaneously without violating the availability and performance constraints. Note that the cluster can have $UpNodes < Num$ at any time, but it is considered

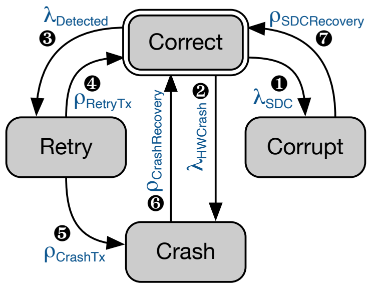

Fig. 2: State machine for integrity models of the different nodes.

available and well-performing as long as it does not violate the availability constraint.

Integrity models. In the integrity models, we inject integrity faults considering the different node types and deployments schemes. *ft* nodes implementing the HAFT approach have two modes of execution. HAFT implements instruction level redundancy to detect any violation to computation integrity. The first mode implements a fail-stop model, specifically, once a violation is detected, computation is stopped (ft_{ilr}). The second mode targets the availability, specifically, after an integrity failure is detected, instead of aborting, the execution is retried using transactions (ft_{tx}). Figure 2 presents the state machine diagram for integrity models of a node deployed in the cloud or on-premises. The node starts with *Correct* state. A transient fault can result in corruption of the state (SDC), crash of the application or masking of the fault. If not masked, a transient fault transfers the node in a *Corrupt* state at rate of λ_{SDC} (❶) or in a *Crash* state at rate of $\lambda_{HWcrash}$ (❷). Note that using ft_{ilr} nodes, this rate also includes the crashes resulting from aborting the application after detecting a transient fault. A ft_{tx} node detects transient faults at rate of $\lambda_{Detected}$ and transfers the node into *Retry* state (❸). A node in a *Retry* state is able to either recover the state at rate of $\rho_{RetryTx}$ and revert the node back into *Correct* state (❹), or if retry is not successful, abort execution and transfer the node into *Crash* state at rate of $\rho_{CrashTx}$ (❺). Both types of nodes do not have any mechanism to recover from crashed states. However, if deployed in the cloud, a node with *Crash* state is automatically replaced by another node to match the SLA agreement at a rate of $\rho_{CrashRecovery}$ (❻). When deployed on-premises, assuming enough resources in the pool, the node transfers back to *Correct* state. Additionally, the corruption of a node may be manually detected at a rate of $\rho_{SDCRecovery}$, which reverts the node back to *Correct* state (❼).

The integrity level achieved is captured by measuring the normalised time spent by the node in each state: *Correct* as correct time, *Corrupt* as corrupted time, *Crash* combined with *Retry* as downtime. Note that over-provisioning of nodes does not help to reduce the time in *Corrupt* state, since it produces multiple nodes with similar integrity behaviour. Alternatively, a different integrity protection technique should be implemented at the node level.

V. Evaluation

In this section we illustrate how PCRAFT's two phase methodology can be used. In a first step we benchmark a single server node deployed with *native*, ft_{ilr} and ft_{tx} nodes using a stateless web server application and two soft-state applications. Then, we parameterise the models built in PCRAFT to calculate the availability and the integrity behaviour at the cluster level and decide the required capacity.

Our evaluation answers the following questions: (1) How much performance, *i.e.*, throughput and latency, can be achieved by a single node of types *native*, ft_{ilr} and ft_{tx}? (2) What is the effect of the deployment scheme (cloud vs. on-premises) on the availability level achieved by a single node, without over-provisioning? (3) What is the effect of using a cluster of nodes on the availability achieved, without over-provisioning? (4) How many extra nodes are needed to ensure a given availability and performance constraints when using ARA and PF with different deployment schemes? (5) How a single node of types *native*, ft_{ilr} and ft_{tx} deployed in the cloud would behave when transient faults exist?

A. Experimental settings

All experiments use a dedicated on-premises deployment. The experimental part attempts to identify the performance of a single node $NodT$, which can then be used to identify the number of nodes (Num) required to achieve the cluster target performance ($SerT$) assuming always the available nodes. As an actual cloud deployment might differ experimentally in the performance achieved by each node $NodT$, it can use a similar experimental path as presented here to obtain Num. In this paper, we want to study the pure effect of over-provisioning and node types on the availability of the cluster. We therefore assume $NodT$ to be the same for nodes deployed in the cloud or on-premises, and we use the same number of nodes (Num) in each deployment to achieve $SerT$.

Each server node has an Intel Xeon E3-1270 v5 CPU clocked at 3.6 GHz. The CPU has 4 cores with 8 hyper-threads (2 per core) and 8 MB of L3-cache. The server has 64 GB memory running Ubuntu 14.04.5 LTS with Linux 4.4. The workload generators run on a server with two 14-core Intel Xeon E5-2683 v3 CPUs at 2 GHz with 112 GB of RAM and Ubuntu 15.10. All machines have a 10 Gb/s Ethernet NIC connected to a dedicated network switch.

We use the following three real-world applications: Apache web server (v2.2.11) [38], memcached (v1.4.21) [39] and Redis (v2.8.7) [40]. Furthermore, we deploy the applications in three variants: *native* unmodified application, ft_{ilr} built using the HAFT LLVM tool chain [6] with ILR, and ft_{tx} which additionally execute the application inside transactions.

B. Experimental measurements

Apache Web Server. Our first set of experiments study the Apache `httpd` web server (Figure 3). We evaluate the throughput vs. latency ratio and the overall CPU usage. We first measure the achievable throughput and latency of the three node variants under test, and the CPU utilisation

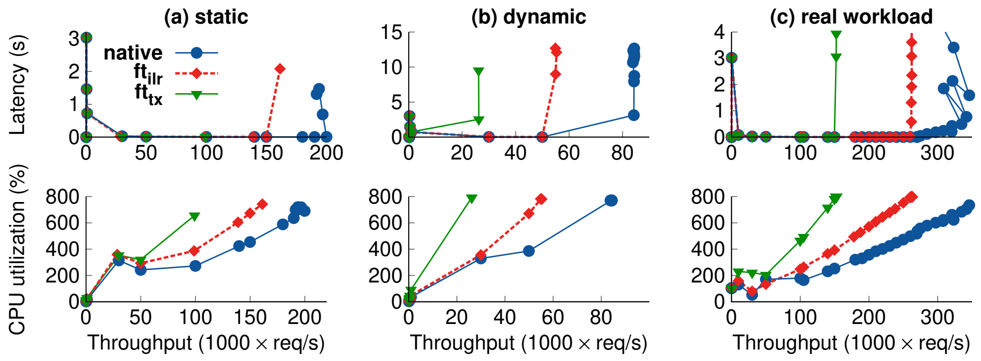

Fig. 3: Apache throughput vs. latency (top) and CPU utilisation (bottom) with 3 different workloads: static (a), dynamic (b) and real-world (c).

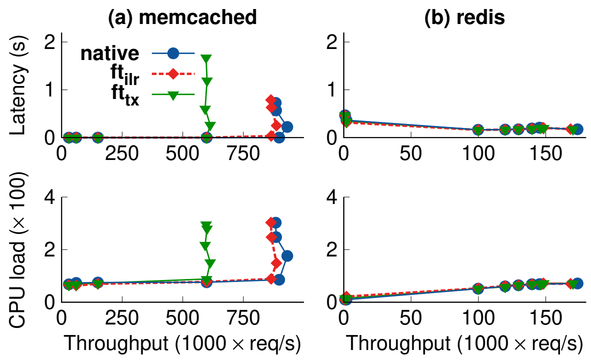

Fig. 4: Memcached (a) and Redis (b): throughput vs. latency (top) and CPU load (bottom).

during the execution of the benchmarks. We use the *wrk2* workload generator [41] to measure the throughput and latency based on fixed request rates issued to the master server. The following three workloads are used. *(i) static content:* the web server fetches static content (such as images or CSS files); *(ii) dynamic content:* the web server fetches dynamic content generated by PHP scripts (v5.4.0); and *(iii) real workload:* the web server operates under real-world conditions by retrieving a WordPress blog page with a MySQL server database at the back-end. We gradually increase the submission rate of HTTP requests until the response times hit unacceptable levels (*e.g.*, > 1 second).

Figure 3a shows the results for static content. The x-axis shows the measured response rate of the submitted requests while the y-axis shows the corresponding latency. The measured response rate is often lower than the introduced request rate when the system is saturated, giving the impression of the line going backwards suddenly since the data points are sorted by the introduced request rate which is constantly increasing, as seen in top graphs of Figure 3.

For dynamic content, Figure 3b depicts the results to fetch a PHP script that solely employs an empty for-loop iterating 10^3 times to basically simulate some CPU-intensive workload. Lastly, Figure 3c presents the results under real workload for blog-like web pages. As expected, the $native$ execution outperforms the other variants, with 200 thousands requests per seconds (kreq/s) to fetch static content. With ft_{tx} we reach only half the throughput, *i.e.*, 100 kreq/s, whereas ft_{ilr} improves the performance (up to 155 kreq/s for static content) but without the ability to properly handle detected errors. For dynamic content, we observe similar trends. The $native$ mode excels with a peak of roughly 86 kreq/s, followed by ft_{ilr} and ft_{tx}, respectively at 52 and 23 kreq/s. Surprisingly, for blog-like web pages, the peak performance for both $native$ and ft is much higher than when solely retrieving static content or a plain PHP page without any database interaction. We observe a peak performance of 321 kreq/s for $native$, and respectively 260 and 160 kreq/s for the two ft variants. This effect can be explained by lower thread contention upon database lookups happening for each request.

In terms of CPU utilisation, as expected we observe an increase as more requests are being issued. In general, all the variants are strictly CPU-bound (the limiting factor is our hardware) and the injected workloads manage to fully saturate all CPU cores. However, the ft variants saturate the CPU much more quickly than the $native$ execution. For example, with static content we reach roughly the 800% CPU limit with a throughput of around 200 req/s. This is confirmed by a corresponding increase in response latency. This behaviour is expected since ft requires more CPU cycles (for the instrumented instructions) and thus saturates the CPU more quickly. Similarly, CPU consumption also increases for the benchmarks that retrieve dynamic content, either with and without database interaction.

Key-value stores: memcached and Redis. Next, we evaluate two widely-used key-value stores: memcached and Redis. To measure throughput vs. latency with memcached, we rely on Twitter's *mcperf* [42] tool. For Redis, we use YCSB [43] with workload A, which comprises 50% read and 50% update operations. Figure 4 shows the results for both systems.

Memcached reaches a peak throughput at 886 kreq/s for $native$ and 600 kreq/s for ft_{tx}. The ft_{ilr} variant is close to $native$, with only a 12% difference overall. Since memcached is limited by the memory bandwidth (8 GB/s on our hardware), there is only a small increase in CPU utilisation once the system is saturated.

With Redis, we observe a peak at around 120 kreq/s before the latency starts climbing. Interestingly, there is almost no visible overhead for the *ft* variants in comparison to $native$. This is because Redis is single-threaded while *ft* harnesses multi-core technology for ILR, as confirmed by the fact that CPU usage slightly increases as more requests are being processed, yet never exceeds 100% (*i.e.*, 1 core).

In summary, we observe across all applications an average throughput ($NodT$) of about 71% for ft_{tx} in comparison to $native$ execution. If we use ft_{ilr}, the throughput climbs up to 92% of $native$ performance.

C. Dependability evaluation

We next use the models described in §IV to evaluate the availability and integrity under faults for single nodes and a cluster of nodes, to calculate the required capacity.

Models settings. Table I presents the main parameters used in the models. The table consists of two parts. The first part presents the probabilities of state transitions in the integrity models from $Correct$ state to any of the other states ($Corrrupt$, $Crash$, $Retry$) when transient faults are injected in a node of a given type ($native$, ft_{ilr} and ft_{tx}). These values are produced using fault injection experiments on a wide range of applications in the HAFT paper [6] (Table 4). The second part presents the time needed to recover the different failed states back to $Correct$ state. The techniques include replacing a crashed node by either failover to a new node ($CrashRecovery$), the manual repair of a node with SDC state ($SDCRecovery$) and the recovery of a node with a detected transient fault by retrying a transaction ($RetryTx$). Note that recovery times can be converted into rates assuming the second (s) as the basic time unit, using $RecoveryRate = 1/RecoveryTime$. In the rest of this section we use the values presented in the table unless otherwise specified.

Availability of single node deployment. We want to study the availability achieved by deploying a single node without any over-provisioning techniques in the cloud compared to on-premises. The cloud differs from on-premises deployment in its ability to automatically fail over by replacing a failed node with another one from a hypothetically unlimited pool, so as to satisfy the SLA agreement. The cloud automatic failover is modelled with $Pool = \infty$ and three values for $CrashRecovery = \{15\,\text{s}, 60\,\text{s}, 1800\,\text{s}\}$, while on-premises uses $Pool = 0$. Both deployments use $Num = 1$ to model a single node in the cluster. Figure 5 shows the results of injecting hardware crash faults with rates from once to 12 per year (x-axis) and the availability achieved in one year (y-axis) for a node deployed in cloud (a) and on-premises (b). The availability calculated represents the operational availability $upTime/totalTime$, where *up* time assumes that Num nodes are operational and *total* time is one year.

Figure 5 (a) shows that the cloud deployment can achieve at least five 9's for a single node in terms of yearly availability.

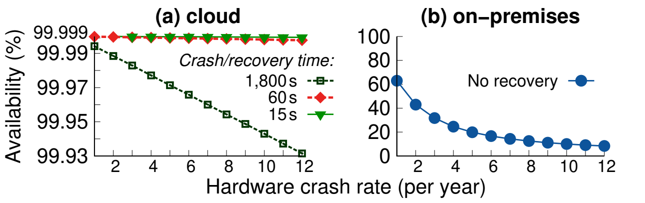

Fig. 5: Availability of a single-node for one year when deployed in the cloud (a) and on-premises (b), without over-provisioning.

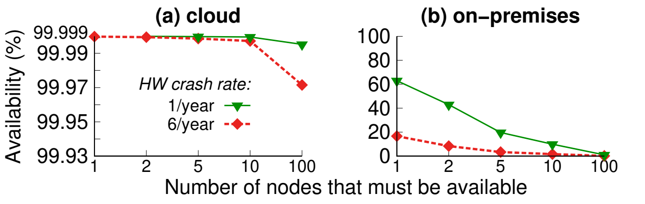

Fig. 6: Availability of a service under a given crash fault rates and without over-provisioning.

With slower recovery time (1800 s), the achieved availability of at least three 9's is consistent with what most current cloud providers would provide for a single node [45]. Figure 5 (b) shows that on-premises cannot achieve the required availability levels even with very low fault rate, and availability quickly degrades with higher fault rates. The faster degradation in availability is due to missing repairs in the on-premises deployment.

Service target throughput with failure free nodes. In the experimental phase, we measured throughput of a single node ($NodT$) for the different node types and calculated the average throughput degradation for ft_{ilr} and ft_{tx} with respect to a $native$ node. By defining the $SerT$ as multiples of $NodT_{native}$, we can calculate the number of nodes Num needed to achieve a service target throughput for all node types using $SerT/NodT$. For example, for $SerT = 1$, we need either one $native$ node or two nodes of either ft_{ilr} or ft_{tx} type. Similarly, we need either 10 $native$, 11 ft_{ilr} or 15 ft_{tx} nodes to handle $SerT = 10$. The given nodes are considered sufficient to fulfil the performance constraints of the service under the assumptions that *(i)* nodes do not fail and *(ii)* the backend infrastructure scales with the number of nodes in the cluster. If the backend infrastructure does not scale with the number of nodes, contention will increase on the backend resources, hence reducing $NodT$ and increasing the response time. In this case, more nodes are required to achieve the same $SerT$, which should then be determined experimentally.

Availability of multiple node deployments. The service requires Num of nodes to achieve the performance constraints assuming always available nodes. Nodes do, however, crash in practice, which leads to degraded service performance. We want to study the effect of nodes crashing on the overall service availability using different deployments schemes when no over-provisioning techniques are used. In Figure 6, we vary

Transient faults	*native*	ft_{ilr}	ft_{tx}
Corrupt (%)	26.19	0.8	1.17
Crash (%)	12.49	75.0	7.72
Retry (%)	–	–	66.99

Recovery time	*native*	ft_{ilr}	ft_{tx}
Crash recovery (s)	15[a]		
SDC recovery (h)	6[b]		
Retry transaction (μs)	–	–	2.5[c]

TABLE I: Probabilistic models parameters: transient fault probabilities [6] (left) and recovery times (right).

[(a)]Common values for failover in HA cluster is 1-30 s [44]. [(b)]Amazon reported 6 h to manually recover from corrupted state [22]. [(c)]Maximum latency of transaction retry with 5,000 instructions (2.0 GHz CPU).

		Extra ARA nodes						
		HCR:	**1/year**			**6/year**		
Node	**Base**	**CR:**	**15 s**	**1 min**	**30 min**	**15 s**	**1 min**	**30 min**
native	10		0	0	0	0	0	1
ft_{ilr}	11		0	0	0	0	0	1
ft_{tx}	15		0	0	0	0	0	1

(a) cloud

		Extra ARA nodes		
Node	**Base**	**HCR:**	**1/year**	**6/year**
native	10		35	113
ft_{ilr}	11		37	121
ft_{tx}	15		46	152

(b) on-premises

TABLE II: Number of active nodes needed (base + extra) for *cloud* and *on-premises* deployments with ARA to achieve 10× native throughput ($SerT$) and three 9's availability level (HCR=$\lambda_{HWcrash}$, CR=$\rho_{CrashRecovery}$).

		Extra pool nodes												
		HCR:	**1/year**						**6/year**					
Node	**Base**	**CR:**	**15 s**		**1 min**		**30 min**		**15 s**		**1 min**		**30 min**	
		PR:	**–**	**1 h**	**–**	**1 h**	**–**	**1 h**	**–**	**1 h**	**–**	**1 h**	**–**	**1 h**
native	10		18	1	18	1	19	1	30	1	30	1	×	×
ft_{ilr}	11		19	1	19	1	20	1	33	1	33	1	×	×
ft_{tx}	15		24	1	24	1	27	1	42	1	42	1	×	×

TABLE III: Number of nodes needed (base + pool) for *on-premises* deployment with PF to achieve 10× native throughput ($SerT$) and three 9's availability (HCR=$\lambda_{HWcrash}$, CR=$\rho_{CrashRecovery}$, PR=$\rho_{PoolRepair}$).

the number of nodes in the cluster (x-axis) when deployed in the cloud (a) and on-premises (b), and calculate the availability achieved at the cluster level (y-axis) when hardware crash faults are injected with two rates (1 and 6 per year). We measure the availability of cluster with $UpNodes = Num$ during one year, *i.e.*, the yearly percentage of time that the service is able to fulfill its performance requirement. We observe that, as we increase the number of nodes expected to be operational, the availability of the service deployed in the cloud remains high, while it degrades fast on-premises even with low fault rate. Therefore, deploying a service on-premises requires over-provisioning to ensure that performance requirement are met.

Capacity for dependable service. A dependable service that relies on a cluster of nodes to achieve a target QoS requires the cluster to satisfy availability constraints, in addition to protecting the integrity of the service. To that end, we can use over-provisioning (*e.g.*, PF or ARA) to meet the availability objectives as well as integrity protection techniques to enhance the integrity of the service (*e.g.*, HAFT). We use the capacity planning process in two ways: *i)* to define the number of nodes needed for a dependable service built using different node types, deployment schemes and over-provisioning techniques, and *ii)* to quantify the integrity of the service.

(1) Number of nodes needed. For *cloud deployments*, Table II (a) shows the number of nodes needed for a dependable service that has a performance constraint $SerT = 10 \times NodT_{native}$ and service availability of at least three 9's. We consider $CrashRecovery$ times for the automatic failover as 15 seconds, 1 minute and 30 minutes, and hardware crash rates of 1 and 6 per year per node. If the availability achieved does not satisfy the required level, extra nodes are provisioned as ARA nodes. The deployment types differ in their "base" number of nodes, yet they all achieve high availability levels of at least three 9's except with high crash rate and slow failover time (HCR = 6/year and CR = 30 min). In this case, it is enough to have a single extra ARA node over-provisioned in the cloud to meet the required availability level.

For *on-premises deployment*, we consider both ARA in Table II (b) and PF in Table III as over-provisioning techniques. The tables consider a service required throughout $SerT = 10 \times NodT_{native}$ and a service availability of at least three 9's. Table II (b) shows that the dependable service requires a number of active nodes $UpNodes$ equal to the sum of the "base" nodes needed for performance and the over-provisioned ARA nodes. For example, a service that requires 10 *native* nodes also needs, assuming a crash rate of 1 per year, 35 additional active nodes to ensure three 9's availability. To understand this high number of over-provisioned nodes, consider that an on-premises deployment can achieve only 10% availability with a cluster of 10 nodes (Figure 6), which is very low compared to the required level of 99.9%. Additionally, ARA nodes are active nodes and can fail due to crash failures. Therefore, we need to over-provision many nodes to ensure that at least 10 are available at any time, except for the allowed downtime (8.77 hours per year

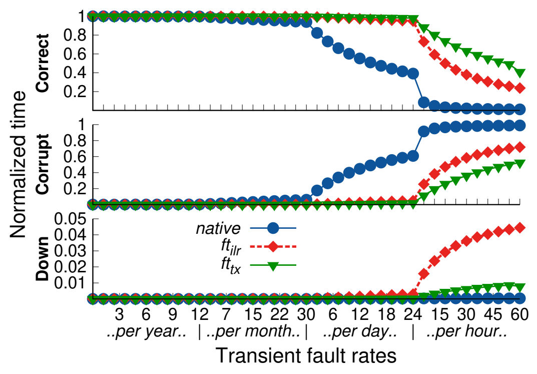

Fig. 7: Normalised time spent by the different node types in correct, corrupted and down states after injecting transient faults.

for three 9's). Note that the number of base nodes for each type is different, which also affects the number of additional ARA nodes.

Table III shows that a dependable service requires a number of nodes equal to the sum of "base" active node needed for performance and the over-provisioned nodes as PF. The table considers $CrashRecovery$ times for the failover from the pool as 15 seconds, 1 minute and 30 minutes and hardware crash rates of 1 and 6 per year per node. Additionally, we consider two cases for handling crashed nodes upon failed-over to a node from the pool: "no repair" (denoted by **–**) and "1 hour repair". If repaired, a crashed node can return to the pool and be used for further failover upon need. The table indicates that the number of nodes required to fulfil the availability constraints is dramatically reduced when repairing crashed nodes, as compared to the "no repair" case. For example, for a crash rate of 1 per year and 15 s failover time, a service would require, in order to achieve three 9's availability using 10 $native$ active nodes, 18 additional passive nodes in the pool if there are no repair vs. one if the pool is repaired at a rate of one per hour.

Comparing Table II (b) and Table III, one can see that PF requires fewer nodes than ARA. Indeed, passive nodes in the pool work in cold mode (turned off) and not exposed to crashes.

(2) Integrity of the service. To study how different nodes behave when transient faults are injected, we use the integrity models presented in Figure 2. If not masked, a transient fault can crash the node or corrupt its internal state with probabilities that vary between different node types due to their ability to tolerate transient faults, as seen in Table I. Note that $native$ nodes do not implement any integrity protection mechanism, while ft nodes are hardened against data corruption. Transition rates between $Correct$ state and other states ($Corrupt$, $Crash$, $Retry$) are defined by the injected transient faults rate multiplied by the corresponding probability.

Figure 7 presents the normalised time that each node type spends in $Correct$ (available), $Corrupt$ (available but integrity is not preserved) and $Down$ (unavailable, crashed or under repair) states in one month in the cloud. The figure shows that with low transient fault rates, all nodes spend most of their time in $Correct$ state. By increasing fault rates, nodes spends more time in $Corrupt$ or $Crash$ states. Specifically, $native$ nodes spend 0.2–5.5% of time in $Corrupt$ state with faults in the month range and 6–58% with faults in the day range, whereas these percentages decrease to 0.007–0.18% and 0.2–4.3% with ft_{ilr} and 0.0026–0.07% and 0.07–1.67% with ft_{tx} for the same ranges, respectively. With high fault rates, ft_{tx} spends more time than ft_{ilr} in $Correct$ state and less in $Corrupt$ or $Down$ states. This is due to the use of transactions to recover from a detected fault in ft_{tx}, as compared to the slower failover recovery in ft_{ilr}. Therefore, ft_{tx} is not only more reliable, but also more available than the other node types. Service dependability should not only consider the availability of the nodes in the cluster, but also the integrity of the available nodes, since they may spend a considerable amount of time in $Corrupt$ state if integrity mechanisms are not implemented.

VI. Conclusion

We have developed a capacity planning process (PCRAFT) to quantify the number of nodes needed to ensure the dependability of stateless services. We consider availability, integrity and performance failures for cloud-based and on-premises deployments. PCRAFT combines a two-phase process—empirical and modelling-based. In the *empirical phase*, we characterise the performance and integrity of the service at the node level to parameterise the *modelling phase*, in which we implement probabilistic models to estimate the availability and integrity of service. Our evaluation of PCRAFT using Apache, memcached and Redis shows that both availability and performance are important to leverage the benefits of dependability mechanisms.

References

[1] A. Rodriguez, "Restful web services: The basics," *IBM Developer*, 2008.

[2] K. Birman, D. A. Freedman, Q. Huang, and P. Dowell, "Overcoming CAP with consistent soft-state replication," *Computer*, vol. 45, no. 02, 2012.

[3] W. Greene and B. Lancaster, "Carrier-grade: Five nines, the myth and the reality," *Pipeline magazine*, vol. 3, no. 11, 2007.

[4] J. Schad, J. Dittrich, and J. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, 2010.

[5] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *CGO*, 2005.

[6] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "HAFT: Hardware-assisted fault tolerance," in *EuroSys*, 2016.

[7] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *SC*, 2013.

[8] S. Mohan, F. M. Alam, J. W. Fowler, M. Gopalakrishnan, and A. Printezis, "Capacity planning and allocation for web-based applications," *Decision Sciences*, vol. 45, no. 3, 2014.

[9] R. P. Mohamad, D. S. Kolovos, and R. F. Paige, "Resource requirement analysis for web applications running in a virtualised environment," in *CloudCom*, 2014.

[10] D. Z. Tootaghaj, F. Farhat, M. Arjomand, P. Faraboschi, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Evaluating the combined impact of node architecture and cloud workload: Characteristics on network traffic and performance cost," in *IISWC*, 2015.

[11] N. Roy, A. Dubey, A. Gokhale, and L. Dowdy, "A capacity planning process for performance assurance of component-based distributed systems," in *ICPE*, 2011.

[12] A. Undheim, A. Chilwan, and P. Heegaard, "Differentiated availability in cloud computing slas," in *Grid*, 2011.

[13] D. S. Kim, F. Machida, and K. S. Trivedi, "Availability modeling and analysis of a virtualized system," in *PRDC*, 2009.

[14] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, "End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach," in *PRDC*, 2010.

[15] H. Qian, D. Medhi, and K. Trivedi, "A hierarchical model to evaluate quality of experience of online services hosted by cloud computing," in *IM*, 2011.

[16] K. Nagaraja, X. Li, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Using fault injection and modeling to evaluate the performability of cluster-based services," in *USTIS*, 2003.

[17] K. Nagaraja, G. Gama, R. Bianchini, R. P. Martin, W. Meira, and T. D. Nguyen, "Quantifying the performability of cluster-based services," *IEEE TPDS*, vol. 16, no. 5, 2005.

[18] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.

[19] HAProxy, https://www.haproxy.org.

[20] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *PODC*, 2007.

[21] "New defective S3 load balancer corrupts relayed messages," https://forums.aws.amazon.com/thread.jspa?threadID=22709, 2008.

[22] "Amazon S3 availability event," https://status.aws.amazon.com/s3-20080720.html, 2008.

[23] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *EuroSys*, 2011.

[24] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed, "Reliable data-center scale computations," in *LADIS*, 2012.

[25] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI*, 1999.

[26] Y. J. Song, F. Junqueira, and B. Reed, "BFT for the skeptics," in *SOSP: Work in Progress Session*, 2009.

[27] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and others., "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *DSN*, 2007.

[28] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: Decoupled acyclic fault tolerance," in *PACT*, 2010.

[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[30] W. Iqbal, M. N. Dailey, and D. Carrera, "Black-box approach to capacity identification for multi-tier applications hosted on virtualized platforms," in *CSC*, 2011.

[31] I. Giannakopoulos, D. Tsoumakos, N. Papailiou, and N. Koziris, "PANIC: Modeling application performance over virtualized resources," in *IC2E*, 2015.

[32] C. Melo, R. Matos, J. Dantas, and P. Maciel, "Capacity-oriented availability model for resources estimation on private cloud infrastructure," in *PRDC*, 2017.

[33] D. S. Kim, J. B. Hong, T. A. Nguyen, F. Machida, J. S. Park, and K. S. Trivedi, "Availability modeling and analysis of a virtualized system using stochastic reward nets," in *CIT*, 2016.

[34] B. Silva, P. Maciel, E. Tavares, and A. Zimmermann, "Dependability models for designing disaster tolerant cloud computing systems," in *DSN*, 2013.

[35] R. M. De Melo, M. C. Bezerra, J. Dantas, R. Matos, I. J. De Melo Filho, and P. Maciel, "Redundant VoD streaming service in a private cloud: Availability modeling and sensitivity analysis," *Mathematical Problems in Engineering*, vol. 2014, 2014.

[36] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic model checking for performance and reliability analysis," *SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, 2009.

[37] Z. Xue, X. Dong, S. Ma, and W. Dong, "A survey on failure prediction of large-scale server clusters," in *SNPD*, 2007.

[38] "Apache HTTPD," https://httpd.apache.org.

[39] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, Aug. 2004.

[40] "Redis," https://redis.io.

[41] "wrk2," https://github.com/giltene/wrk2, 2015.

[42] "mcperf," https://github.com/twitter/twemperf.

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[44] "High availability failover optimization, tuning HA timers, PAN-OS 6.0.0," https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000ClOSCA0, 2013.

[45] "Amazon S3 service level agreement," https://aws.amazon.com/s3/sla/, 2019.