Integrated approach to Dynamic and Distributed Cloud Data Center Management

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Tiago Filipe Rodrigues de Carvalho

B.S., Electrical and Computer Engineering, Instituto Superior Técnico, Portugal

M.S., Information Technology - Information Security, Carnegie Mellon University, USA

M.S., Information Security, Faculdade de Ciências da Universidade de Lisboa, Portugal

Carnegie Mellon University

Pittsburgh, PA

December, 2016

Acknowledgements

This thesis is a result of a long effort with multiple personal and professional hurdles. I would not have finished it successfully without the support of several people. First and foremost, I would like to thank two people that were along for the entire ride.

To my wonderful, caring and infinitely patient and strong wife, Juliana. The last seven years were an adventure and, believe me, I loved every moment of it. No matter if near or far, through facetime, phone or through our photo of the day, I have always felt your unyielding support. Thank you, Juli!

To my advisor, Prof. Hyong S. Kim for continuously pushing me, keeping me on track and for always be readily available for a research discussion or a simple pep talk. I would definitely not have reached the finish line without your guidance.

To the members of my doctoral committee, Prof. Priya Narasimhan, Prof. Anthony Rowe and Prof. Ricardo Morla for the invaluable feedback. I was truly honored to have you in my committee.

To all the members of our research group throughout my years at CMU. Thank you, Chen, Liang, Sihyung, Andy, John, Kob, Akk, Non, Namtarn, Andrew, Kiki, Justin, Ke and Senbo for all the discussions, talks and help. You made the group such a pleasant environment for research.

To Prof. Nuno Neves for all the feedback and for making everything within your reach to help me take this thesis to the finish.

To Eng. José Alegria for the confidence and for making it possible for me to enroll in the PhD program in the first place.

To everyone making the Carnegie Mellon Portugal program a success. Support for this research provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program.

A very special thank you to my family for all the support over the years. To my parents, Vitor and Helena, for all the love and education that made me the person I am today. To my siblings André and Susana for not asking when would I finish it and to Rodrigo for the reverse psychology. To my nieces, Madalena and Margarida, for always putting a smile on my face even in the most downbeat moments.

Finally, a word to my grandparents Albertina, Helena and Carlos for always being happy for me and showing such pride even if this work kept me at a distance for the final years of their lives.

Abstract

Management solutions for current and future Infrastructure-as-a-Service (IaaS) Data Centers (DCs) face complex challenges. First, DCs are now very large infrastructures holding hundreds of thousands if not millions of servers and applications. Second, DCs are highly heterogeneous. DC infrastructures consist of servers and network devices with different capabilities from various vendors and different generations. Cloud applications are owned by different tenants and have different characteristics and requirements. Third, most DC elements are highly dynamic. Applications can change over time. During their lifetime, their logical architectures evolve and change according to workload and resource requirements. Failures and bursty resource demand can lead to unstable states affecting a large number of services. Global and centralized approaches limit scalability and are not suitable for large dynamic DC environments with multiple tenants with different application requirements.

We propose a novel fully distributed and dynamic management paradigm for highly diverse and volatile DC environments. We develop LAMA, a novel framework for managing large scale cloud infrastructures based on a multi-agent system (MAS). Provider agents collaborate to advertise and manage available resources, while app agents provide integrated and customized application management. Distributing management tasks allows LAMA to scale naturally. Integrated approach improves its efficiency. The proximity to the application and knowledge of the DC environment allow agents to quickly react to changes in performance and to pre-plan for potential failures. We implement and deploy LAMA in a testbed server cluster. We demonstrate how LAMA improves scalability of management tasks such as provisioning and monitoring. We evaluate LAMA in light of state-of-the-art open source frameworks. LAMA enables customized dynamic management strategies to multi-tier applications. These strategies can be configured to respond to failures and workload changes within the limits of the desired SLA for each application.

Table of contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Data Centers and Cloud Computing: A Brief History

The first Data Centers (DCs) were originally used to house the first large and complex computers. These facilities started to change with the advent of microcomputers during the 1980s and the widespread use of the Internet during the 1990s. The need to make data available to thousands of clients led companies to create their own DCs. The evolution of microcomputers to small powerful servers led to the creation of infrastructures with large numbers of server racks.

IBM introduced the notion of virtualization to share resources among different applications within a mainframe. The software had to be developed specifically to run on those large machines. To achieve resource flexibility, most enterprise applications evolved into distributed architectures. Extra machines could be added as soon as the application required more resources. The distributed architecture allowed scaling of application requirements. Even so, each application would require its own infrastructure, leading to underutilization and cumbersome management. This compartmentalization was often extended to the network infrastructure. The risk of an application's traffic affecting other applications and the lack of flexibility led to dedicated physical network segments for each application. Data processing environment paradigms like Grid Computing were used to share resources of a large number of machines. However, the Grid paradigm would only support a specific subset of applications.

The next big step in DC evolution came with the widespread use of virtualization. Years after the first application-level virtualization concept of IBM mainframes, virtualization was used to emulate full hardware environments. Users of one operating system are able to run software in other platforms. The application of virtualization to DCs brought other advantages. Applications could be easily deployed and moved between different servers leading to faster application provisioning. Applications sharing the same server could be logically isolated from each other. For instance, upgrading one virtual machine (VM) operating system would not affect the others. The choice of operating systems would also be independent of the physical machine.

Cloud computing is the realization of the idea of computing as a utility [1], i.e. users can rent resources to run their own software and pay only the used amount. Vaquero et al [2] summarized a definition of a cloud as a “large pool of virtualized resources … that can be dynamically reconfigured to adjust to a variable load”. These resources would be offered as a service and users would only pay for used resources. The cloud model can be applied to: the infrastructure (Infrastructures-as-a-Service or IaaS), where the hardware is rented to the user using VMs; software platforms (Platform-as-a-service or PaaS), where the user can develop her applications using a development platform defined by the provider; and software (SaaS), where the user acquires a license to use an application that is deployed in the provider’s DC. The Cloud computing paradigm allows high consolidation of servers, optimized resource consumption, fast provisioning of new applications, independence from hardware manufacturers, and application isolation. The idea of isolation, together with cost savings, is the main factor that drives companies to migrate their applications to the cloud. In perfect isolation, the applications can maintain the same performance and security guarantees provided by dedicated and private infrastructure. More recently the concept of virtualization has been extended to networks. New technologies allow DC providers to deploy virtual networks [3], which can isolate the traffic between different applications and facilitate sharing of resources in the network.

However, DC management solutions for Cloud Computing are still in their infancy. Integrating all the new technologies and, at the same time, being able to manage the interests of different applications is highly challenging.

1.2. Diversity and Complexity

Large companies operating cloud DCs host over a million servers. A single DC can host hundreds of thousands of servers and this number is expected to grow. Virtualization allows several different applications to be deployed on the same server. The number of VMs per server can vary widely with the hardware capabilities and application requirements. Additionally, the communication capacity among servers depends on their location in the network. These characteristics make DCs a complex environment to manage and maintain.

1.2.1. Physical Infrastructure Diversity

The physical infrastructure can be very diverse with respect to servers, network devices and middle-boxes (i.e. load-balancers, firewalls, IDS, etc.).

- **Different generations of hardware**: As DCs evolve, there might be several stages of hardware acquisition. More servers are added as the demand on DCs increases. This leads to several generations of servers with different capabilities in the DC.
- **Replacement due to failures**: The occurrence of failures imposes changes on server and network characteristics. As hardware evolves quickly, several components may be upgraded, leading to machines with different characteristics even in the same cluster.
- **Diverse manufacturers**: DC managers usually equip their infrastructure with the same manufacturer for management simplicity. However, buying from different manufacturers has some advantages. It reduces risk by lowering the dependency on one manufacturer and reduces acquisition costs by increasing competition among manufacturers. It also lowers dependency on proprietary technologies and increases options to align infrastructure capability with functional requirements [4]. On the other hand, different manufacturers can also lead to various proprietary technologies in the DC environment.
- **Different application requirements**: The DC hosts applications with very different requirements. Some applications require fast processing, I/O and network connections while others require parallel processing on several cores. To be able to efficiently support a wide range of applications, DC managers may acquire machines with different capabilities.
- **Legacy technologies and applications**: Legacy applications often hold back hardware upgrades. For instance, large applications designed for mainframes or older operating systems may be too costly to upgrade. Some technologies can become obsolete. But DC managers continue to use them while other parts of the DC are upgraded. It would be too costly to completely upgrade the infrastructure at once.

There are heterogeneous server and network capabilities within the same DC [5]. Servers can be heterogeneous in terms of memory, number of CPU cores, CPU speed, disk space and I/O speed. Network switches and routers can have different bandwidth, latency and number of ports. The architecture of the network leads to large differences in latency between pairs of hosts. Clusters of servers can also vary in size and interconnection capacity.

1.2.2. Application Diversity

DCs host a multitude of heterogeneous applications. While this diversity can occur in any DC, it is more pervasive in public multi-tenant clouds with the Infrastructure-as-a-Service paradigm. Users deploy their applications in a common infrastructure. The DC provider controls only the virtual machines and their placement in the network. User applications can have varied architectures and QoS requirements. There are several application characteristics that can influence how and where applications are deployed:

- **Application architecture**: From a logical point of view, an application consists of one or more components that we will refer to as *services*. Each service can have several instances where each instance runs on one VM. The application architecture defines the number of different services, the computing requirements for each service and the communication requirement among services. The resources with the most impact on application performance can vary among application services.
- **Dependencies on DC services**: User applications often require generic services provided by DC. For instance, firewall, intrusion detection systems or load balancing can be provided transparently to the application. Devices providing these services are often called middle-boxes. The locations of the application’s VMs depend on the location of the servers providing these services. These DC services can also be mobile and its placement varies according to the locations of the VMs.
- **Performance Requirements**: Applications have different performance requirements. Highly interactive services need very low latency (e.g. web shopping websites), while other static websites tolerate a longer response time. Other applications may require high availability of a specific subset of services. Therefore, each service has different semantic and quantitative requirements. These requirements are defined as Service Level Agreements (SLA). The use of SLAs is often limited to availability guarantees. Availability is the only metric current DCs guarantee in IaaS deployments.

This diversity makes resource allocation using generic centralized mechanisms extremely difficult. The dependencies among services play an important role on the performance of the application. Thus, the resource allocation for one service depends on the rest of the application.

1.2.3. Context Diversity: Synergies and Conflicts between Applications and Infrastructure

Applications and infrastructure characteristics can be highly diverse within the DC. In a virtualized environment, applications share physical resources. We need to consider both the effects of the interaction among co-located applications and the suitability of an application in the underlying infrastructure. We need to consider the following aspects:

- **Co-located applications**: Despite the logical isolation, the physical resources of a server are shared among several applications. To increase resource usage, DC managers could over-provision servers. Typical resource allocation algorithms only verify if a server has enough resources available to hold a logical VM. However, the characteristics of both the application as well as the resource usage on the host should be considered. For instance, two very-low latency applications may not be deployed together in a single-CPU server, as both would need to access resources with high priority.
- **Host-aware allocation**: It has been shown that the power consumed by workloads can depend on the host characteristics [5]. Considering host characteristics in resource allocation of applications helps lowering power consumption. Other often-neglected characteristics of the hosts can significantly impact application performance. CPU cache characteristics may impact some applications [6]. Different types of applications require different tuning of resources. For example, Facebook uses specific tuning of their servers to handle a large number of UDP connections [7]. These optimizations may be helpful for some applications and harmful for others and should be taken into account when allocating applications.
- **Topology-aware allocation**: Distributed applications are highly dependent on network latency and bandwidth. Allocations should consider the topology and status of the network. Some applications must run inside a cluster, while others may run on servers spread across the network for resilience.
- **Security**: In a public Cloud, different users use the same resources. Virtualization provides logical isolation among applications owned by different users. However, isolation can sometimes be compromised by software vulnerabilities or even human error. To ensure security required by some applications, users may impose constraints on which applications can share physical hosts. A user with many applications may be willing to spend more to have resources shared only among his applications. Another important issue is legacy applications requiring old software. The DC manager may want to isolate these applications from others.

Applications can be managed more efficiently if the environment characteristics are known. For example, an application might tolerate some degree of interference while still meeting their SLA requirements.

1.3. Dynamic Behavior

Challenges for DC management also arise from the dynamic behavior of DC applications and infrastructure.

1.3.1. Allocation Patterns

An import characteristic of applications is how often new VMs are requested. Some applications, like multi-tier applications can run on the same VMs for their entire lifetime. New VMs are only launched due to workload variations or failures (see Section 1.3.3) and not due to the intrinsic nature of the application. However, applications that do batch processing might launch new VMs to run only a time-limited task. These applications are constantly reserving and releasing resources. Thus, they are more flexible and require more agile allocation mechanisms. On the other hand, long running applications are less demanding in terms of allocation processing but require more stable environments to run.

1.3.2. Application Architecture Evolution

Another aspect to consider is the evolution of an application over time. Users may launch new features, refactor the application or make use of new technologies. Thus, during its lifetime, an application can suffer changes on its architecture, on its resource requirements or on in their dependencies on the DC services. Cloud management mechanisms need to be able to adapt to these changes dynamically. The changes in the configuration and allocation of one application can affect other co-located applications. Thus, all applications should be aware of the environment they are running on and actuate when changes can affect their performance or resilience. This requires granular monitoring of the environment.

1.3.3. Workload Variation: Automatic Scaling

Even if the application architecture does not change, an application workload can change dynamically:

- Workload can change during the day. The effect of these changes depends on the provisioning strategy. We can have low resource utilization periods if provisioning is done with respect to expected peak usage. This inefficient host utilization can be improved by resorting to over-provisioning. However, there is the risk of congestion and consequently degradation on application performance;
- Workload evolves as the popularity of the application evolves and new resources need to be assigned;
- Sudden changes due to specific events. These changes can be expected or not, depending on if the event was planned (e.g. marketing campaign) or not (e.g. new popular item on shopping website);

To handle dynamic workloads, DC managers resort to automatic scaling mechanisms. However, once again, the proper mechanisms to use depend on application characteristics and on the context surrounding the event. Automating scaling can be either horizontal, where new VMs are provisioned to handle new traffic, or vertical, when the resources used by a VM are increased. Horizontal scaling assumes the application is developed to handle multiple simultaneous instances. On the other hand, vertical scaling requires the host to have enough free capacity to handle the new requirements. Eventually, as an application workload changes, either the affected applications or some other co-located applications need to migrate so that enough requirements are available.

1.3.4. Failures and other Environment Changes

The physical DC environment can also change: new servers can be added, the network architecture could change or failures may occur leading to sudden changes in availability of the physical infrastructure. Ideally, managers would like the DC environment to automatically adapt when conditions change. For instance, applications looking to improve performance should automatically make use of the new resources without requiring manual requests or configuration.

Another important aspect is application resilience. An application should maintain its performance as failures occur, or at least present a controlled progressive degradation. To improve resilience, the allocation process needs to take into account the application architecture, the physical environment and the performance requirements. There are two main methods to deal with failures:

- **Static (fault-masking)**: multiple instances are deployed to ensure, that if a failure occurs, the service continues to operate with sufficient performance. This approach is costly but might avoid downtime;
- **Dynamic (fault-recovery)**: fail-over instances are prepared to take over when a failure occurs. Although less costly, this solution assumes a period of unavailability during recovery.

The most adequate scheme to use depends on the application architecture and capabilities. It can also be determined by cost, as multiple active instances are normally more expensive as they imply higher resource reservation. Thus, the final solution is trade-off between cost and application requirements (in terms of SLAs defining response time, MTTR, MTBF, etc.).

1.4. Cloud Management

Hundreds of thousands of dynamic applications can co-exist and share resources within these infrastructures. Cloud management systems need to handle frequent allocation requests and requirement changes.

1.4.1. DC Entities and their Goals

The DC ecosystem consists of several entities with conflicting interests and different timeframes:

- **Physical Infrastructure (Resource Providers)**: Servers, switches, routers, power supplies, coolers, etc. These are resources to be allocated to applications and services. The main goal of resource allocation is efficiency. DCs should accommodate as many applications as possible, without compromising performance.

- **User applications (Resource Consumers)**: Applications owned by users. Each application has its own specific performance objectives and resource requirements.
- **DC services (Resource Consumers and Providers)**: Generic services that can be used by multiple applications (e.g. firewalls, IDS, load balancers, etc.). These services can be both service consumers and/or providers. These services can be virtualized and deployed just like user's services.

1.4.2. Management Systems

Cloud Management systems encompass several operations to be able to run applications with acceptable performance. The first basic function is *resource allocation*. The system searches for appropriate resources and deploy the application. In current DCs, specific services like data or image storage are physically separated and often allocated separately. The allocation process may also include network services to define the addressing scheme and/or virtual private networks. A second important feature is application *monitoring*. The management system must be aware of the DC environment. This includes application location, characteristics and performance. Finally, a management system should provide *customized application management*. The system should allow applications to adapt to changes not only on their workload and characteristics but also on the underlying infrastructure (e.g. failures, maintenance or reconfiguration). Adverse conditions should be detected and appropriate VM management operations (e.g. live migration, replication, scaling, etc.) have to be deployed.

There are several management systems for large Cloud DCs [8], [9]. OpenStack is supported by several large players in the DC market and reflects many of the current design concerns of DC managers. OpenStack splits management into three main components: compute, networking, and storage. The *compute* module allocates VMs to physical servers. A centralized component determines the location of a VM for each allocation request. Compute nodes on each physical server are responsible for periodically announcing their resource availability and deploy local VMs. The *network* component defines the networking model for each application. It manages, for instance, configuration of VLANs, virtual private networks and mappings to public addresses. The *storage* node manages VMs in storage volumes. Finally, a graphic *interface* allows a manager to control the DC from a central point and users to make allocation requests.

The current default OpenStack's VM allocation mechanism has two main steps: First, it uses constraints in the VM allocation request to filter hosts retrieved from the central database. The filtered list is then sorted according to customized cost functions. The cost functions depend on the user request or on host characteristics. Furthermore, the system combines multiple cost functions. Once the host is selected, OpenStack contacts the selected compute node to deploy the VM.

The generic approach taken by OpenStack can be highly customized but it is a centralized mechanism that sequentially processes VM requests. There are several optimization algorithms to determine resource allocation. These algorithms aim to maximize the number of deployed applications with respect to one or two parameters (for instance, CPU, bandwidth, network, configuration cost or failure resilience) [10]–[16]. However, centralized solutions do not scale for large and highly diverse DCs. Optimal solutions may require a large number of applications to be reallocated. Given the dynamics of DCs, the state of the environment can change faster than the time it takes to migrate all applications to the new allocation. Additionally, some applications must be running continuously and may not be able to migrate immediately or may incur a high cost [17], [18]. Optimal placement solutions may become obsolete shortly after being found. Some solutions assume that the traffic matrix is fully known [19], which is infeasible in highly heterogeneous environments. Others assume that the DC has identical VMs [11]. This optimization approach is more suitable for DCs that operate large applications with a large number of identical VMs performing similar tasks like web search or social networking. In an ever-growing multi-tenant DC scenario, it is not possible to apply these solutions due to diversity of the applications and infrastructures.

1.5. An Integrated Approach to Distributed and Dynamic Management for a Cloud Data Center

We explore a generic approach for automatic management of highly complex and dynamic applications within a multi-tenant cloud DC environment. Given the diversity and dynamic changes in DCs, we propose a *distributed* and *dynamic* approach for DC management. Allocation decisions should be distributed throughout the DC. Autonomous agents running on each DC server analyze and allocate application requests. Instead of a single centralized entity with complex tasks for a very large number of application requests, we propose thousands of autonomous agents, each assigned to process a few application requests. A fully distributed system can be more effective and easily scale to a very large, complex and dynamic cloud environment. A distributed approach effectively reacts to congestion or failures. Each agent that allocates resources gathers the necessary information to be able to analyze an application request. We apply resource discovery protocols so that each agent can gather information about available resources in other servers.

Our approach *integrates* application management operations into the cloud. Autonomous local agents adapt to the time-variant nature of most DC applications and infrastructure. Agents customized to each application can monitor the application locally, and quickly react to changes. The dynamic nature of our proposed system constantly renews information about resource availability and quickly adapt to sudden events that affect application performance. Distributed application agents increase the capacity to monitor applications, maintain state information and make more granular decisions. Agents run inside the cloud, in close proximity of the entities they manage. They can access information with finer granularity and react quickly to environment changes. Our proposal provides the *management elasticity* to the advertised *resource elasticity* of the cloud.

1.5.1. Management as a Service

The decision of moving applications to the cloud is mostly supported by a he pay-per use model. Arguably the user would only pay for the resources it uses. However, in order to achieve the desired performance levels, the user has to overestimate the resources it needs. Current DC providers offer automatic horizontal scaling based on pre-defined thresholds. The responsibility for the availability of a service is left to the user. For instance, Microsoft Azure requires users to deploy instances of a service on VMs in at least two fault domains, to achieve a 99.95% availability [20]. The user is left to design the fault tolerance mechanisms and set the conditions for automatic scaling. We believe this approach contradicts the goal of the cloud since the user needs to manage intrinsic failure events of the cloud.

1.6. Contributions

Management solutions for current and future Infrastructure-as-a-Service (IaaS) Data Centers (DCs) face complex challenges. First, DCs are now very large infrastructures holding hundreds of thousands if not millions of servers and applications. Second, DCs are highly heterogeneous. DC infrastructures consist of servers and network devices with different capabilities from various vendors and different generations. Cloud applications are owned by different tenants and have different characteristics and requirements. Third, most DC elements are highly dynamic. Applications can change over time. During their lifetime, their logical architectures evolve and change according to workload and resource requirements. Failures and bursty resource demand can lead to unstable states affecting a large number of services. Global and centralized approaches limit scalability and are not suitable for large dynamic DC environments with multiple tenants with different application requirements.

We believe that highly distributed systems fare better in this ever-growing and highly dynamic scenario. Among distributed paradigms, multi-agent systems provide the flexibility and autonomy required to manage individual applications.

This thesis proposes LAMA, a multi-agent cloud management system that:

- **distributes** management tasks over all hosts to increase scalability with the number of hosts and operations;
- **dynamically** adapts to changes in the environment or application needs;
- **integrates** application management into the infrastructure for faster and more efficient reaction to events.

The main contribution of this thesis is the study and development of a distributed, integrated and dynamic management system for highly diverse and volatile DC environments. Our approach allows management systems to *scale* with the DC size and *automatically and dynamically adapt* to changes on its characteristics.

Contributions:

1. Distributed Management and Control for Scalability

Current cloud management frameworks follow a centralized paradigm. Management is split into multiple centralized services. Each service performs a required management task for all hosts (e.g. scheduler, monitoring, orchestration, image management, networking, security). LAMA differs from current cloud management system as it does not split management workload per task. Instead, LAMA splits management workload into autonomous agents that can handle all required management tasks for only few requests.

LAMA distributes workload for all management tasks: scheduling, provisioning, monitoring, orchestration, image management and networking. LAMA eliminates typical bottlenecks seen in centralized systems. Provisioning requests can be processed by any host in the DC. Monitoring data is distributed among the DC network. Network management is handled by SDN controllers in different hosts. LAMA scales naturally as the size of DC increases. Whenever a new host is deployed it includes a new management agent. The management capacity increases as the DC capacity grows.

We demonstrate the advantages of our distributed approach through the following results:

   a. We demonstrate, using a network simulator, that LAMA schedules enough instances in parallel to fill 95% of a cloud DC with capacity for 70000 instances in only 120 milliseconds.

b. We demonstrate, using implementation in a production data cluster, that LAMA scales better than OpenStack when provisioning a high number of concurrent requests. LAMA maintains constant time to activate an instance as the number of concurrent requests is increased. OpenStack's time to activate each virtual instance increases linearly with the number of concurrent requests. In our cluster, with capacity for 82 instances without over-allocation, OpenStack takes 100 seconds to provision all the instances while LAMA completes the task in 8 seconds.
c. We show that LAMA eliminates centralized bottlenecks for monitoring services in the DC. Workload in centralized systems increases linearly with the number of hosts and instances managed. In LAMA, the monitoring workload on each link depends only on the instances deployed in a single instance.

2. Dynamic Management for Better Adaptation to Events and Past Behavior

Current cloud management frameworks provide auto-scaling mechanisms based on static rule-based mechanisms. LAMA introduces autonomous management agents that can adapt application deployment periodically or when a failure occurs. LAMA introduces the concept of management strategies. Management strategies are used to determine application deployment (e.g. number of instances per service, constraints on location of instances, etc.) while taking into account current performance of the application, past history of downtime and service-level goals. Strategies not only determine the number of instances needed to process current workloads, but also pre-plan for failures using backup instances. Latent instances (warm, hot, cold) are in different phases of the deployment process. The phase determines how fast those instances become active. Additionally, coordination messages between agents enable detection of conflicts of allocation of latent instances. This enables agents to continuously update deployment to be ready to respond to the next failure.

LAMA's management strategies allow continuous adaptation to different events and application states. This allows us to achieve different service levels for different types of applications across different periods of time.

In order to illustrate the advantages of our dynamic approach:

a. We analyze the different recovery times we can achieve by using different types of latent instances in LAMA. Recovery time can range from several minutes to no downtime. This enables LAMA agents to adjust recovery times according to each application service level requirements.

   b. We demonstrate that LAMA satisfies the availability required by applications with different SLAs in the presence of multiple sequential failures. The dynamic nature of app agents allows refreshing the application deployment after changes caused by failures. We deploy multiple applications with diverse availability requirements ranging from zero to a few minutes of downtime. We show that recovery time remains below the recovery value determined by a user-set SLA after several sequential failures. This includes zero downtime for applications that need to be continuously available.
   c. We demonstrate, in detail, how an application agent adapts the deployment of an application as failures occur. We show that LAMA can avoid downtime for applications that are close to violating their SLA. We deploy a multi-tier application with service-level parameters defined within a sliding window. We demonstrate that the application keeps the minimal number of deployed instances required to achieve the desired performance and be able to recover within the available recovery time. If a failure causes downtime, the SLA is affected. LAMA immediately adapts the application deployment to the new SLA by adding new recovery or active instances. If the time available to recover as defined the SLA becomes very low (below the time to recover from a hot instance), the app agent deploys extra active instances. The application downtime is thus bounded by the SLA defined for a sliding-window.
3. Integrated Management to enable Environment-Aware Application Management

Current cloud frameworks allow cloud users to manage their applications through a centralized API. These management systems provide metrics about the state of virtual instances. However, performance of the cloud applications does not depend only on the state of the instances. It depends on the entire application ecosystem, i.e. application, host, network and co-located instance characteristics.

LAMA allows application management agents to access information about the whole application's ecosystem. LAMA also allows cloud users to provide algorithms tailored to their applications' needs and characteristics. This enables deployment of monitoring algorithms that can detect failures that depend on the host behavior and application characteristics. Traditionally systems, lacking customization, would miss it or generate false positives.

We illustrate the advantages through the following experiments:

   a. We present a case study of failures that can affect the application even with low workloads. We demonstrate how to design a customized algorithm to be deployed in LAMA. It uses information not made available on other platforms. We demonstrate how this algorithm determines the right action to perform and eliminates false positives from basic rule based techniques. Using LAMA, we build a monitoring strategy to detect disk interference at the host level for a multi-tier online application. A decision tree is built to diagnose and react to two types of interferences that cause significant impact on the application's response time: (1) Disk write interference on the host of a database service that causes significant increase on the application response time when the CPU IO Wait metric of the host surpasses 25%; (2) Disk read interference on the host of the web server that causes significant increase on the application response time when the CPU User and System metrics surpasses 90%. Resource usage analysis on the virtual instances only leads to false negatives, while plain resource usage thresholding for the hosts leads to false positives.

b. We propose a management strategy that operates according to application's characteristics and service-level parameters defined by the user. This strategy uses internal information about the time it takes to deploy and configure new instances. We show how users can have different recovery times per failure simply by adjusting SLA parameters for their applications. The availability requirements range from allowing several minutes of downtime to avoiding any downtime. We apply this strategy to a multi-tier online application. We show that the app agent maintains recovery time below the SLA. We demonstrate that the app agent takes differentiated decisions per service. As provisioning times vary with the service's image size and instance configuration, the app agent deploys different recovery instances per service accordingly.

1.7. Dissertation Organization

In Chapter 2 we present a detailed description of the LAMA framework. Chapter 3 describes the algorithms and protocols used for resource allocation. In Chapter 4, we analyze LAMA instance provisioning workflow and compare its performance with OpenStack. Chapter 5 introduces LAMA's distributed monitoring framework and discusses its advantages. Chapter 6 describes a dynamic management strategy for multi-tier web applications. It demonstrates how cloud users can deploy customized management strategies for their applications. In Chapter 7, we present LAMA's web interface that allows visualization of the state of the framework and applications. Chapter 8 summarizes and presents our conclusions.

Chapter 2

The LAMA Framework

In IaaS infrastructures, a large number of third-party apps share a large pool of resources. Increasing size of IaaS DCs and number of hosted applications lead to a highly dynamic computing environment. Cloud management frameworks need to handle frequent provisioning requests, to manage failures and recovery for servers and applications, and to adapt to variable and unpredictable workloads. The fact that cloud providers know little about applications and the virtual instances they host, only makes management more challenging.

LAMA is a multi-agent system (MAS) that manages IaaS cloud data centers. Efficient management of these infrastructures involves a high number of tasks (e.g. monitoring, provisioning, failure detection and recovery, network management, application scaling, and access control). LAMA distributes management tasks among multiple autonomous entities spread over the DC. LAMA's agents are dynamic entities continuously adapting to changes in their environment.

2.1. LAMA Design Vectors

We built LAMA based on three main design concepts: distributed processing, dynamic adaptation and integrated management of applications.

2.1.1. Distributed Processing

Current DC cloud infrastructures can host hundreds of thousands of servers. A single company can have more than a million servers across its multiple DCs. A single centralized management system is not able to scale indefinitely. We apply a distributed management approach with two main goals in mind:

- **Scalability**: A DC management system should be independent of the DC size. In multi-tenant DCs, users create a high number of small applications. Thus, the management complexity should be driven by the application size and complexity. To allow DC growth, management operations should be independent on the number of servers and deployed applications. A distributed approach allows agents located in any server in the DC to make localized management decisions.
- **Granularity**: The time frame to react to events depends on application characteristics and on the current state of the application. For instance, for lower response times to failure we may need to increase the granularity of monitoring and make decisions locally and independently of centralized management systems. To be able to achieve higher processing and monitoring granularities, some of the load should be processed locally and communication should be spread throughout the network.

In our approach, resources are managed by multiple autonomous entities. Each of these entities builds its own database of free resources by establishing peer-to-peer communication with other resource management entities. Each entity will be able to make resource allocation decisions independently. On the other hand, an independent autonomous agent will manage each user application. These agents are also distributed across the network. The agent behavior will depend on the application characteristics and performance goals. As a result of this distribution of the management load, each agent will be able to make more granular decisions and access data with finer granularity.

2.1.2. Dynamic Adaptation

Current DCs environments are highly dynamic in time and space. Time-variant nature arises from constantly changing workloads and from events that affect the capacity of the DC infrastructure. The granularity of these changes can vary from a few microseconds to months. The changes can also be gradual over a period time or bursty. Additionally, failures can make a server unavailable or cause performance degradations. On the other hand, technologies like virtualization allow easy migration of applications across the DC. Therefore, congestion hotspots can change quickly between different locations on the DC.

To handle the dynamic nature of the DC environment, each agent representing a provider or an application implements specific actions to adapt to:

- **Changes in infrastructure**: The agents managing a set of physical resources should adapt to variations in infrastructure like changes in network topology or addition or removal of servers;
- **Adapt to volatile workloads**: Some services have very volatile workloads. The agent should include mechanisms to quickly provision extra resources to handle workload bursts.
- **Resilience to failures**: Failures occur frequently in DCs. The agents should consider instances' failure dependency to avoid application downtime. The agent should quickly react to failures guaranteeing that the application recovers to normal operation within the limits specified in its SLA.
- **Adaptable Overhead**: The use of management tools like monitoring or VM migration should be adjusted to the characteristics of the application. Management details concerning, for example, monitoring granularity and reaction time should depend on the requirements of the application. Low response time applications need faster adjustments than long response time or batch applications.

LAMA provides features to allow fast and effective reaction to workload changes and failures.

2.1.3. Integrated Management

We use distributed and dynamic features of the platform create an integrated management environment for applications. Applications are managed by agents deployed in the cloud environment. Each application has specific characteristics and QoS demands. An integrated approach allows creation of differentiated management strategies to satisfy specific application requirements. Building centralized systems with differentiated management algorithms is complex and incurs in high overhead.

LAMA's integrated management approach has two main advantages:

- **Holistic view of the application environment**: The agents have information not only about the instances that compose the application, but also about their hosts. This allows agents to better diagnose application issues and create recovery plans for potential failures. Thus, we can build more dynamic and efficient management algorithms.
- **Proximity to data**: In traditional systems, application management is done outside the cloud. An integrated approach allows app agents to be close to the instances they manage. This enables real-time access to performance data and, consequently, faster reaction to events that affect application performance.

Agents that manage applications maintain a view of the application that includes: a logical architecture based on user specifications, a mapping of each logical component to the allocated physical resources and an estimate of the application's performance. Based on the evolution of its view of the application, the agent decides the necessary steps to guarantee current and future performance of the application. LAMA provides a programmable environment that allows users to define their own management strategies.

2.2. Multi-Agent System (MAS)

In its most basic form, Cloud Computing defines a pool of resources to be shared by several different applications. Thus, there are two main types of entities in the DC: *providers*, which represent entities with resources to be shared, and *applications*, which represent entities that consume resources. Providers are more efficient when all their resources are fully utilized. Applications need to be able to access resources, whenever they require them. This leads to conflicting interests between providers and applications and among different applications sharing resources.

2.2.1. The Agents

The main building block of our system is a multi-agent system (MAS). Each agent operates autonomously representing an entity in the DC. Thus, we define three basic types of agents:

- **Provider Agents**: Autonomous entities responsible for managing a small set of resources. The providers control the allocation of resources. In this sense, providers represent the management system at each resource hub. The provider agent can be local to the provider or remote. For instance, a provider agent managing resources of a server can run on that same server. For a set of network resources, we can have an agent deployed on a server controlling those resources.
- **Application (or App) Agents**: Autonomous entities responsible for acquiring the resources an application needs to operate with a specified performance. Complex applications can be distributed throughout the network. Thus, one application can have multiple agents deployed close to the allocated resources. Also, in a virtualized Cloud Computing environment, application services can migrate between servers. Likewise, application agents should be able to move around the DC to be closer to where the application resources are currently reserved.
- **Dispatcher Agent**: Centralized autonomous entity responsible for support operations. The dispatcher maintains a registry of provider and app agents in the system. Each agent authenticates itself to the dispatcher during initialization. The dispatcher suggests new peers for provider agents and handle new applications requests from cloud users by assigning them to a new provider agent. The dispatcher is not involved in actual management of applications.

To understand how our MAS agents fits into the DC infrastructure consider the layered diagram of Figure 1. The typical virtualized environment is composed by a hypervisor that controls the access by the

VMs to the physical resources. The provider agents are placed on top of the physical machine operating system. The provider agents work in close connection with the hypervisor executing VM management (e.g. create and delete VMs, set resource allocations, migrate applications, etc.) and monitoring actions. Application agents are logically placed on top of provider agents, as they do not interact directly with the operating system or the hypervisor. Each application will deploy an application agent on servers where instances of each service are deployed. One of the application agents is elected as application controller. The controller maintains information about the application (i.e. architecture, dependencies, current allocation and performance) and defines actions to improve the performance of the application or its predicted resilience. If the controller fails, any application agent can take over as the new controller.

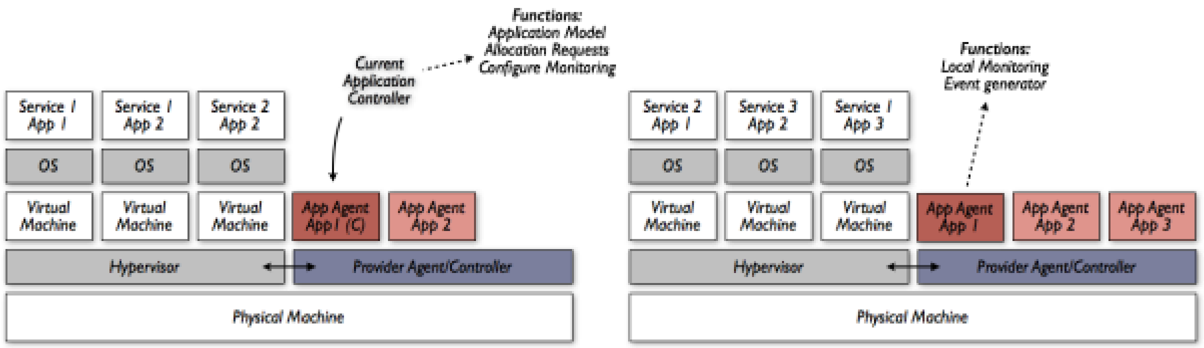


Figure 1 – Architecture of the MAS with provider and application agents.

2.2.2. Agent Architecture

The generic architecture of an agent in a MAS is depicted in Figure 2. Every agent has a central unit that processes information about the environment collected by sensors. This information is used to create a model, which represents a partial view of environment. The model contains only the environment information that is relevant to the agent's goals. Each agent has a set of mechanisms to influence the state of the environment: the effectors. Finally, each agent needs to interact with other agents to either gather information or to indirectly change the environment.

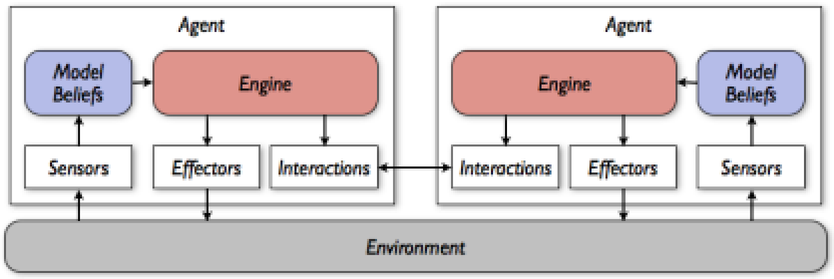


Figure 2 – Generic logical agent architecture: every agent maintains a view of the environment using sensors' results and uses effector to change that environment according to their interests.

In our MAS, we implement two types of agents representing applications and resource providers. Table I contains a short description of the purpose of each component for each type of agent. The main difference is that application agents' models are centered on the application while the provider agents' models are based on the resources they manage. It is important to note that only providers can directly reserve resources in a physical server. This is important because application agents represent the interests of external users. Therefore, the provider can implement access controls to the physical resources (for instance, constrain resource allocations or control which VMs can be co-located for security purposes).

Table I – Functions of agent components per agent type.

	Application Agent	**Provider Agent**
Model	The agent builds a model of the application architecture including its services, dependencies among services, dependencies on DC services and on fault domains. It also maintains allocation information and current performance of the application.	The agent maintains information about (1) a subset of available resources, (2) the current state of the provider including allocations, resources available, and application behavior.
Sensors	The agent monitors the performance of the application services.	Sensors monitor resource usage per tenant application.
Effectors	The agent does not directly actuate on the environment. Instead, all resource allocation requests go through provider agents.	The agent deploys application agents and sets resource allocations for services.
Interactions	Interacts with provider agents to request resource reservations.	Interacts with other providers to create a view of available resources. Interacts with applications to respond to allocation requests and alert for resource contention or failures.

2.3. Application Model

In LAMA, clouds users specify their applications (or simply apps) structure and requirements by defining a graph of *services*. Services designate logical components of an application (e.g. database, web server). The application agent will use that specification to determine what support services are required, the number of virtual machines and their allocations. The complete application model is depicted in Figure 3.

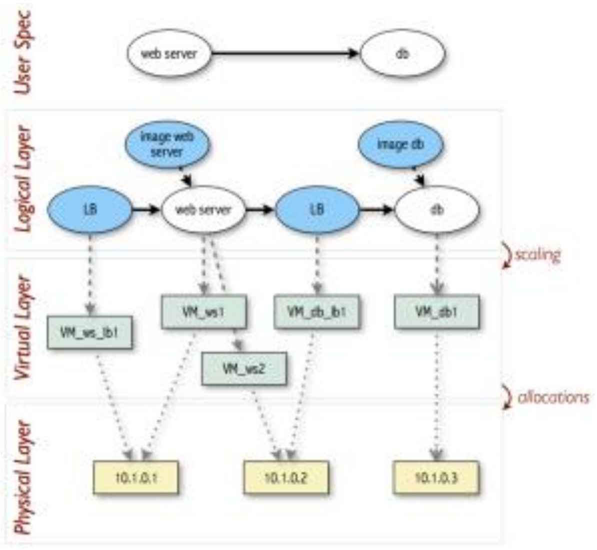


Figure 3 – App Architecture Graph.

The user spec is a high-level representation of the application provided by the user. The logical, virtual and physical layers are internal representations used by LAMA agents.

2.3.1. User Spec

The application is defined through a *user spec*, a GraphML [21] document that defines the structure of the application. The user references the images of each service, service level requirements (or SLA) for the application, and other service configuration parameters (e.g. allocation constraints, minimum resource requirements, auto-scaling configurations, etc.). An example of a user spec is defined in Figure 4.

```
graph [
  name "rubbos"
  label "RuBBoS (MySQL)"
  directed 1
  connector rubbos
  strategy resilience_graph
```

```
    sla_period 200
    sla_performance_index 0.9
    sla_miss_fraction 0.1

    node [
       id client
       label "Client Generator"
       image "../rubbos-client.qcow2"
       subtype application
       user "lama"
       password "lama"
    ]

    node [
       id apache
       label "Apache Web Tier"
       image "../rubbos-apache.qcow2"
       subtype application
       scaling auto
       scaling_port 80
       user "lama"
       password "lama"
    ]

       node [
       id "mysql"
       label "MySQL"
          subtype service
       image "../rubbos-mysql.qcow2"
          image_init ../rubbos-mysql.init.qcow2"
       user "lama"
       password "lama"
       resource_ram 2GB
    ]

    edge [
       source client
       target apache
       label "Client Flow"
    ]

       edge [
       source apache
       target mysql
       label "DB Flow"
       type client
    ]
 ]
```

Figure 4 – Sample User Specification of an Application.

The user-provided logical specification directs the agent to create an internal logical architecture. The graph can have the following properties:

- ***name***: The name used to identify the application. This should be a unique identifier. Thus, LAMA might add a suffix to guarantee that the name is unique;
- ***label***: A user friendly name for the application (not necessarily unique);
- ***connector***: A user-defined class with application specific code to update instances. LAMA defines a series of hooks in the management workflow to handle application lifecycle events. Example of these events are the deployment or termination of a new instance. The user is responsible for defining code to setup instances (e.g. edit configurations, run applications specific commands, install packages, etc.);
- ***strategy***: The user can define a strategy to use to manage the application. The strategy determines how the app agent scales the application, reacts to failures, or what to metrics to monitor and create for application visualization.
- ***strategy parameters***: The user can specify custom parameters to be used by the management strategy. In the example of Figure 4, we define three SLA parameters. These are used by an SLA-based management strategy that we present in Chapter 6

Each node specifies a service. Nodes can have the following attributes:

- ***id***: An identifier of the service that must be unique within the application.
- ***label***: A user-friendly name for the service;
- ***subtype*****:** Each service can have two subtypes. The default type is *application* to indicate an application service. The other type is *client*. As LAMA was designed as research framework, we introduce the client subtype to indicate a service that will hold a client generator. This type of service is treated a little differently than application services, as LAMA guarantees that there is one client instance ready to be used to generate workload. Once a client is started, LAMA launches a new client instance for a new workload. We use this feature to run fully automated experiments in our cluster;
- ***image***: Identifies the image to use for the new instances.
- ***image_init***: Identifies the image to use for the first instance. For some applications, the user might want the first image to be different than the subsequent images. For instance, in MySQL Cluster setup, the initial images can be load with data in the database. Subsequent images added during scaling will receive state from the remaining images. Alternatively, the user can define a single stateless image and create some procedure to download the data required. This alternative normally takes longer to deploy but creates smaller images.
- ***user, password***: Credentials to access the instances.

- ***resource_****: Users can specify the amount of resources to be used for instances of each service. It can be used to indicate the following resources: CPU Speed, CPU Cores, RAM, Disk Speed, Disk Space, Network Rate.
- ***scaling*** *[optional]*: If 'true', indicates that a service can be horizontally scaled, i.e. the app agents can add additional instances. If the value is set to 'auto', indicates that LAMA should handle the scaling of the service automatically. In this case, the app agent will add a load-balancer service to handle request distribution. The user should define the parameter ***scaling_port*** to indicate which port is listening for requests.

Edges represent a relationship between services. They are used to indicate dependencies, so that LAMA knows which services need to be update upon changes in the application deployment. They are also useful for visualization purposes. Edges can have the following attributes:

- ***source***: The client service in the relationship;
- ***target***: The server service in the relationship;
- ***label***: A user-friendly name for the relationship;
- ***type***: An edge of type *client* indicates that the source nodes should be updated, whenever there is a change on the target node. LAMA ensures that these updates are triggered whenever new instances are added or deleted. By default, LAMA does not trigger those updates.

2.3.2. Logical Layer

The logical layer contains the *services* that compose the application. It is constructed by adding support services to the structure defined in the user spec. For each image defined in the user spec, a new image service is added. The image services are responsible for storage, backup and maintenance of VM images. For each image service, the app agents will have to reserve the necessary disk space before the image can be uploaded. A dependency is explicitly created in the application graph between the image service and the service itself. App agents can take that relation into account when evaluating and managing the application.

For services marked as scalable, app agents add a new support load-balancer as a predecessor node. The load-balancer services manage load among instances of services that are designed to scale. From this point on, the load-balancer service is handled the same way as any other application service.

2.3.3. Virtual Layer

The virtual layer represents the virtual *instances* that compose the application. A service can comprise multiple instances. If the service is scalable, the number of instances is increased to accommodate new workloads. If not, backup instances can to improve the application resilience to failures. The app agent is responsible for determining how many instances must be deployed for each service. By default, the app agent deploys an instance per service and then scales it according to workload variations.

2.3.4. Physical layer

The physical layer contains the servers used by the instances of the application. The app agent sends requests for new instances to the provider agent. The requests can contain resource requirements and constraints to where each instance can be allocated. The provider agents use its peer network to find servers that meet the request requirements.

As we will see in Chapter 6 this holistic view of the application structure is used to identify potential failures and plan how to recover from them.

2.4. Provider Agent

Provider agents are designed to provide features for app agents to manage their applications. These functions include searching and reserving resources, launching app agents for new applications, subscribe relevant monitoring data, and execute virtual instance operations. Dynamic management strategies are enabled by the distributed nature of all management tasks in LAMA. Provider agents can execute most cloud management tasks independently. This means that multiple app agents can adapt to events in parallel by contacting their local provider agent, thus accelerating recovery from failures.

2.4.1. Architecture

The architecture of a provider agent is depicted Figure 5.

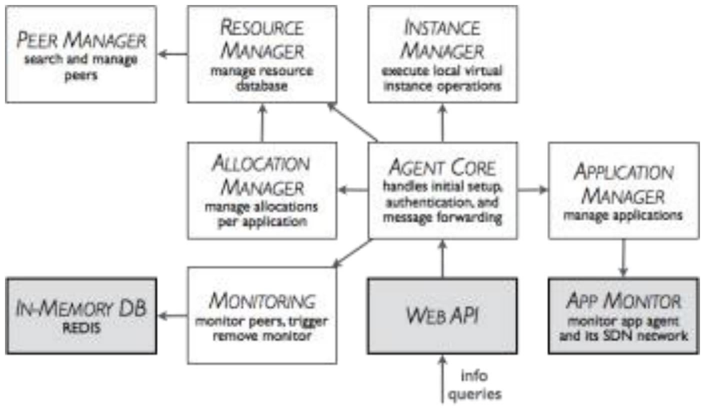


Figure 5 – Architecture of Provider Agents

2.4.1.1. Agent Core

The core module of the agent is responsible for:

- **Initialization**: The core module configures the agent (determine current address) and starts the remaining modules.
- **Authentication**: The core module authenticates the agent with the dispatcher agent. It receives a new identifier when it is first initialized.
- **Communication:** The core module handles all events from other agents. Upon reception of a new event, it determines which module(s) can handle the event. The agent also starts up a web interface to answer queries about the status of the provider and local application agents. LAMA includes a web interface that allow cloud providers and users to access information on the state of the infrastructure and applications. As each provider agent is able to answer information about itself and the app agents it hosts, we are able to distributed the web interface workload among all the DC servers.

2.4.1.2. Resource, Allocation and Peer Managers

The resource manager, the allocation manager and the peer manager are the modules responsible for creating a database of resources and allocating resources to applications.

The resource manager maintains the local resource database. On initialization, it needs to scan the resource available at the local host. This scan is reviewed periodically. To increase the number of resources in the database it requests a new peer search. This drives the peer manager to search for new provider agents with resources available. When new peer provider agents are found, its resources are added to the database. These resources are used to answer to allocation requests from local app agents.

The allocation manager manages the distributed workflow of a new allocation requests. When a new resource request is received, it is first reserved in the local database. This initial reservation is then reserved remotely. The allocation process terminates once the reservation is confirmed. The allocation process is described in more detail in Chapter 3. We study its performance under different workloads and cluster state.

We designed LAMA to be easily extended to support allocation of any type of resources. This allows providers to advertised any type of resources or capabilities. Currently, the LAMA framework supports allocation of disk space, disk IO, RAM, CPU usage and network I/O. It also allows providers to advertise the CPU speed of their servers. Thus, app agent can impose CPU constraints for new instance requests. For instance, an app agent can request that allocations are made only for servers with a minimum CPU speed.

2.4.1.3. App Manager

The app manager is responsible for initialization and management of local app agents. When a new application request is received, the app manager creates the app agent. The app agent is initialized using the user spec for the application.

For each application, LAMA creates a virtual private network (VPN). The VPNs are deployed using Open vSwitch [22] virtual bridges. Open vSwitch allows us to setup a Software Defined Network (SDN). The app manager deploys an SDN controller, to allow the app agent to control the communication between the application's instances and a DHCP server process to assign private IP addresses to new instances. An application agents runs in their own network namespace and its process uses a low-privilege user. Thus, they do not have access to the network interfaces or data in the physical host. The VPN allows application instances to communicate not only among themselves but also with the application agent. This allows the instances to publish application specific metrics directly to the app agent.

The app manager includes an app monitor, which is responsible for monitoring the app agent and network processes and for restarting them if they die. Finally, the app manager is responsible for forwarding requests to and from the application to other providers or to respond to requests from the web interface.

2.4.1.4. Instance Manager

The instance manager is responsible for instance deployment. It includes function like copying instance images from other provider agents and define, create, remove or change the state of instances. We use the KVM (Kernel-based Virutal Machine) hypervisor to run virtual machines. Virtual machines are managed through the libvirt API.

2.4.1.5. Monitoring

Provider agents manage a monitoring sub-system with two main goals:

- provide app agents with data from entities in the application's environment (hosts, instances and network used by the application);
- allow users to customize methods for detecting and reacting to events that affect application's performance.

Local resource monitoring is done using collectd [23], a plugin-based open source monitoring tool. A key-value in-memory database (using Redis[24]) provides a publish-subscribe mechanism that notifies agents of new metrics in real-time. Redis serves as a proxy to all metrics. App agents can ask the local provider agent to subscribe metrics for hosts or instances that are part of their application environment.

Provider agents manage all remote subscriptions. This avoids abuse from app agents and redundant subscriptions. It also provides accountability of the volume of monitoring data per app agent. Explicit subscription from app agents guarantees that providers only subscribe metrics from remote peers that are actually going to be used by app agents.

Additionally, provider agents allow app agents to request data with a specified granularity. This allows app agents to adjust the monitoring behavior according to application needs [25]. However slower provider agents might limit monitoring granularity options according to its capabilities.

2.4.2. Active and latent resource allocations

One of the main goals of the LAMA framework is to allow fast reaction to failures and other events that can affect application performance. LAMA implements a feature that allow app agents to pre-plan the future evolution of the application in case of potential failures.

LAMA provider agents can allocate resources for **active** or **latent** instances. Active instances run and process application workload. Latent instances have resources reserved for later requirements. Latent instances can be in *cold*, *warm* or *hot* states, depending on what stage they are in the provisioning process. These instances are used by app agents to pre-plan recovery from failures. For active instances, provider agents reserve all the resources required by the instance. For latent instances, reservations of certain resources are made according to particular later event. Provider agents need to guarantee that, for a given event, resources are not over-allocated above a configured limit. This customized local limit allows provider agents to over-allocate resources to increase server resource utilization. Table II shows which resources are allocated per provisioning stage. Active allocations always have precedence over latent reservations. If a new active resource allocation leads to an overload situation, the provider agents remove affected latent allocations and notify the corresponding peer or app agents. On the other hand, app agents notify provider agents whenever the dependencies of latent allocations change.

Table II – Possible instance states. The table shows which resources types are active or reserved (A) or Latent (L) for each instance state.

			RESOURCES ALLOCATED				
LEVEL NO.	STATE	PROVISIONING STAGE	DISK SPACE	DISK I/O	RAM	CPU	NET I/O
1	**Cold**	Host/resources reserved.	L	L	L	L	L
2	**Warm**	Image copied to host.	A	L	L	L	L
3	**Hot**	Instance created+paused.	A	A	A	L	L
4	**Active**	Instance active.	A	A	A	A	A

2.5. Application Agent

Application agents are autonomous entities that continuously monitor and adapt to changes in the state of the application. Its decisions are determined by monitoring strategies that a user can customize to fit their goals to the application.

2.5.1. Architecture

The architecture of app agents is depicted in Figure 6.

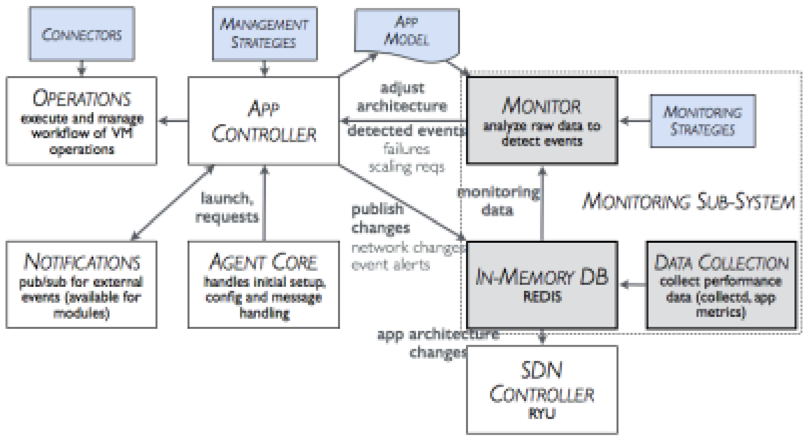


Figure 6 – Architecture of Application Agent.

The application agents share the same **core** as provider agents. Thus, module initialization and communications between them operates in the same fashion. However, app agents authenticate to their local provider agents. App agents can only start allocation requests after authentication.

The **app controller** module maintains a view of the current app model and is responsible for coordination between the different modules. It also can publish relevant events of the app's life-cycle to cloud users. The app controller initializes a user-defined **management strategy**. Management strategies use the application graph as a view of the application structure. They implement a custom evaluation of the status of the application. The evaluation is based on the structure of the application and monitoring data available from LAMA's monitoring infrastructure. Management strategies are typically triggered by a monitoring strategy. Users can subscribe to common events provided by the framework (e.g. host or instance failures) or define their own monitoring rules or algorithms. The goal of these strategies is to determine provisioning operations (e.g. add or remove active or latent instances) for scaling and responding to failures. In Chapter 6, we propose and evaluate a dynamic management strategy for multi-tier web applications.

The **operations** module handles the workflow of all instance actions (e.g. start, remove, migrate). For instance, starting an instance includes a series of commands to copy the instance image, set up network and verify connectivity. Users can provide custom code to connect to instances and execute custom configurations or verify application's status. The 'connector' attribute in the user spec (see Section 2.3.1) is used to define a class that handles instance configuration.

Each application runs in its own virtual private network. The **SDN controller** manages the application's private network and enforces any security constrains imposed by the user. The app agent cannot access the SDN controller directly. Instead a local in-memory key-value database (based on **Redis**) is used to provide communication between the SDN controller and the rest of the modules. This interface allow us, to limit app agents actions for security purposes.

The in-memory database is also used to provide communication between the low-level monitoring tool and the monitoring module. The **monitor** module collects raw data and detects high level events. It allows a user to configure **monitoring strategies**. These strategies define custom code that can generate higher level events (e.g. failure detection) from low level system metrics (e.g. resource usage). LAMA provides a set of default algorithms that can be configured by the users (e.g. state machines for failure detection, window-based analysis and rule based decisions).

2.6. LAMA as a Research Framework

LAMA's characteristics make it an ideal framework for research purposes. LAMA was designed to be an open environment that allows the integration of advanced app management algorithms. Cloud users may deploy customized management code for their applications. This code can make use of LAMAs management API that includes operations for monitoring (subscribe metrics about hosts and instances, setup aggregations metrics, etc.) and provisioning (create, remove active or backup instances).

LAMA also includes other features and modules to enable fully automation of experiments and data collection:

- **Client Services**: As we have seen LAMA allows client services to be specified in the user spec for the application. Client services are used to generate workloads for applications. They can also publish metrics through the application's VPN.
- **Event Publish System**: Controllers in application agents publish relevant application lifecycle events to the in-memory database. LAMA provides an option for cloud users to subscribe to these events.
- **Scenario Constraints**: LAMA allows constraining which provider agents should have peer relationships and instance allocation. This allows creating very specific scenarios for analysis.
- **Automation Packages**: We developed two additional packages for emulating failures in instances and host (LAMA Thanatos) and to automate schedule of experiments (LAMA Experiments). LAMA also includes management commands that enable users to remotely install provider agents and to stop, reset, and restart each provider or the entire LAMA framework.

2.6.1. LAMA Thanatos

LAMA Thanatos is a distributed failure emulation tool. An agent is deployed on each of LAMA servers. In its current version, it allows emulation of failures at the host level (host crash and resource congestion) and instance (instance crash) levels.

2.6.2. LAMA Experiments

LAMA Experiments is a distributed experiment scheduler. This tool was used for all experiments performed in our testbed cluster. Experiments can be configured using YAML files. An example of an experiment configuration is presented in Figure 7. This sample experiment starts by deploying new app. Once that app is active, it schedules three more events: the start of a client workload generator after 60 seconds, a crash of a random instance of service 'apache', and the termination of the experiment after 700 sec. LAMA experiments allows us to easily launch large scale experiments.

LAMA experiments has two additional relevant features:

- **Parallel/Concurrent Scheduling**: LAMA Experiments can deploy helper agents in all the servers in the cluster. This enables us to schedule multiple parallel requests. This accelerates the execution of multiple concurrent requests to test high load scenarios.
- **Real-time Reporting**: A command line tool allows us to configure an HTML report with a timeline of the events generated during the experiment and time-series charts of resource usage metrics.

```
duration: 1500                                          # maximum total duration
users:                                                  # default user for the experiment
  user1:
    lama:
      username: rubbos_01
events:                                                 # at 10 sec, create a new
- time: 10                                              # app named rubbos-001
  actions:                                              # with user spec defined # by
    - cmd: create_app                                   rubbos-mysql
      app_name: rubbos-001
      app_type: rubbos-mysql                            # configuration
      config:                                           # parameters for strategy
        sla:
          sla_period: 200
          sla_performance_index: 0.9
          sla_miss_fraction: 0.1                        # 60 sec after rubbos-001 # is
- time: 60                                              active, launch a
  dependencies:                                         # client generator for
    - active_app:                                       # app named rubbos-001
        app_name: rubbos-001
  actions:                                              # configuration to be
    - cmd: launch_client                                # used in the client (the # app
      app_name: rubbos-001                              connector should
      config:                                           # know how to apply it)
        workload_number_of_clients_per_node: 300
        workload_up_ramp_time_in_ms: 60000
        workload_down_ramp_time_in_ms: 10000
        workload_session_run_time_in_ms: 600000
        monitoring_debug_level: 0
        monitoring_all_response_times: 1                # 180 sec after rubbos-
- time: 180                                             # 001 is active, generate # a
  dependencies:                                         failure (crash a
    - active_app:                                       # random instance of the
        app_name: rubbos-001                            # ‘apache’ service
  actions:
    - cmd: failure
      failure_type: instance_crash
      app_name: rubbos-001
      params:
        service: apache
        choice: random                                  # 700 sec after rubbos-
- time: 700                                             # 001 is active,
  dependencies:                                         # terminate the
    - active_app:                                       # experiment
        app_name: rubbos-001
  actions:
    - cmd: experiment_end
```

Figure 7 – Sample Experiment Configuration.

2.7.Assumptions

Assumption 1. We consider an Infrastructure-as-a-Service scenario with a high number of small to medium applications

This work addresses the problem of high workload and overhead of managing a high number of applications centrally. LAMA is designed to distribute the management load of different application among the hosts of the DC. Thus, LAMA assumes a public-cloud Infrastructure-as-a-Service scenario with a high number of small to medium applications (a few tens of virtual instances). Thus, we assume that the cost of management and monitoring of each application is low. As currently deployed, the application agent is indivisible. This means that as an application grows, the processing overhead of the application agent increases. The application agent uses host resources and might interfere with the co-hosted applications.

In our design, we include the possibility of using multiple distributed agents per applications. As we have seen, the user can define and deploy multiple monitoring strategies. Each of these monitoring strategies can run in a different agent. In the future, we plan to study how a management strategy for a single application can itself be distributed. In a production environment, the complexity of management strategies should be carefully evaluated before deployment.

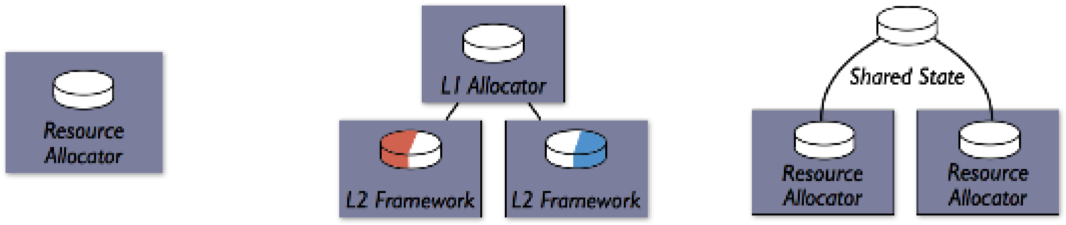
Chapter 3

Distributed Resource Allocation

We study the advantages of distributing DC management functions. The first basic operation in DC management system is to reserve computing resources to user applications

3.1. Introduction

Traditional resource allocation is done centrally with a global database of system metrics. They are split into three different types shown in Figure 8. In a monolithic approach, as used in OpenStack, a single resource allocator processes one request at a time and accesses the entire DC state. On a 2-level approach, as used in Mesos [26], a first-level resource allocator assigns request to multiple second-level allocation frameworks. Disjoint subsets of the DC resources are split among different second-level frameworks. In a shared state approach [27] multiple parallel resource allocators access copies of the global state of the DC. Despite the use of parallel processing, the shared state approach is still centralized in nature as all allocators access the same global state. The main advantage of these systems is the ability to access the complete DC state to process allocation requests. On the other hand, as the DC grows, the overhead of maintaining that centralized state also increases. Monolithic approaches cannot handle the ever-growing demand in DCs and suffer from head of line blocking. Complex requests can block processing of other simpler requests. The two-level approach constrains the resources that can be used by each allocator simultaneously. It sporadically limits the capacity to make decisions and creates blocking. The shared state approach generates allocation conflicts. Due to its inherent centralized nature, collision is only detected after a request was complete locally. In this situation requests need to be processed again.



(a) Monolithic approach. (b) 2-level approach. (c) Shared state approach.

Figure 8 – Typical resource allocation approaches.

We apply our fully distributed approach to process resource allocation requests. We take advantage of the placement of resource allocators next to the resources themselves. The provider agents deployed on each DC server should be able to make autonomous allocation decisions. However, a single provider may not have enough resources to accept allocation requests from complex applications. Thus, we define interactions among provider agents that enable them to make allocation requests remotely. When humans are presented with a problem, they naturally resort to their close friends or connections that might have the necessary resources to solve it. Likewise, each agent constructs and maintains a short-list of other peer agents that have access to free resources. Agents communicate among themselves to gather information about resources.

An application setup starts with an application agent being deployed randomly in any DC server. The application agent starts the allocation process by sending an allocation request to the provider agent in the same server. Provider agents maintain short-lists containing references to other provider agents. We refer to these other provider agents as *peers* as they communicate using a peer-to-peer overlay network. The provider agent uses the short-list to determine if it can satisfy the application agent's allocation request. There are four main components:

- *Provider Short-List*: describes how provider agents setup and maintain short-lists of peer providers;
- *Allocation Protocol*: describes how provider agents make allocation decisions based on their short-lists;
- *Short-list Maintenance*: describes how provider agents adapt their short-list to external events;
- *Resource-based Routing*: describes how provider agents redirect allocation requests they cannot fulfill.

3.2. Provider Short-list Setup

3.2.1. Short-list Properties

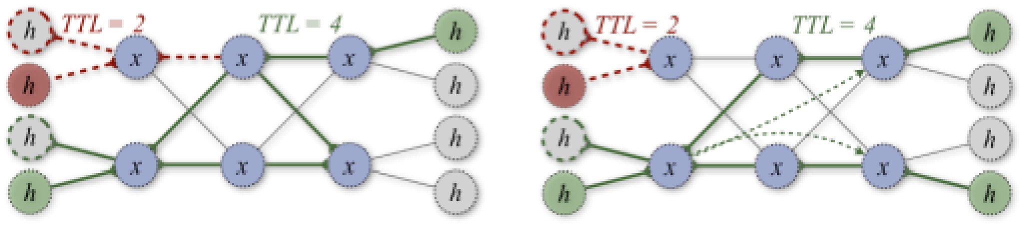
At the initial stage, provider agents are not aware of other provider agents. We develop a protocol to find and create a shortlist of other provider agents. We consider three design goals:

- *Independence:* As DC utilization increases, we want different provider agents to have distinct lists of peers. Thus, provider agents should create their short-list independently. Diversified short-list increase the probability that, if a provider cannot satisfy one allocation request, one of its peers might;
- *Locality*: During the allocation phase, provider agents select other peers to provide services of the same application. It is often best for application's performance that its services are placed in servers close to each other. Thus, the choice of peers prefers the nearest agents;
- *Controllability*: The protocol should have low communication overhead to avoid affecting applications performance.

3.2.2. Random Hop: Randomized Hop-by-Hop Forwarding

Resource discovery is typically based on centralized directories, broadcast or multicast queries or random search [28]. Using a centralized directory-based protocol violates our goal of using fully distributed protocols. Using multicast solutions to find provider agents, leads to the message being broadcast to the entire network. The overhead would be uncontrollable given the high redundancy present in the DC networks. Thus, we use a randomized protocol to route packets. Contrary to the protocol proposed in [28] we do not know where direct neighbors are nor do we intend to find all the nodes in the network. We do not copy lists of peers or else we would violate the *independence* design goal.

Provider agents send resource probe messages that are randomly routed through the network. When another provider agent receives a probe message, and it has enough resources available, it replies directly to the agent that sent the message. Upon receiving this response, the provider agent verifies if the peer provider is already on the short-list. If not, it adds the new peer. Each provider agent repeats this procedure until it gathers the necessary number of peer agents. In order to meet the *locality* design goal, we progressively increase the TTL value in the packet. Figure 9a shows possible random routes that request packets can follow for different TTLs. The dashed line represents a request for TTL equal to two hops. In this case, the host will only be able to add peers located in the same cluster. The thicker lines represent a request with TTL equal to four. At this point, the provider agent randomly finds peers located farther away. However, the probability that the packet is lost due to TTL expiration is also larger. Traditionally TTL is only present on IP header. However, hop counts are also available in a recent layer-2 proposal through TRILL [29]. Thus, the first few packets reach the closest neighbors. As the TTL is increased, the provider agent finds peers farther away.



(a) Random routing hop-by-hop (b) Random routing by edge device

Figure 9 – Possible routes taken by peer probing packets sent by provider agents when extending their short-lists.

We also control link utilization to meet the control design goal. As the provider agent only sends one packet at a time, it can pace the packets in order to maintain network utilization at any desired level. Given that the protocol is resilient to packet loss, its random nature and low overhead requirements, we run it over UDP.

3.2.3. Publish/Subscribe

A provider agent needs to stay informed about changes in peers in its short-list. This information is updated through a publish/subscribe mechanism. Each time the provider agent adds a peer to the short-list, it sends a subscribe message to that peer. Every time the resources available change, the provider sends a message to all the peers that subscribed its updates. Each update carries the total resources available in the provider agent's server.

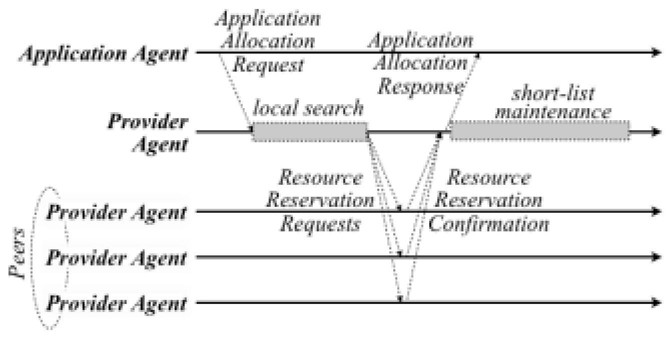
3.2.4. Random Edge: Reducing Spurious Request Packets

Since we are using random routing, some packets may be lost without ever reaching an end node. Depending on the technology used in the DC [16], [2], [17], [18], the protocol may be optimized to reduce the number of probe packets that do not reach another provider. If we assume that the edge nodes know the routes and distance to other edge nodes, we can eliminate the packets that are lost in the network core. Instead of choosing a random hop to forward the packet to, each edge switch chooses another edge node randomly. It can use the distance to the other edge node as a reference. Figure 9b illustrates this behavior. Notice that the packets are sent directly to the edge nodes. They no longer go through hops where, given the limited TTL, they will not be able to find a peer provider.

Naturally, this solution immediately assumes some knowledge about the routing technology in the DC. However, we can adapt this solution for most protocols used in the DC today. As an example, consider TRILL. TRILL has knowledge about the existent edge rbridges. In this case, we can randomly select the edge rbridge and forward the packet. The destination rbridge will then randomly forward the packet to one of the servers directly connected to it.

3.3. Multi-Resource Distributed Allocation

Each provider is able to respond to allocation requests from any application agent. The allocation workflow is depicted in Figure 10. The allocation process is triggered when the provider receives an application allocation request from an application agent. The provider looks into its short-list of provider agents to verify that they have enough resources to satisfy the request. If this local search is successful, it starts the second phase of actually reserving those resources at the corresponding provider agents. Once and if it receives confirmation that resources were allocated it sends a successful response to the application.


Application Agent
Application Allocation Request
Application Allocation Response
Provider Agent
local search
short-list maintenance
Resource Reservation Requests
Resource Reservation Confirmation
Peers
Provider Agent
Provider Agent
Provider Agent
Figure 10 – Distributed Resource Allocation

3.3.1. Initial local search

An application request is composed by a set of *n* services each with resource requirement vector $R=[0,M]^K$. The provider agent has a shortlist of peers, which can be defined by a set of *p* provider agents each with available resources vector $A=[0,M]^K$. The goal of the local search is to verify if the set of services can be satisfied by the set of provider agents in the short-list.

The goal of the provider is to quickly return a response to the application. Thus, in this paper, we take a simple greedy approach to this problem. For each service, the algorithm tries to find a provider that can fit the service. If we are able to allocate all the services, the request is successful. If not, the provider keeps track of the services it was not able to allocate. In this case, the provider can take two actions:

- If it finds a provider with more available resources in its short-list, it forwards the request to that provider. This resource-based routing procedure in described in Section 3.5;

- If it does not find a better provider it starts a procedure to adjust the short-list until it is able to fit all services requested by the application. This short-list adjustment procedure is described in Section 3.4

3.3.2. Resource reservation on peer providers

If the provider finds enough resources in the short-list, it has to actually reserve resources on the selected peer providers. The provider sends one Reservation message per selected peer. Once all peers confirm the reservation, the provider sends a response with a list of the assigned servers per service to the application agent.

3.3.3. Trigger short-list maintenance

Once the allocation process is complete, the short-list maintenance procedure is triggered. If the allocation was successful, the provider must verify whether the peer providers used to allocate resources need to be replaced in the short-list. If the allocation process failed, the provider needs to improve the short-list in order to be able to meet the next request.

3.3.4. Termination

Given our distributed approach, each provider agent does not know if any provider in the DC can satisfy an allocation request. The agent has only knowledge of its own resources and those in the short-list. Therefore, it continues to improve its short-list to find better providers or enough resources to fulfill the request. Given that we control the overhead imposed, this continuous search has not much impact in performance. However, at some point the provider needs to give up and return a negative answer to the application agent. The application agent can set a time limit for allocation response. Additionally, provider agents also automatically set a time limit to make an allocation decision. We use a threshold of 100 times the average time it takes to complete successful requests computed over a sliding window. In Section 3.6.2 we demonstrate that the allocation time is stable enough to set that limit.

3.4. Short-List Maintenance

During its lifetime, the goal of the provider is to have a short-list of peers that allow him to successfully allocate resources to applications. As allocations are made, the state of providers changes. Thus, each provider needs to dynamically update its short-list.

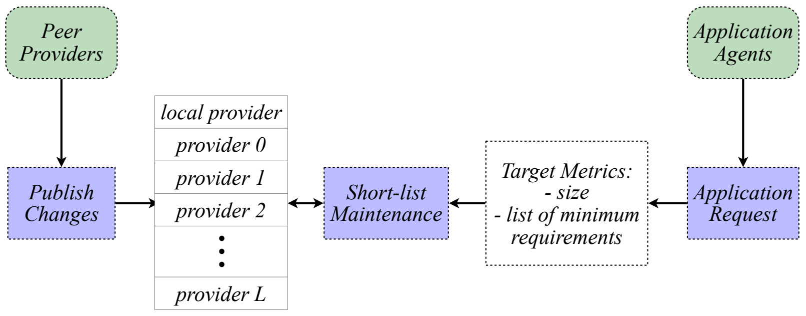


Figure 11 – Short-List Maintenance

The short-list maintenance process is represented in Figure 11. A provider's short-list at a provider may become outdated for three reasons:

- The provider has just received an application request. In this case, two situations may occur:
    - The application request was successful: in this case the short-list has now less resources available than before;
    - The application request was not successful: in this case, the provider must improve the short-list to be able to satisfy the next request.
- A provider in the short-list has reserved resources for some applications and sent a message updating its available resources;
- Providers have become unavailable due to management decisions or failure.

The short-list is continuously updated according to a set of metrics. Initially, the provider uses a single parameter: the size of the short-list. We described the initial short-list setup in process in Section 3.2. After the initial setup, the process is repeated every time the short-list does not fulfill the target metrics.

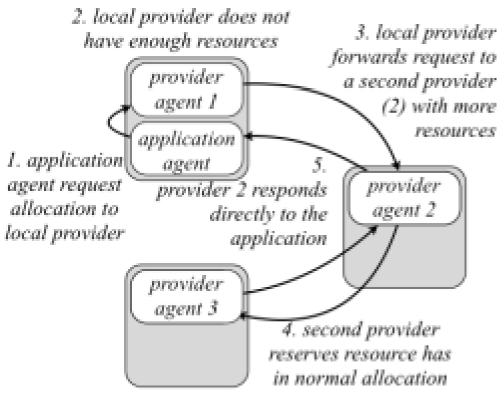
The arrival of an application request changes either the state of the short-list or the target metrics. If the application request is successful, it consumes a certain number of resources in the short-list. If the provider fails to satisfy the request, it must update the target metrics according to the request. The provider uses the services it was unable to allocate (see Section 3.3) to create a list of minimum requirements. From this point on, the messages requesting a new provider will carry a list of minimum resource requirements. These messages are routed randomly through the network just like in the initial setup process. However, peer providers only respond if they have the amount of resources specified in the message available. Every time the provider receives a response of a provider that can satisfy an element in the list of minimum requirements that element is removed from the list. The process should continue until the target metrics are fulfilled: the short-list has the targeted size and there are enough providers that meet the minimum requirements.

During the lifetime of the provider, the short-list may grow in size. The provider may want to reduce the size of the short-list to maintain the allocation process simple. The provider may need to limit number of peer providers it can process. This limitation can be imposed based on the amount of computing resources currently available at the provider. A provider hosting many demanding applications may be less willing to spend resources allocating other applications. Thus, reducing the size of the short-list also limits the applications that it can allocate. On the other hand, an idle provider agent can use some extra resources creating larger short-lists to be able to answer positively to most future resource allocation requests.

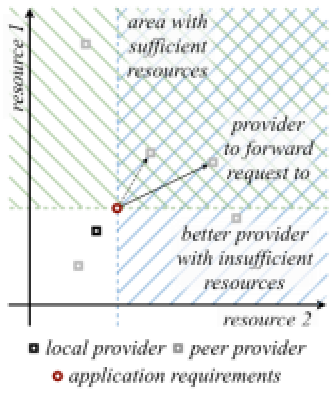
A successful allocation depends not only on the size of the short-list but also on its quality in terms of resources per host. We may include several heuristics to improve the chance of successful allocation. The simplest heuristic specifies a target average resource per host. In our current implementation, the provider agent uses the last allocated applications as a reference. Peers that repeatedly do not have enough resources to allocate any of the services of those applications are removed from the short-list. We go through an example of how allocation request events can impact short-list size in Section 3.6.5.

3.5. Resource-Based Routing

Whenever a provider cannot meet an allocation request, it has a decision to make: either extend the short-list to meet application's requirements or forward the request to a different provider. The provider will only forward the request if it believes some other provider is in better state to fulfill the request. To help peer providers decide, each provider advertises the total resources it has in their short-list. Thus, whenever a publish message is sent, a provider includes not only its available services, but also the total amount of resources of all the providers in their short-list. Figure 4a depicts the request forwarding process. When a provider receives an allocation request, it checks its own as well as its peers' resources. If it does have enough resources it then searches for a peer with more resources in the short-list. If it finds one, it forwards the request to the chosen peer provider. The peer provider will allocate the application normally (if it has enough resources) and respond directly to the application agent.


2. local provider does not have enough resources
provider agent 1
application agent
1. application agent request allocation to local provider
3. local provider forwards request to a second provider (2) with more resources
5. provider 2 responds directly to the application
provider agent 2
provider agent 3
4. second provider reserves resource has in normal allocation

(a)


resource 1
area with sufficient resources
provider to forward request to
better provider with insufficient resources
resource 2
local provider
peer provider
application requirements

(b)

Figure 12 – Resource-based allocation request routing.

To decide where to route failed requests, the provider uses the resource-based routing mechanism depicted in Figure 4b. In the figure, we represent the resource vectors of each provider and the resource requirements vector of the application in a resource space. For the example in the figure, we use a 2-dimension resource space. However, the procedure is the same for n-dimensional resource space. During the allocation process, the provider computes a vector corresponding to the sum of all resources demanded by the application. The vector is represented in the figure by the circle point. The providers who do not have enough resources are filtered out (single shaded areas in the figure). The provider then computes the root mean square distance between the application resource requirements vector and each resource vector of the remaining peers. The provider will consider the peer presenting the largest distance. The goal is to increase the chance of that provider being able to serve the request.

3.6. Experimental Results

In this section, we demonstrate (1) the protocols efficacy, (2) that the protocols can scale to increasingly complex conditions (number of entities, concurrent requests, diversity of applications), and (3) they can be as effective as an optimized system given the dynamic nature of the DC computing environment.

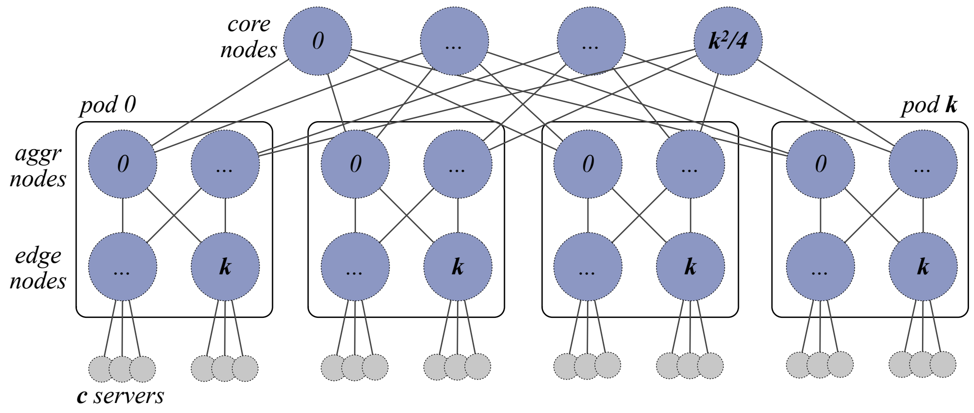


Figure 13 – Fat-Tree topology with custom cluster size.

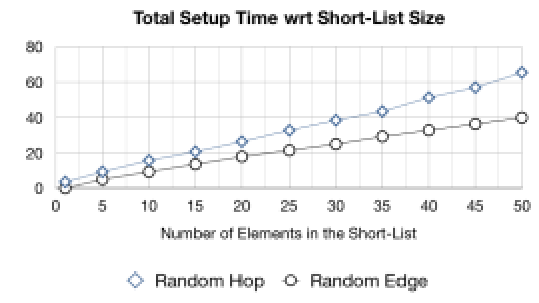
The DC architecture used throughout the experiment corresponds to a typical Google DC architecture with 40 servers connect by 1Gbps top-of-the-rack switch [30], [31]. We connect these clusters using a Fat-Tree topology [32] as it is a generally accepted topology for these infrastructures [30], [31]. In the Google cluster scenario, the 40 servers in the cluster have 1Gbps connections to the rack switch and this edge switch has 8 1Gbps connections to the top level. This corresponds to an oversubscription of 5 and a Fat-Tree parameter $k=16$. This infrastructure accommodates a total of 5120 servers. During the experiments, we vary the parameter of the fat-tree, k, and the cluster size, c. Figure 13 provides an example of this topology (the number of switches in the figure corresponds to $k=4$). We analyze the performance of our approach through simulation using the Network Simulation 3 (NS-3) [33]. We implemented the protocols as described in this paper and the topologies referred above.

3.6.1. Short-List: Peer Provider Search

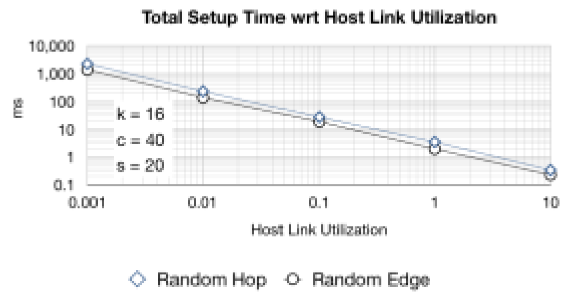
This section studies the performance of the protocol that constructs a short-list of peer providers. This process is applied during initial short-list setup and during short-list maintenance. As referred in Section 3.2 we propose two modes: random hop and random edge. In the first option, each switch randomly chooses an output interface to forward the probe packet to. In the latter, each edge node randomly chooses another edge node to receive the probe packet and forward it to one of its directly connected servers. The choice of an edge switched is constrained by the TTL value in the packet. The provider also guarantees that utilization of the link to the edge nodes does not surpass a certain value. As base parameters for this experiment, we use a topology factor, $k=16$, cluster size $c=40$, maximum link utilization (using 1Gbps links), u=0.1% and a short-list target size of 20. In the following subsections, we study the time to create the short-list (t_{SL}) when we vary each one of these parameters. We start every provider simultaneously and measure the time it takes for all providers to build their short-lists.

3.6.1.1. Short-List Size

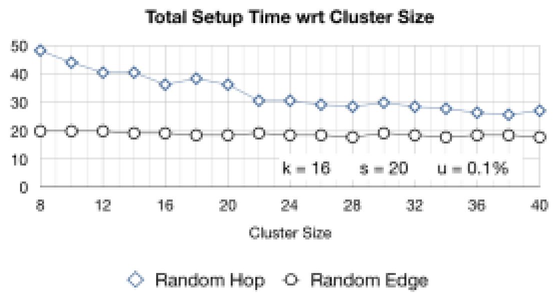
We vary the target short-list size from 1 to 50 peer providers. The results are displayed in Figure 14a. We can see that the time to setup the short-list varies linearly with the short-list size when we use the random hop mode. This is an interesting result as in a highly redundant topology, like the Fat-Tree, cycles could affect the protocols performance due to spurious packet as soon as we increased the TTL. However, this is compensated by the higher probability of find a new provider (not already contacted) to the short-list. The performance significantly improves when we use the random edge mode. In this case, there are no spurious packets. We can observe that it exhibits a logarithmic variation. This happens because the protocol spends some time searching for close nodes. But, as TTL increases, the probability of finding a new peer is higher, leading to performance improvement.



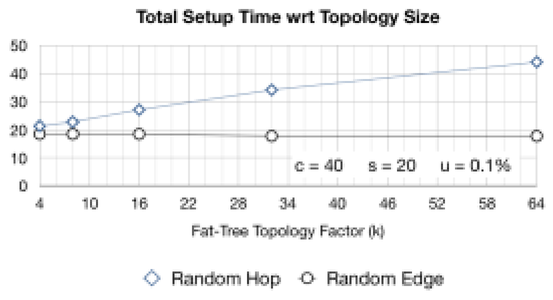
(a)



(b)



(c)



(d)

Figure 14 – Time to create the short-lists when varying list size, maximum link utilization and topology size.

The short-list size impacts the size of an application (in number of VMs) that a single provider can allocate. Thus, we can see that a provider can quickly find a large number of providers to be able to support large applications.

3.6.1.2. Host Link Utilization

Each provider determines the rate at which it sends probe packets to limit the overhead in the access link. We study the time it takes to build the short-list in both modes as we increase the utilization of the link from 0.001% to 10%. The results are depicted in Figure 14b. We can observe that even using only 0.1% of the channel the protocol takes only 30ms to terminate in the random hop mode. The variation of $log(t_{SL})$ is linear with respect to $log(u)$. This is a logical result because, as we increase utilization, we only increase the rate of packets transmitted. The whole protocol logic and, therefore, the probability of finding new peers, remains the same.

3.6.1.3. Cluster Size

In this experiment, we vary the cluster size from 8 to 40 servers. The results are depicted in Figure 14c. The cluster size impacts the protocol in how much the probe's TTL has to be increased for a provider agent to be able to find enough peers to fill the short-list. For cluster sizes smaller than the short-list target size (s=20), the cluster is not enough for providers to complete their lists. Thus, the protocol lasts until the TTL is increased to search for peer providers farther away. We can observe a significant drop when $c=20$. The variation is always decreasing because larger clusters decrease the probability of selecting repeated providers. In random edge mode, the decrease in allocation time is very small due to the absence of spurious probes when TTL is increased.

3.6.1.4. Topology Size

We now vary the value of the Fat-Tree factor, k, from 4 to 64. The results are depicted in Figure 14d. Increasing the topology size means increasing the number of clusters and number of links in the DC. This significantly impacts the performance in the random hop mode as it increases the number of probes lost due to expiration of TTL. In random edge mode, the topology size does not impact the performance.

3.6.2. Application Allocation Time: Isolated Requests

In this section, we evaluate the time it takes to reserve resources in the network as the load increases. We randomly generate applications and deploy application agents in the network in a random node. We then measure the time between the initial allocation request and the final response. All servers are setup with the same number of resources. Each server offers four different resources (e.g. CPU, Memory, Disk IO, Network IO). For each resource, we define three different profiles:

- *Low usage*: resource usage randomly chosen between 0 and 5% of the providers' capacity;

- *Medium usage*: resource usage randomly chosen between 5% and 15% of the providers' capacity;
- *High usage*: resource usage randomly chosen between 15 and 50% of the providers' capacity.

We run three sets of experiments. In each set we use a different application generator:

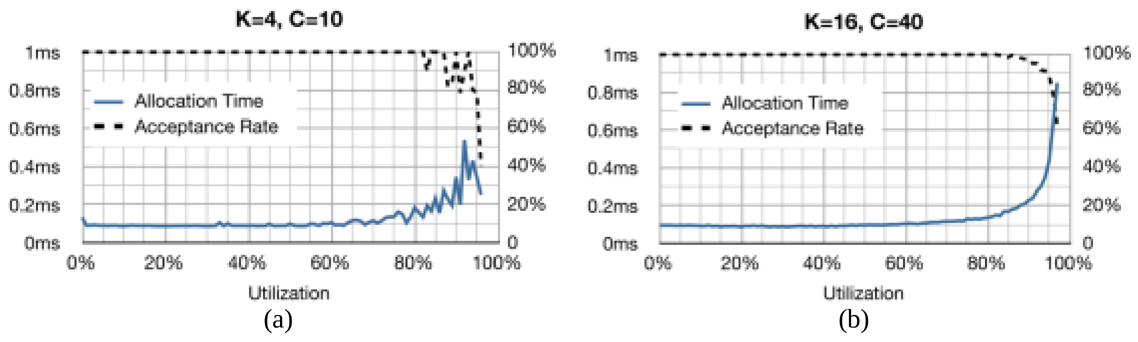
- *Uniform service generator*: creates applications composed by a single service with medium usage profile for all resources;
- *Heterogeneous service generator*: creates applications composed by a single service, where the profile for each resource is randomly selected according to a uniform distribution.
- *Heterogeneous application generator*: randomly chooses the number of services that compose an application between 1 and 10 according to a uniform distribution. It uses the heterogeneous service generator to create each service.

We run each set for three different topology sizes (*k ={4, 16, 64}* and *c={10, 40, 40}*. These topologies correspond to DCs with 160, 5120 and 81920 servers. We use an initial short-list size equal to 5 and a maximum utilization of 0.1%. We generate applications one second apart. In NS-3 we typically simulate the speed of network protocols. However, we use the Linux function clock_gettime to estimate the time it takes to process each packet and embed that time in the simulation.

In this experiment, we measure the time it takes to return a successful allocation and the acceptance rate (success requests vs. failed) as the utilization of the DC increase. The utilization of the DC corresponds to the utilization of the dominant resource. The dominant resource is the resource that has more demand by all the applications allocated combined. For instance, if CPU is the dominant resource, 90% DC utilization means that, on average, 90% of each CPU is reserved.

3.6.2.1. Uniform Services

The results obtained with the uniform services generator are displayed in Figure 15. We observe that the performance is constant up to high levels of occupation of the DC (above 90%). This means that the nodes adapt to the service size required by maintaining in their short-list peers with enough resources to host the application. We also observe that this behavior does not change even as we increase the size of the DC. The time to allocate one service is very similar across all topology sizes (roughly 1ms). For high values of DC utilization, it is hard for the nodes to find new peers with enough available resources for their short-list. This leads to a sudden increase in the time to allocation services. We can use this stability information to ensure the termination of the protocol as described in Section 3.3.4. Application agents can also set timeouts for their requests of few tenths of milliseconds. If the request is not satisfied in that time, there is a high probability that it will not be successful.



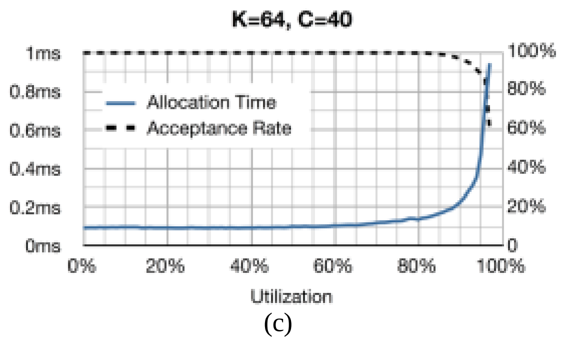


Figure 15 – Time to allocate resources as utilization of the DC increases (uniform service).

3.6.2.2. Heterogeneous Services

We repeat the experiment but now generating heterogeneous services. Some of the services can demand a high value of a single resource. The results obtained for increasing topology size are depicted in Figure 16. We observe that allocation time is stable, for all topologies, until the utilization of the DC approaches 80%. Remember that, according to our setup, a random service can demand up to 50% of the resources available on 1 server. This is the point when servers have fewer resources and their short-lists need to be rebuilt to allocate new servers. However, resources start being scarcer all over the DC. Thus, requests with a high demand of some resource have a higher impact on the short-list maintenance process. This leads to an increase in the time to allocate as the utilization of the DC increases. Once again, we see almost no difference in the performance of the protocol as we increase topology size. The main difference occurs for the smaller topology. This is due to the ratio of the size of services (in number of resources required), when compared with the resources available in the DC.

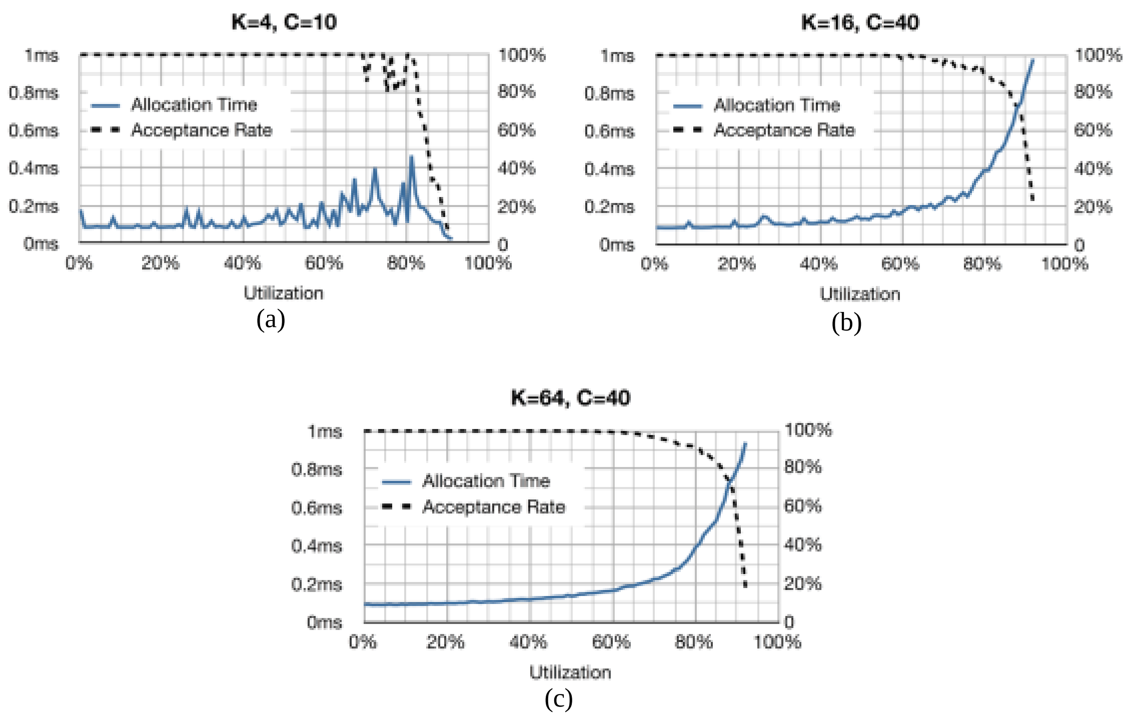
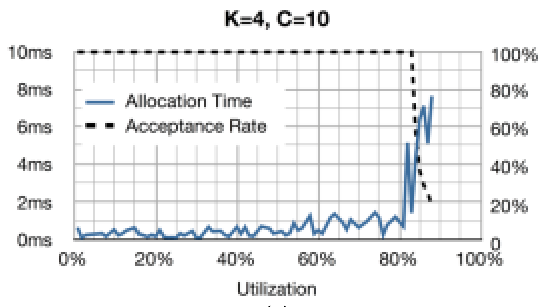


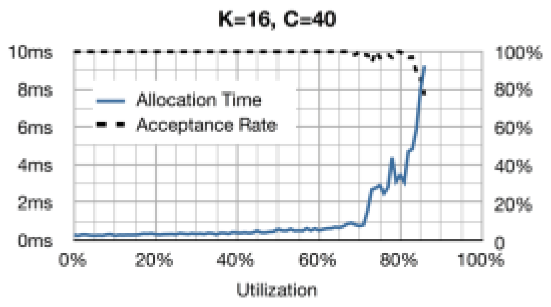
Figure 16 – Time to allocate resources as utilization of the DC increases (heterogeneous service generator).

3.6.2.3. Heterogeneous Applications

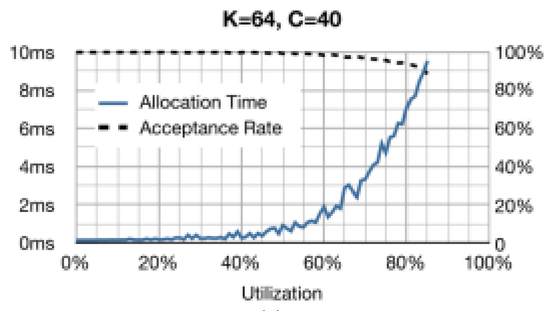
In this phase of the experiment we generate full applications with up to 10 random services. In Figure 17, we represent the time to allocate the whole application and the acceptance rate. The increase on the time to allocate when compared with the previous experiment comes from the application size. Previously we were allocating only one service. The behavior of the protocol is quite similar to the observed in the previous experiment. The only difference is that there is a steeper increase when the utilization surpasses 80%. This is due to the nature of the applications we used in this experiment: large number of services with possible high resource demand.



(a)



(b)



(c)

Figure 17 – Time to allocate resources as utilization of the DC increases (heterogeneous application generator).

3.6.2.4. Environment Dependence

As we increase the size of the network and the number of servers, the allocation time remains constant for the same utilization and application type. We therefore conclude that the behavior of the protocol is independent of the number of nodes in the topology.

To understand how the protocol behaves with respect to the nature of applications we compute a normalized value of the allocation time. For each request, we divide the allocation time by the number of resources requested. We plot the curves obtained for the *k=16* topology in Figure 18. As we can see the curves for all application types are very similar in shape and value. From this, we can conclude that the allocation time depends mostly on the total number of resources that the application requires. It is, however, independent on the complexity of the application.

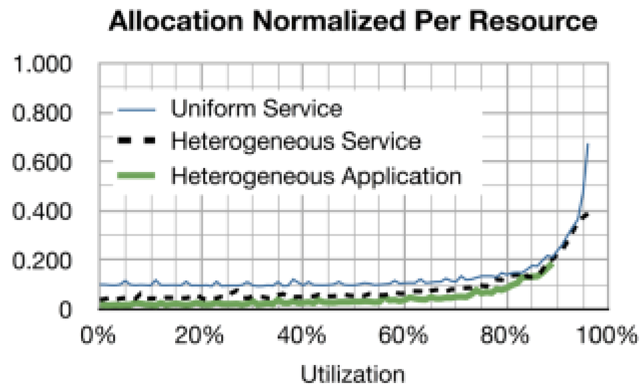


Figure 18 – Allocation Time Normalized per Resource.

Finally, we observe that utilization of the DC is the factor that impacts most the behavior of the protocol. Naturally, as resources become scarce, the provider agents take more time to find the resources necessary to allocate the application. This is also related to the short-list size and quality. If a provider agent has enough resources, it can increase the size of the short-list to be able to respond immediately to more complex resource requests. The time to allocate only starts increasing steeply when the DC approaches its full capacity.

3.6.3. Application Allocation Time: Concurrent Requests

We now study the performance when the DC is faced with a burst of allocation requests. We run this experiment on the standard Google topology (*k=16*). We generate *n* simultaneous allocation requests and measure the time it takes for each application to receive a response. The results obtained are presented in Figure 19. We plot the curves, for average allocation time, the percentage of reject requests and the final utilization of the DC. We can see that the allocation time is approximately constant up to higher values of utilization of the DC, increasing slightly as we approach 100%. We observe also that the average allocation value is very similar to the values we obtained when allocating a single service at a time in the previous experiment. The system also does not reject requests until the utilization is very close to the limit (slightly above 60000 services). We can conclude that the time to allocate a service does not depend on the rate of arrival of requests either.

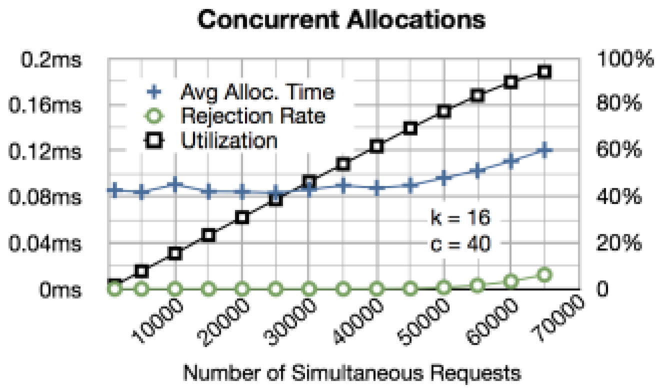


Figure 19 – Allocation Time for Concurrent Allocation Requests.

3.6.4. Efficiency

From the experiments presented earlier in Figures 15, 16 and 17, also demonstrate how efficiently our distributed solution uses the DC. For uniform services, the protocol only starts refusing services when the full DC utilization is close to 90%. For more complex applications requiring more resources, the acceptance rate starts dropping as utilization reaches 80%. These values are highly dependent on the resource demand models defined earlier. Thus, this simple greedy approach to allocation is able to achieve a high utilization. Furthermore, this value of utilization is achieved without targeting a particular resource type. The allocation protocol does not focus on optimizing any specific resource.

For a static allocation, the DC utilization efficiency depends mostly on the ratio between the resources available on each server and the resources consumption profile. However, the dynamic characteristics of the DC can have a huge impact on the overall utilization efficiency. When the workload of applications varies frequently, the ability to quickly respond to bursts of requests allows quick re-adaptation of allocation to the new conditions.

3.6.5. Short-List Dynamics

In Figure 20 we plot a sample of the evolution of the short-list size at one provider agent. Short-lists are increased in reaction to failed allocation requests. In case of a successful request, the provider can also trigger short-list maintenance to increase the number of resources available and prevent future failed requests. We marked six relevant events in the evolution of the short-list. In this experiment, the initial short-list size was zero. This means the provider does not have peer providers, and could only immediately allocate its own resources. At event *a*, the provider agent receives a new allocation request for which it does not have enough resources. The agent starts increasing the size of the short-list to satisfy the request. At event *b*, the provider is able to allocate the application. To prepare for future allocation requests, the provider uses the last application as a reference to rebuild the short-list. Thus, the provider agent checks if it can allocate an application like the previous one. In this case, the response is negative and the agent keeps searching for peers to add until the target size is achieved. At event *c*, a new allocation request arrives. Every time a request is processed, the provider keeps track of peers that repeatedly did not have sufficient resources to allocate services. The provider deletes these peers from the list after a given number of failed attempts. At moment *d* the short-list is again complete. Finally, at moments *e* and *f*, after successful requests, the provider agent removes some more elements with insufficient available resources.

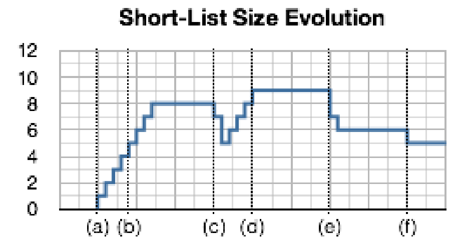


Figure 20 – Evolution of the size of one provider's short-list during one execution.

3.7. Assumptions

Assumption 2. We have the ability to define and control network protocols.

Our de-centralized proposal for searching for new peers assumes that the provider can deploy customized network protocols like TRILL in the DC or broadcast peer probe packets. This might not be possible due to security concerns. Broadcasts are often blocked, and the network might have constraints on the protocols that can be used.

Thus, the protocol to search for new peers might need to adapted to network that is not flat or open. One possible solution is ad-hoc hierarchical groups. The first provider agent in a local network could take a coordinator role. It would be responsible for handling peer probe requests in the local network. Other provider agents would first try to find a local coordinator and join the group. The local coordinators could then form a network between them using a centralized coordinator. This would allow provider agents to find peers in different networks.

3.8. Summary

Resource allocation is an elementary function of any cloud management framework and, in particular, LAMA. Application agents request allocation of virtual instances to local provider agents. The provider agent searches available resources using a peer-to-peer network. This partial view of resources available in the DC is dynamically adjusted as resources are requested and reserved.

In this Chapter, we study the efficiency of LAMA's distributed resource allocation mechanism. We demonstrate its performance and adaptability by simulating various scenarios in a large data center. We observed that resource allocations maintain good performance for large number of concurrent requests. A natural question that arises when using multi-agent systems is if the agents view (in this case, local resource database) is able to adapt fast enough to answer environment demands (in this case, new allocations requests). Our experiments demonstrate that performance remains stable up to high levels of data center utilization.

In the next Chapter, we compare the performance of the entire provisioning workflow of our distributed system with a state-of-the-art centralized framework (OpenStack).

Chapter 4

Distributed Provisioning

4.1. Introduction

The single most important task of cloud management framework is the ability to provision virtual instances. Cloud frameworks should guarantee that this function does not constitute a bottleneck during periods of high volume of provisioning requests. As cloud services become more popular, the volume of requests tends to increase and congestion situations will occur more frequently. In this chapter, we analyze LAMA's provisioning mechanism. We compare its performance to OpenStack, the most popular open-source cloud management framework. OpenStack has a centralized logic. Resources are managed in a centralized database. OpenStack comprises several independent services that manage a specific function for all hosts. We aim to understand how provisioning performance might vary in diverse scenarios and how it differs between our distributed (LAMA) and a centralized (OpenStack) approach.

We start by describing the workflows for instance provisioning in both frameworks: OpenStack and LAMA. We then create series of three scenarios with gradual increase of concurrency between provisioning requests. Finally, we provide an analysis of the differences between the two frameworks.

4.2. OpenStack Workflow

The OpenStack framework has multiple centralized services. The following components are involved in resource provisioning:

- **Dashboard**: Web interface that allows cloud administrators to manage the framework and users to manage their applications. It can be used to trigger provisioning requests;
- **Keystone**: OpenStack authentication service. All operation requests need to be accompanied by an authorization token;
- **Nova**: Centralized controller and scheduler. It manages the workflow of provisioning requests and handles resource search and allocation for instance provisioning;
- **Compute**: Distributed agent running on each server that can host virtual instances. It handles local provisioning of new virtual instances;
- **Glance**: Service responsible for storing and serving the virtual images;
- **Quantum**: Network service that manages network for all applications deployed in the OpenStack framework.

OpenStack support multiple mechanisms for communication between its services. It can use Advanced Message Queuing Protocol (AMQP) frameworks like RabbitMQ or peer-to-peer technologies like ZeroMQ.

To provision a new virtual instance, a user needs to upload the instance's image file. The following description of the workflow (depicted in Figure 21) assumes that the image was uploaded to the Glance service:

(1) When a user makes a request for a new instance, the dashboard requests a token for the operation from the authentication server. Alternatively, the requests can be made to the Nova controller API. In this case, the user must obtain a token in a previous step;

(2) The dashboard sends a new provision request to nova, which includes the authorization token and the instance type (which defines resource requirements); the nova scheduler needs to validate token with the Keystone authentication service;

(3) The Nova scheduler searches the database for the requested resources and selects the server that will host the instance. It then sends a reservation to the compute node, who in turn confirms if it can host the instance. The compute node handles the remainder of the workflow;

(4) The compute node requests details about the instance from the Nova scheduler;

(5) The compute node requests the network configuration info from the Quantum service. The Quantum server validates the compute node's token, configures the instance in the DNS and DHCP service and

returns the relevant configuration information to the compute node. The compute node also has to configure the local virtual network interfaces;

(6) The compute node requests the image to be copied from the image service. After validating the compute node's token, the Glance image server starts transferring of the image.

(7) Once the image is copied, the compute node initiates the instance. It notifies the nova scheduler, once the instance is ready.

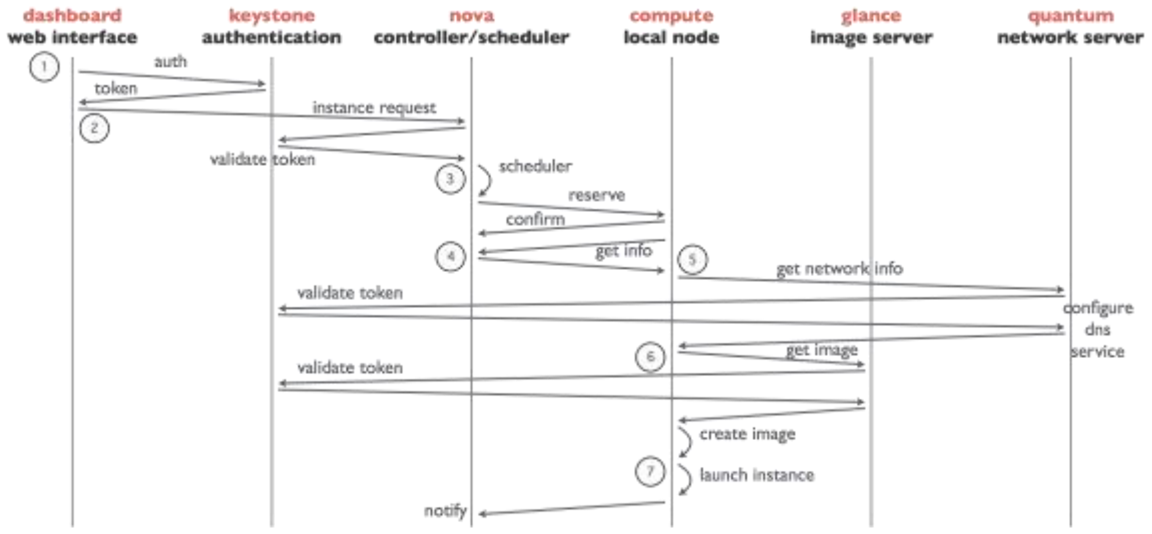


Figure 21 – OpenStack Provisioning Workflow.

We have seen that the OpenStack framework is based on task-specific services. Each service performs the same task for all the applications and host in the data center. We consider this a centralized paradigm, despite the fact that each service can be placed in a different server and communicate with others like a distributed framework.

4.3. LAMA Workflow

LAMA utilizes a MAS approach where autonomous agents can handle multiple tasks. The workflow for provisioning a new instance is depicted in Figure 22. There are two main steps. The initial step creates an application. This step is only required if the instance is not being added to an existing application. The only centralized entity is the dispatcher agent that selects a provider agent and forwards the initial application request.

The workflow for creating a new app proceeds as follows:

(1) A user creates a new application by sending a new request to the dispatcher agent. This request can be done through the LAMA's web interface or through the command line API;

(2) The dispatcher agent registers the app and selects a provider agent, using random round robin, to forward the app request to. It will continue search until finding a provider agent that is able to launch a new application agent.

(3) Once the provider confirms acceptance of the application request, the dispatcher sends the provider information to the cloud user;

(4) The provider configures a VPN for the new application: configures the Open vSwitch bridges, launches an SDN controller and a DNS server process.

(5) Once the network is configured, the provider agent launches the app agent process. Upon initialization, the app agent authenticates to the provider agent, thus confirming a successful start-up.

Once the app agent is deployed, launching an instance takes the following steps:

(1) The user requests a new instance by sending a new request to the provider agent.

(2) The provider agent forwards the request to the app agent. The app agent updates the application model and sends a new allocation request to the provider agent.

(3) The provider agent follows the LAMA's resource allocation mechanism (see Chapter 3): it searches the resources in the local database. If it selected resources are not local, it needs to remotely reserve the resources at the peer provider agent. In the end, it notifies the app agent that the resources were allocated.

(4) The app agent sends a request to the selected provider to start the instance.

(5) The host provider agent retrieves a copy of the instance's image. The image may have been stored in a different peer provider. As we have seen before, image services are allocated in the same manner as other instances. They however, only take disk space resources.

(6) A final notification is sent to the in-memory database at the local provider. Users can subscribe these notifications directly from provider agents.

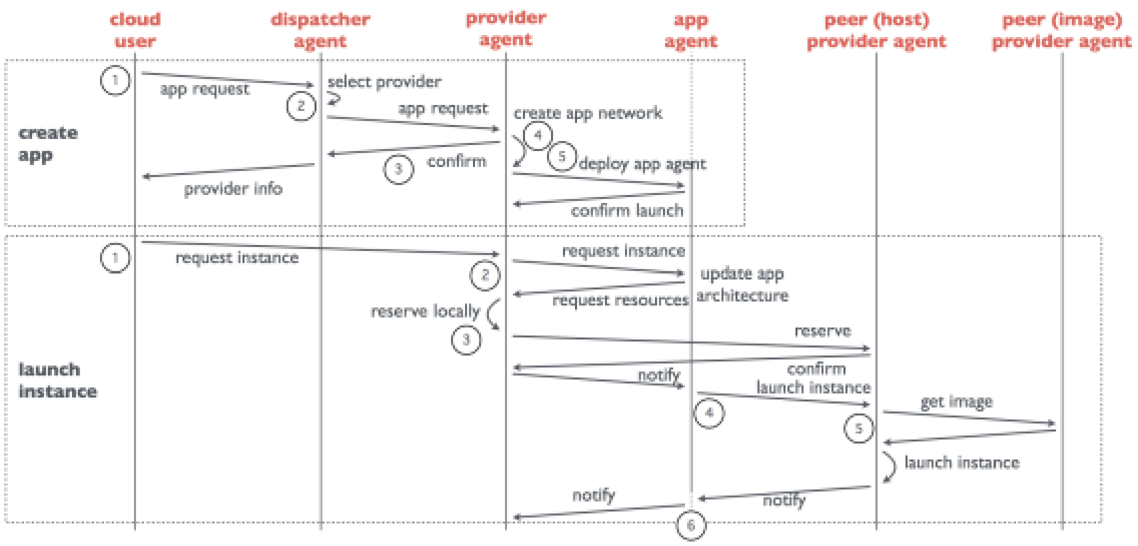


Figure 22 – LAMA Provisioning Workflow.

4.4. Experiments

We compare the performance of OpenStack and LAMA for sequential requests (light load) and parallel requests (high load). For that purpose, we deployed the two frameworks in our cluster. We use the LAMA Experiments module to automatically schedule the launch of all instances.

4.4.1. Testbed and Experiments Setup

Table III presents the characteristics of testbed cluster used in the experiments. To deploy OpenStack centralized services (Keystone, Nova, Glance and Quantum) we use one *type-4* server with 8 logical cores and 8GB of RAM. We use the same node for LAMA's dispatcher agent. The dispatcher agent is a light process, that does not require as many resources. However, we do not deploy a provider agent in the same server as the dispatcher agent. The total number of servers available to host instances is the same for both frameworks. This setup leaves a total of 22 servers (82 logical cores and 172GB of RAM) to serve as hosts.

Table III – Characteristics of the Testbed Cluster Servers.

SERVER TYPE	NO. SERVERS	CPU	CPU MAX (MHZ)	PHY. CORES	LOG. CORES	RAM (GB)
1	9	Intel(R) Core(TM)2 Duo CPU E8500	3167	2	2	4
2	2	AMD Athlon(tm) 64 X2 Dual Core Processor	3200	2	2	4
3	7	AMD FX(tm)-4100 Quad-Core Processor	3600	2	4	16
4	5	Intel(R) Core(TM) i7-3770 CPU	3400	4	8	8
Total	23			56	90	180

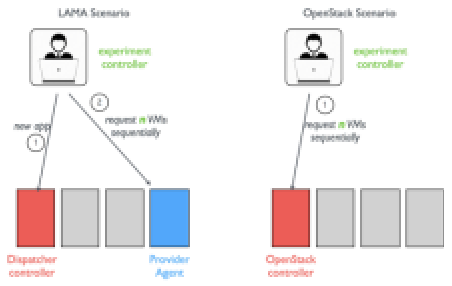
In these experiments, we use an image of the CirrOS operating system. This is a tiny Linux distribution of only 12MB, designed precisely for cloud tests. For each instance we reserve 1 logical core and 512GB of RAM, which corresponds to the 'm1.tiny' instance flavor in OpenStack. This allows us to deploy up to 82 instances in the cluster without over-allocation.

4.4.2. Experiment Scenarios

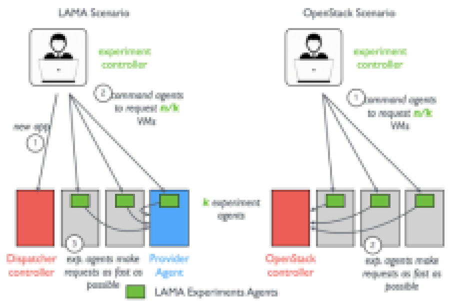
OpenStack allows users to create projects, which are typically used by cloud users to group related instances. Virtual instances are created for a given project. Applications in LAMA work like OpenStack projects. When running OpenStack scenarios, all projects are created before the start of the experiment. LAMA applications, however, are created during the experiment. Creating LAMA applications beforehand would provide an unfair advantage to LAMA, as it would remove the only centralized agent in the system, the dispatcher agent, from the workflow. Using LAMA Experiments, we create three scenarios:

- **Single User, Sequential Requests** (Figure 23a): We create a sequence of n requests from a single node. In the OpenStack scenario, all requests are directed to the nova API. In LAMA scenario, we request a new app to the dispatcher agent. The requests for new instances are sent to the provider agent where the app agent was created.
  - This scenario does not overload frameworks with many concurrent requests. It serves as a baseline to compare with the other two scenarios.
- **Single User, Parallel Requests** (Figure 23b): Using the distributed LAMA Experiments agents we create a total of n requests as fast as possible. Maximum number of simultaneous requests is n/k where k is the number of experiment agents deployed. In the OpenStack scenario, all requests are directed to the nova API. In the LAMA scenario, for each request, experiment agents try to create the application. If it already exists, the dispatcher returns the provider agent address that contains the app agent. The request to create a new instance is then directed to the app agent.
  - Compared with the sequential requests scenario, this experiment increases the contention between requests for the same application.
- **Multiple Users, Parallel Requests** (Figure 23c): Using the distributed LAMA Experiments agents we create a total of n requests as fast as possible. Each request creates an instance for a different project (in OpenStack) or application (in LAMA). Just like in previous scenario, OpenStack requests are directed to the centralized API. For LAMA, in this scenario, all requests will create a new application and add a new instance. However, now instance creation requests are directed to multiple provider agents as app agents are distributed among all servers.

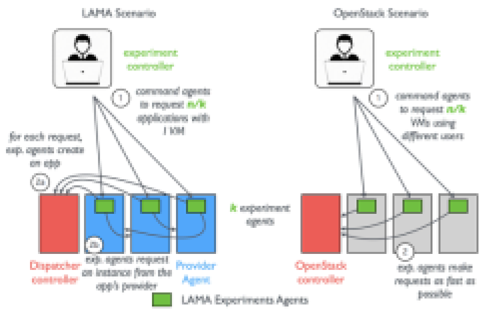
- This scenario creates multiple projects/applications with a single instance. For LAMA, each application will be managed by a different agent in a different node.



(a) Single User, Sequential Requests



(b) Single User, Parallel Requests


LAMA Scenario
experiment controller
command agents to request n/k applications with 1 VM
for each request, exp. agents create an app
Dispatcher controller
exp. agents request an instance from the app's provider
Provider Agent
k experiment agents
OpenStack Scenario
experiment controller
command agents to request n/k VMs using different users
OpenStack controller
exp. agents make requests as fast as possible
LAMA Experiments Agents

(c) Multiple Users, Parallel Requests

Figure 23 – Provisioning Experiment Scenarios

4.4.3. Experiment Metrics

For each experiment, we measure:

- **API Call time:** Measures the time that the framework takes to respond to the API call to create a new instance.
- **Activation Time**: Measures the total time since the request is made until the instance is active.
- **Number of errors**: Counts the number of instances that could not be allocated either due to framework error or lack of available resources.

4.5. Results

For each experiment, we do several runs for different values of n = 1, 5, 10, 20, 30, …, 90 instances. The cluster can only allocate 82 instances, thus this maximum value of n will saturate the cluster, to the point that some instances have to be rejected. In the parallel experiments, we launch one experiment agent per node (except for the dispatcher or OpenStack server node), thus $k = 22$.

4.5.1. Single-User Sequential Requests

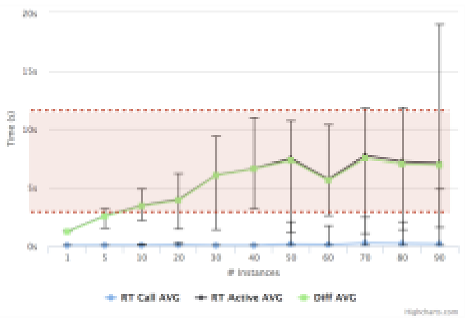
Figures 24 to 26 show the results obtained for the single-user with sequential requests scenario. Figure 24 (a and b) shows the evolution of average API call and activation times per instance (error bars represent minimum and maximum) as we increase n for each of the two platforms. Figure 24c shows a comparison of the activation times for both frameworks. Figure 25 show the number of instances that the frameworks were unable to provision (as expected, in this scenario 8 instances where not provisioned when requesting 90 instances, as they do not fit in the cluster).

For LAMA, we observe an initial increase in the average time it takes to provision an instance. This is due to concurrency when copying the image to the host where the instance is deployed. The image is small and is copied quickly. However, because the API call times are very short ($< 1s$), we still face concurrent copies. We can see that when we run experiments with more instances, the time to activate instances stabilizes (minimum and maximum values between 3 and 12 seconds).

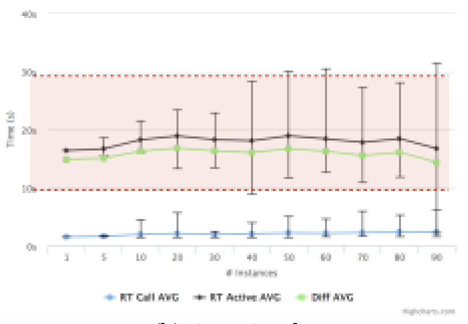
OpenStack presents a more stable average behavior. However, the API call takes, on average, around 3 seconds to respond (as a production framework, the call makes more operations). However, the variation is wider. The time it takes to activate the instance varies between 10 and 30 seconds. The API call time ends up spreading the image copy processes over time, thus reducing contention.

We can have a better understanding of the relative progress of the experiment using Figure 26. It represents an inverted timeline of the experiment with 50 instances. For each instance, we represent three points. In the x axis, each instance is represented by its ordinal number in the sequence of requests. The y coordinates are: the instant of time when the instance was requested, the instance of time when the user got the response of the API call, and the instant of time when the instance become active.

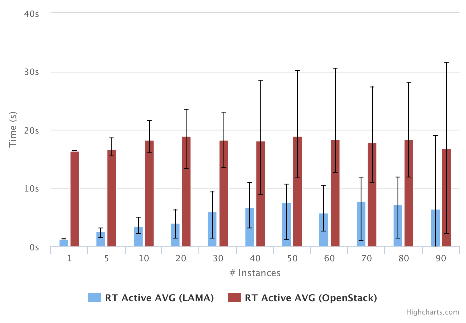
We can see that, given its faster API call, the experiment using LAMA makes all requests within 2 seconds. The instances are activated in parallel, leading to a total running time of about 13 seconds. On the other hand, it takes 80 seconds to make all the requests using OpenStack and 100 seconds to activate all instances.



(a) LAMA



(b) OpenStack



(c) Time to activate instances: OpenStack vs. LAMA

Figure 24 – Single-User, Sequential Requests: Evolution of call and activation times for LAMA and OpenStack. The green line represents the difference between active and call times.

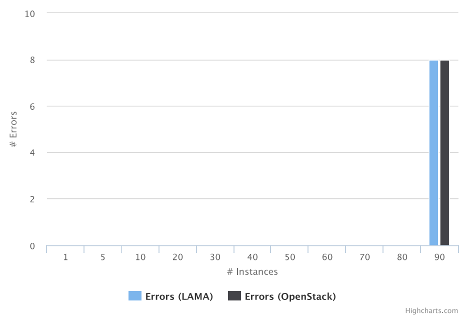


Figure 25 – Single-User, Sequential Requests: Failed Requests: OpenStack vs. LAMA.

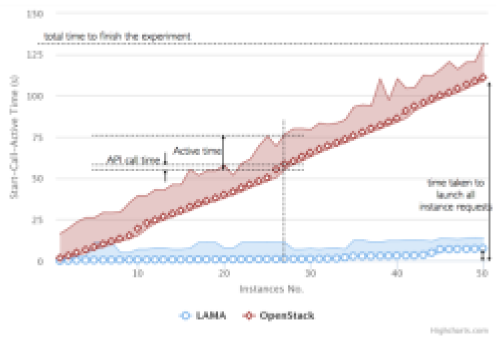
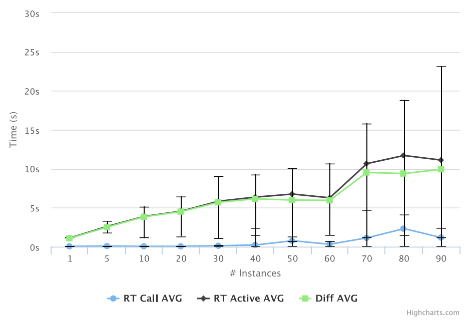


Figure 26 – Single-User, Sequential Requests: Timeline of the' experiment with n=50. For each instance, we plot: (x, y1, y2, y3) = (instance ordinal number, request timestamp, API call response timestamp, Activation timestamp).

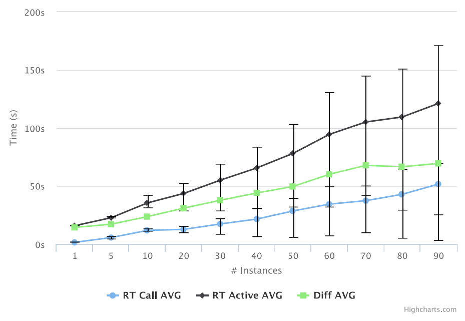
4.5.2. Single-User Parallel Requests

Figures 27 to 29 show the results obtained for the single-user with parallel requests scenario. Figure 27a shows that LAMA maintains exhibits similar performance for experiments with up to 60 concurrent instances. The increase on average activation time for a higher number of instances is due to local contention while creating the instances. From the timeline, in Figure 29, we can see that the total time LAMA takes to start all requests is similar. Thus, the contention while copying the image is similar to the previous experiment.

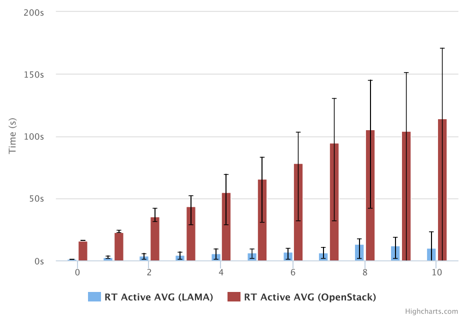
From Figure 27b, we can see a deterioration of performance for OpenStack for both the API call and the time to activate the instances. The average time to allocate each instance seems to grow linearly with the number of simultaneous requests. From the timeline, in Figure 29, we see that the API call time quickly increases to approximately 30 seconds when we create a bulk of requests. For more than 80 simultaneous requests some timeouts occur (see Figure 28). These errors where mostly due to timeouts on the API call, as the nova API became congested. We can see the difference in overall performance between the two frameworks in Figure 27c.



(a) LAMA



(b) OpenStack



(c) Time to activate instances: OpenStack vs. LAMA

Figure 27 – Single-User, Parallel Requests: Evolution of call and activation times for LAMA and OpenStack per isntance. The green line represents the difference between active and call times.

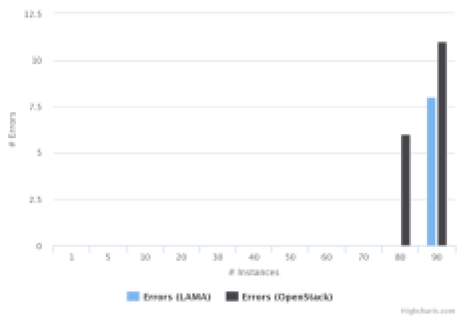


Figure 28 – Single-User, Parallel Requests: Failed Requests: OpenStack vs. LAMA.

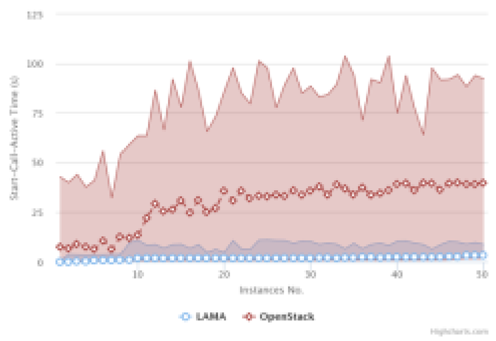
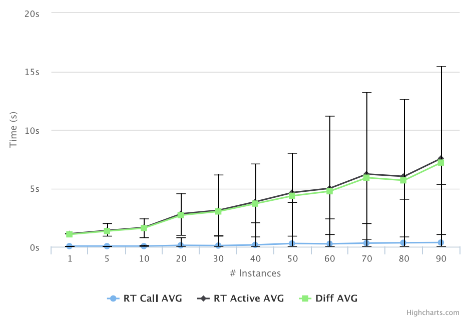


Figure 29 – Single-User, Parallel Requests: Timeline of the experiment with n=50. For each instance, we plot: (x, y1, y2, y3) = (instance ordinal number, request timestamp, API call response timestamp, Activation timestamp).

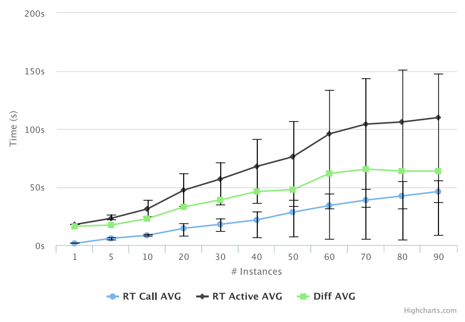
4.5.3. Multiple-User Parallel Requests

Figures 30 to 32 show the results obtained for the multiple-user with parallel requests scenario. For the OpenStack framework, the performance degrades slightly. API call response time and activation times are similar (Figure 30b) to the previous single-user scenario. The experiment timeline for *n*=50 is also similar (Figure 32). However, we notice a clear increase in the number of failed requests (Figure 31). The main reason for the increase in the number of errors was congestion in OpenStack's network server. The Quantum server handles the configuration of the networks for all projects. Using different projects, increased the number of network the server had to manage.

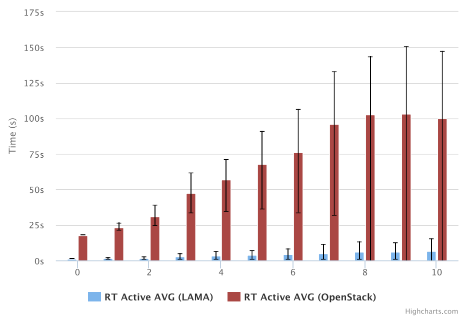
For LAMA, we observed that creating multiple applications had a positive impact in the API call time (Figure 30a) and in the activation times. The API call time improved in particular when we requested 50 or more instances simultaneously. The average and variation of activation times were also reduced. Using multiple applications distributed the load among different app agents located in different servers (4/5 app agents per server). Nevertheless, LAMA provisioning still experiences some congestion in the hypervisor when multiple instances are provisioned at the same time in the same server.



(a) LAMA



(b) OpenStack



(c) Time to activate instances: OpenStack vs. LAMA

Figure 30 – Single-User, Parallel Requests: Evolution of call and activation times for LAMA and OpenStack. The green line represents the difference between active and call times.

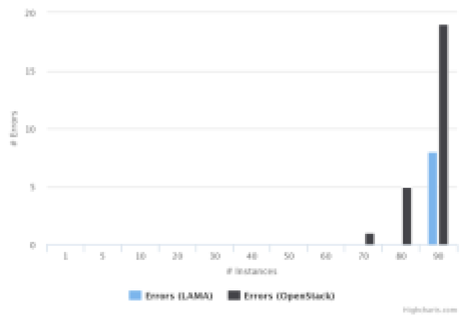


Figure 31 – Single-User, Parallel Requests: Failed Requests: OpenStack vs. LAMA.

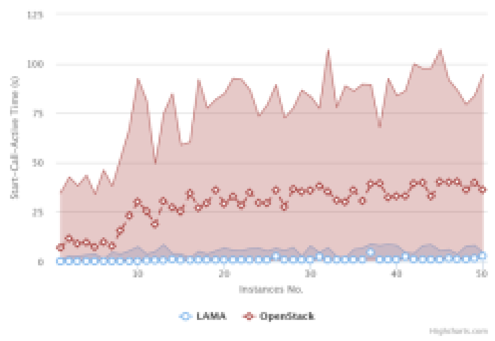


Figure 32 – Single-User, Parallel Requests: Timeline of the experiment with n=50. For each instance, we plot: (x, y1, y2, y3) = (instance ordinal number, request timestamp, API call response timestamp, Activation timestamp).

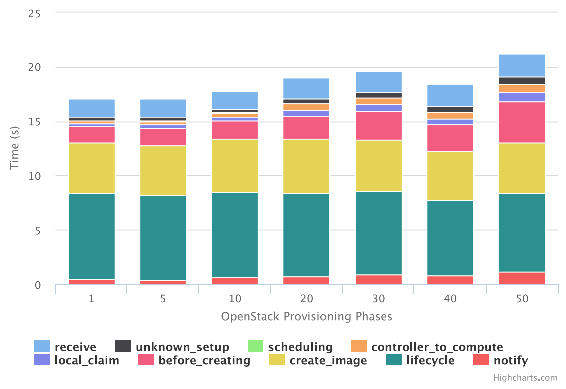
4.6. OpenStack Provisioning Analysis

OpenStack experiences a degradation in provisioning times as the load increases. However, we aim to understand, in detail, the causes for that change. It is especially relevant to understand if those causes are related with the centralized logic of OpenStack. We analyzed the OpenStack logs to track how much time is spent on different provisioning phases per instance.

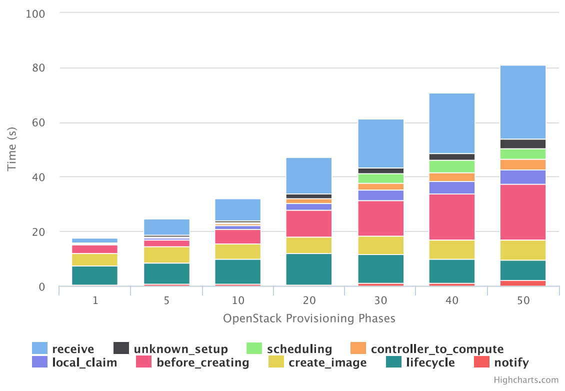
We split the time take by OpenStack provisioning workflow into the following phases:

- **Phase 'receive'**: Reception of the requests. Measured from the moment the request is made until it is received by the nova controller;
- **Phase 'unknown setup'**: Initial configuration of OpenStack to provision the instance (we were unable to identify the exact actions OpenStack performs). Measured from the moment the request is received by the nova controller until it is received by the nova scheduler;
- **Phase 'scheduling'**: Determine the compute node that will receive the instance. Measured from the moment the request is received by the scheduler until we see the selected host information;
- **Phase 'controller_to_compute'**: Transmission of the request from the Nova controller to the compute node. Measured from the moment the request is scheduled until the computer node acknowledges its reception;
- **Phase 'local_claim'**: The compute node reserves the local resources for the instance. Measured from the moment the request is received by the computed node until it announces that it has reserved the resources for the instance;

- **Phase 'before_creating'**: The compute nodes executes local configurations required by the instance. It includes configuring the local network to accommodate the instance. Measured from the moment the compute node announces the reserved resources until it actually creates the instance image.
- **Phase 'create_image'**: The compute node copies and creates the image. Measured from the moment the image is created until the compute node creates the instance in the hypervisor.
- **Phase 'lifecycle'**: The hypervisor creates the instance. Measured from the moment the compute node creates the instance until the instance is active;
- **Phase 'notify'**: The compute node notifies the nova controller that the instance is active. Measured from the moment the compute node announces locally that the instance is active until we receive a notification sent by the nova controller.



(a) Sequential (Single-user)



(b) Parallel (Multiple-user)

Figure 33 – Time spent in each provisioning phase per instance in OpenStack.

We plot the average time spent by each instance in each of the provisioning phases in Figure 33. Figure 33a displays the results for sequential request in a single-user scenario, while Figure 33b shows the results for parallel requests with multiple users. The statistics show that most phases (receive, unknown_setup, scheduling, controller_to_compute, local_claim, before_creating, and notify) perform progressively worse as we increase the load. This is due to the workload increase in the server we use to manage OpenStack. OpenStack can be scaled by adding more replicas for each of its services. This would naturally reduce the number of instances the cluster could receive as more servers were used for management.

Three phases standout by their either significant or unexpected increase in duration:

- The **receive** phase (blue columns in Figure 33, steps (1) and (2) in Figure 21): The API web server performance degrades significantly affecting the initial time to receive requests. With 50 simultaneous instances, this phase accounts for roughly 30% of the provisioning time. This could be addressed by scaling the web server horizontally (more servers) and by increasing the number of simultaneous threads.

- The **scheduling** phase (green columns in Figure 33, step (3) in Figure 21): The scheduling phase degrades when we request more than 30 instances concurrently. This happens due to the logic of the scheduling procedure. The nova scheduler searches for resources in the central database. After that it tries to reserve the resources in the compute node. The reservation is only completed once the compute node confirms that it can allocate the resources. When we increase the number of concurrent requests, resource collisions occur more frequently. In this case, the compute node rejects the reservation and the nova scheduler needs to search for new resources. A change on the logic of the provisioning mechanism could solve this issue. For instance, the scheduler could reserve the resources atomically in the central database or simply add some more randomness in the resource scheduling could help reduce collisions.
- The **before_creating** phase (pink columns in Figure 33): This phase degrades due primarily to network configuration. This procedure depends on the centralized network server. As we increase the number of concurrent requests, the workload in the network server increases. We also observed timeouts in the network server.

LAMA avoids most of these problems with its distributed architecture. The distribution of the resource search phase, where each provider agent has a partial database of the resources, reduces the likelihood of collisions. On the other hand, LAMA scales naturally, as all servers contain a provider agent capable of answering requests. This avoids the need of constantly worrying about scaling the management system. The dispatcher, the only centralized component, handles very simple low frequency tasks. Scaling OpenStack is possible, but cumbersome. An experiment made by the Ubuntu engineering team [34] has shown the ability to allocate 168000 instances in 640 servers. During the experiment, where they run into multiple scalability issues, they observe a significant degradation in the time to activate instances in the framework (30s to 90s). It also took more than 10 hours to deploy 100000 instances.

4.7. Assumptions

Assumption 3. Pre-load images to storage.

We consider the performance of the provisioning mechanism. In order to provision new instances, their images need to be uploaded to storage. This is true for both OpenStack and LAMA. OpenStack uses a central service to store images. LAMA processes images per application. Each app agent is responsible for determining where images for each service are to be deployed. However, there are several conditions we are unable to accurately emulate in our experiments to achieve meaningful results. First, the performance of the image uploading phase depends on the network bandwidth of the access path. Second, LAMA's distributed approach would benefit from multiple entry points to the DC infrastructure. Deployment of multiple entry points can be constrained by security policies. Finally, image storage services are normally deployed in storage area networks with better disk and network capabilities. Thus, the presented results do not include possible benefits from distributed images storage.

Assumption 4. No post-deployment configuration.

The metric 'activation time' measures the time it takes for an instance to become active. In order to be usable by the application, some instances require additional configuration. LAMA handles automatic post-configuration of virtual instances. OpenStack expects manual configuration by the users (most commercial frameworks include solution automatic instance configuration). This, final step of provisioning is not taken into account in these experiments. Thus, the time for an instance to become an active application instance is longer than the activation time considered.

4.8. Summary

In this Chapter, we present a comparison of the provisioning task for two frameworks: our distributed LAMA framework and the logically-centralized OpenStack. We are able to observe major differences in the performance of the two systems. LAMA maintains its performance as we increase the number of concurrent requests. Contention when deploying multiple instances in the same host, is the only type of congestion LAMA cannot avoid. OpenStack struggles as we increase the number of requests.

Chapter 5

Distributed and Integrated Monitoring and Diagnosis

This Chapter describes LAMA's monitoring sub-system. We demonstrate the benefits of distributed monitoring in the time to react from failures or events that might affect application performance. We also explore the advantages of our integrated approach diagnosing performance problems due to interference at the host.

5.1. LAMA Monitoring Sub-System

The monitoring sub-system has two main goals:

(1) Provide app agents with real-time data from its ecosystem (hosts, instances and network used by the application);

(2) Allow users to customize methods for detecting and reacting to events that affect application's performance.

The architecture of the monitoring modules is presented in Figure 34. Local resource monitoring is done using collectd [23], a plugin-based open source monitoring tool. A key-value database (using Redis [24]) provides a publish-subscribe mechanism that notifies agents of new metrics in real time. Redis serves as a proxy to all metrics. Application instances can also publish custom metrics to be sent to the monitor module of the app agent through a private network.

The user defines one or more monitoring strategies that determine metrics to subscribe, algorithms to apply and granularity of monitoring. If the instance is deployed in a different server, the provider agent is responsible for subscribing metrics on the remote server and publishing them to the local Redis instance. The provider can subscribe metrics of the host and the virtual instance. The subscribed metrics are defined by the monitoring strategy in the app agent. App agents only subscribe metrics from the local Redis database. This avoids redundancy of data with co-located agents subscribing the same metrics from remote servers. The monitor module in the app agent thus receives this metrics as soon as they are published. It then applies a detection algorithm defined in the monitoring strategy.

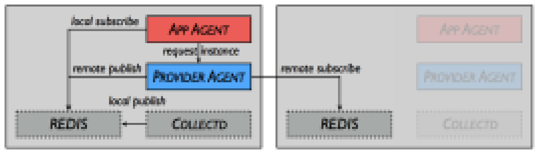


Figure 34 - LAMA Monitoring module.

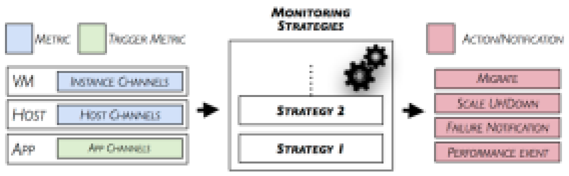


Figure 35 – Monitoring Strategies.

App agents use monitoring strategies to react to events in the application's ecosystem. Strategies subscribe metrics from the monitor module and generate actions to be performed by the app agent. Users can fully customize a strategy to be used by an app agent. A high-level overview of the execution of monitoring strategies is shown in Figure 35. The user can configure multiple strategies, each specific to a distinct function (e.g. failure detection, scaling or identification of other relevant events). There are three main components in strategy definition:

- **Inputs and Triggers**: Inputs are the metrics of the ecosystem used by the strategy (app events, instance, host or network metrics). The strategy can be run periodically, or triggered by new values of a metric chosen by the user. The user can also configure the granularity of monitoring;
- **Algorithm**: The user defines how the metrics are processed. It could be done by either user's custom code or using LAMA's existing constructs. LAMA provides basic data stream blocks such as sliding window averages, threshold comparison, rule-based processing and finite state machines;

- **Output Actions**: The strategy can trigger an output action or notifications. The actions could be scale up or down instances according to the workload, migration of instances or simply notifications to the app controller.

5.2. Related Work

It is difficult to obtain detailed information about commercial cloud providers and their infrastructures. However, we can observe some limitations in their monitoring services. Provider monitoring services vary in terms of granularity and number of metrics. Amazon [35] provides metrics with 5-minute periods. Users pay for an additional cost if they want data with 1-minute periods. Azure [36] metrics are retrieved every 3 minutes. Enabling verbose mode provides extra data but reduces monitoring granularity to 5-minute intervals. Google [37] metrics are collected in 1-minute intervals. These metrics can take 3 to 4 minutes to be available after being monitored. OpenStack [9] is the most popular open-source management platform. It has a monitoring module, Ceilometer [38], and a separated orchestration system, Heat, that handles orchestration of the application based on alarms from ceilometer. Overall, the centralized nature of OpenStack management leads to longer detection and response times.

There are numerous research on monitoring platforms for large infrastructures [39]–[41]. Nagios [42], Collectd [23] and PCP [43] are examples of low-level platforms that make use of local agents to extract a wide range of metrics. We use collectd agents to monitor information on the host infrastructure and instances. Most systems propose hierarchical solutions [38], [42], [44] to scale to large systems. They focus on low level metrics and use sampling to handle large volumes of data [45]. Wuhib et al. [46] propose a distributed P2P monitoring to handle resource usage overload. However, it is agnostic to application characteristics. Quality of Service (QoS) of cloud applications is critical specially in the presence of VM interferences. [47] [48] purpose QoS based management that focuses on scheduling of well-known workloads. They do not address changes in the environment with failures or workload dynamics. There is no work addressing the design of monitoring systems for end user management of their own applications.

Our work focuses on a fully distributed monitoring platform that allows customization of each application monitoring according to its requirements. The agent reacts to events that affect application's performance using information about the application state and architecture but also about the hosting infrastructure. It enables customized diagnostic of failures and faster response time. To our knowledge, our proposed platform is first to address such integrated environment. Its open and flexible design enables the use of other techniques proposed in the literature. Applicable techniques include dynamic granularity [25], adaptive [49], model-based analysis [50], stream and window based analysis [51] and machine learning and classification [48].

5.3. Experiments Setup

We analyze the advantages of having app agents integrated and distributed in LAMA. We demonstrate:

(1) how an integrated approach with access to the state of the app's ecosystem improves efficiency of failure detection;
(2) how our monitoring and diagnosing architecture can improve load distribution in the network;
(3) the impact of increasing granularity on failure detection time.

We implement and deploy LAMA in our datacenter with 23 interconnected heterogeneous servers. The available resources amount to a total of 90 logical cores and 180GB of RAM. Our experiments focus on online multi-tier transaction applications. We use the LAMA Experiments module to automatically deploy a online transaction benchmark, RUBBoS [52] that emulates a bulletin board. LAMA reads a logical specification of the application (see Figure 36), and provisions the application's services (apache and MySQL). LAMA also adds image service per application service and load-balancers before scalable services.

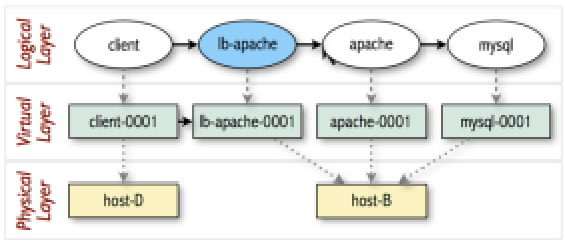
5.4. Case Study - Integrated Failure Detection

We explore the advantages of integrated monitoring on detection and diagnosis of performance degradations. RUBBoS has various web pages with different characteristics. We focus our analysis on the response time of two of those pages:

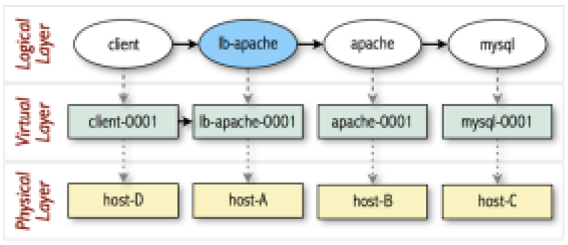
(1) **'StoriesOfTheDay'**: Web page that retrieves the stories to be displayed at the present moment;
(2) **'StoreComment'**: web page that stores a user comment about a story in the database. We emulate disk interferences at the host level.

5.4.1. Factors that Influence the Impact of Host Interference

We start by analyzing the factors that influence the impact of interferences at the host on application performance. We consider two setups: all three application instances deployed in the same host (Figure 36a) and each of the three instances in a different host (Figure 36b). The hardware characteristics of the servers used in the experiments are detailed in Table IV.



(a) Single host setup



(b) Multiple Host Setup.

Figure 36 – Application Model for RUBBoS.

Table IV – Hardware Characteristics of Server used in the Distributed Monitoring Experiments.

	PROCESSOR		RAM	DISK	
	PHY/LOG CORES	SPEED		TRANSFER RATE (SPEC)†	CACHE
	Units	*GHz*	*GB*	*Gb/s*	*MB*
Host A	2/4	3.6	16	6	16
Host B	2/4	2.5	16	6	16
Host C	2/4	3.3	16	6	16
Host D	4/8	2.88	8	6	64
Host E	4/8	3.4	8	6	64
Host F	4/8	3.4	8	3	32

† Transfer rate as specified by the disk vendor (typically proportional to but not matching physical speed).

We create hard disk interferences by running a disk IO-intensive process and evaluate its impact on application performance (we use the *btest* [53] tool). We add an additional feature to *btest* that allows the introduction of a small delay between each operation. This interval allows us to trigger different interference levels. We use two types of disk interferences:

- **Interference Type Read (IT-R)**: a *btest* process reading from a file on disk;
- **Interference Type Write (IT-W)**: a *btest* process writing to a file on disk.

The parameters that determine the intensity of the interference are:

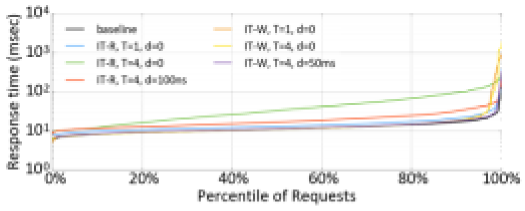
- **Delay (*d*)** between each read or write operation;
- **Number of threads (*t*)** used by the *btest* process to request operations in parallel.

We deploy the application in our cluster using LAMA and schedule a sequence of 2-minute-long interferences with different configurations. The interference configurations are:

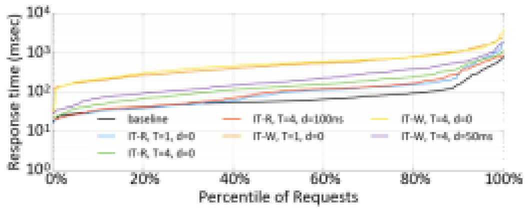
- Three IT-R interference with:
  (1) $d = 0$ and $T = 1$;
  (2) $d = 0$ and $T = 4$;
  (3) $d = 100ns$ and $T = 4$;
- Three IT-W interference with:
  (1) $d = 0$ and $T = 1$;
  (2) $d = 0$ and $T = 4$;
  (3) $d = 50ms$ and $T = 4$.

We compare the application performance during each interference with a *baseline* period without any interference.

The response times for the two reference pages during each period for the single-host setup are presented in Figure 37. The Figure shows the response time per percentile for each of the pages during each of the seven periods considered (baseline plus interferences).



(a) 'StoriesOfTheDay'



(b) 'StoreComment'

Figure 37 – Response time by percentile of requests in a single host scenario.

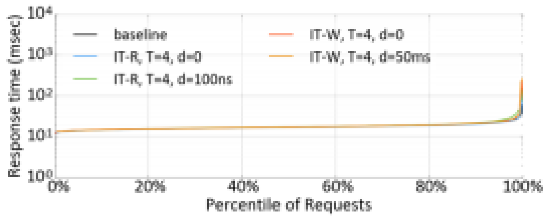
Table V contains the average, 80th and 95th percentiles for each of the experiments. This experiment's results are included in rows 0 (baseline) to 8 and variations are relative to the baseline case.

Table V – Numeric Results for the Diagnosis Experiments: Variation is computed with respect to the equivalent case (i.e. same interference) in the baseline scenario.

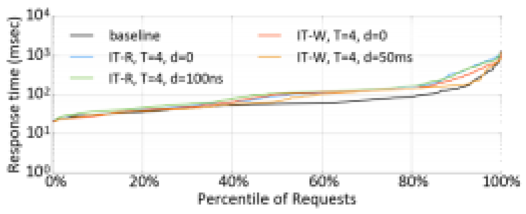
		Interference				Response Time (ms)											
						'StoriesOfTheDay'						'StoreComment'					
		Type	Host (Load Balancer, Apache or MySQL)	Delay	Number of Threads	Avg		P80		P95		Avg		P80		P95	
0	Single Host -					12		15		20		92		93		351	
1	Baseline	IT-R		0	1	16	+33%	18	+20%	27	+35%	154	+67%	154	+66%	675	+92%
2	(aka. Single Host	IT-R		0	4	44	+267%	66	+340%	119	+495%	193	+110%	244	+162%	671	+91%
3	4C/6Gbps/16M)	IT-R		100ns	4	20	+67%	25	+67%	41	+105%	143	+55%	165	+77%	545	+55%
4		IT-R		0	8	37	+208%	53	+253%	98	+390%	153	+66%	205	+120%	484	+38%
5		IT-W		0	1	21	+75%	15	0%	24	+20%	585	+536%	795	+755%	1528	+335%
6		IT-W		0	4	23	+92%	15	0%	24	+20%	612	+565%	775	+733%	1354	+286%
7		IT-W		50ms	4	13	+8%	15	0%	21	+5%	273	+197%	388	+317%	817	+133%
8		IT-W		0	8	14	+17%	16	+7%	24	+20%	523	+468%	683	+634%	1351	+285%
9	Multiple Host					17	+42%	20	+33%	23	+15%	86	-7%	86	-8%	266	-24%
10		IT-R	LB	0	4	17	+42%	19	+27%	22	+10%	146	+59%	157	+69%	594	+69%
11		IT-R	LB	100ns	4	18	+50%	20	+33%	25	+25%	150	+63%	164	+76%	571	+63%
12		IT-W	LB	0	4	18	+50%	20	+33%	23	+15%	126	+37%	141	+52%	438	+25%
13		IT-W	LB	50ms	4	19	+58%	20	+33%	23	+15%	104	+13%	141	+52%	229	-35%
14		IT-R	A	0	4	25	+108%	30	+100%	46	+130%	116	+26%	136	+46%	370	+5%
15		IT-R	A	100ns	4	20	+67%	23	+53%	31	+55%	108	+17%	131	+41%	413	+18%
16		IT-W	A	0	4	17	+42%	19	+27%	23	+15%	89	-3%	95	+2%	344	-2%
17		IT-W	A	50ms	4	17	+42%	19	+27%	23	+15%	116	+26%	97	+4%	480	+37%
18		IT-R	M	0	4	18	+50%	20	+33%	25	+25%	68	-26%	75	-19%	207	-41%
19		IT-R	M	100ns	4	18	+50%	20	+33%	24	+20%	97	+5%	96	+3%	369	+5%
20		IT-W	M	0	4	21	+75%	20	+33%	24	+20%	450	+389%	551	+492%	1023	+191%
21		IT-W	M	50ms	4	17	+42%	19	+27%	22	+10%	155	+68%	206	+122%	385	+10%
22	Single Host					6	-50%	7	-53%	8	-60%	72	-22%	69	-26%	245	-30%
23	8C/6Gbps/64M	IT-R		0	1	7	-42%	8	-47%	10	-50%	134	+46%	151	+62%	562	+60%
24		IT-R		0	4	11	-8%	13	-13%	19	-5%	111	+21%	135	+45%	322	-8%
25		IT-R		0	8	76	+533%	107	+613%	162	+710%	272	+196%	375	+303%	860	+145%
26		IT-W		0	1	8	-33%	7	-53%	9	-55%	488	+430%	669	+619%	1125	+221%
27		IT-W		0	4	7	-42%	7	-53%	9	-55%	502	+446%	644	+592%	1188	+238%
28		IT-W		0	8	7	-42%	7	-53%	9	-55%	469	+410%	589	+533%	1288	+267%
29	Single Host					9	-25%	7	-53%	9	-55%	431	+368%	535	+475%	1270	+262%
30	8C/3Gbps/32M	IT-R		0	1	8	-33%	8	-47%	11	-45%	464	+404%	619	+566%	1433	+308%
31		IT-R		0	4	12	0%	13	-13%	19	-5%	409	+345%	504	+442%	1116	+218%
32		IT-R		0	8	127	+958%	160	+967%	303	+1415%	650	+607%	787	+746%	1652	+371%
33		IT-W		0	1	17	+42%	7	-53%	10	-50%	1159	+1160%	1640	+1663%	2853	+713%
34		IT-W		0	4	31	+158%	7	-53%	10	-50%	1218	+1224%	1715	+1744%	2764	+687%
35		IT-W		0	8	11	-8%	7	-53%	9	-55%	1156	+1157%	1586	+1605%	2557	+628%

We observe differences in the impact of the interferences on each of the two pages. ‘StoriesOfTheDay’ page is mostly affected by multithreaded IT-R. The delay parameter has a proportional impact on the response time. We see an increase of approximately 60% for the IT-R interference with the delay and 2.5 to 5 times in the response time with four threads. On the other hand, ‘StoreComment’ page is affected by all interferences. A single-threaded IT-W is enough to overload the disk capacity to write. IT-R interferences increase the response time by 100% while IT-W interferences raise mean response time by 500%. IT-R interferences overload the CPU and the processor spends most of the time executing processes in the user and system levels. This causes disk read intensive processes to perform worse. IT-W interferences cause an increase in disk IO wait. This can cause problems to any process that needs to access the disk like, for the ‘StoreComment’ page, the database writing a comment. It affects mostly the write requests. Read requests are often optimized using cache mechanisms that can reduce disk accesses. This demonstrates that the impact of interference in the host depends heavily on the **application logic**.

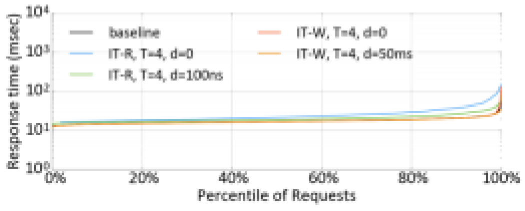
We repeat the experiment using the multiple host setup. We deploy each of the application instances to a different host. We run multi-threaded IT-R and IT-W interferences on each of the physical hosts. Figure 38 shows the response time in percentile during each interference per host and page. The numerical results are shown in rows 9-21 of Table V. The variation is computed with respect to the single-host experiment with the same interference. Naturally, we observe a general improvement to the response time compared with that of the single host scenario. Each interference only affects one service at a time. The ‘StoriesOfTheDay’ is only significantly affected by the IT-R interference with no delay at the Apache host. The response time increases by 100% compared to the single host baseline. The ‘StoreComment’ page is mostly affected by IT-W interference in the MySQL host. This is a natural consequence, as MySQL is the service that does most of app’s access to the disk. A new comment always has to be written to the database. IT-R interferences are mostly significant in the host of the load-balancer service. The impact is almost half of the impact seen in the single-host scenario. Thus, in complex multi-tier applications not all **component types** are sensitive to a given interference.



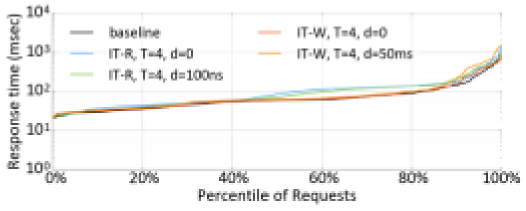
(a) Load Balancer / ‘StoriesOfTheDay’



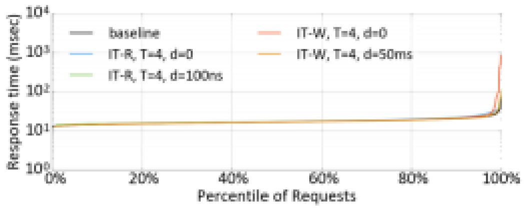
(b) Load Balancer / ‘StoreComment’



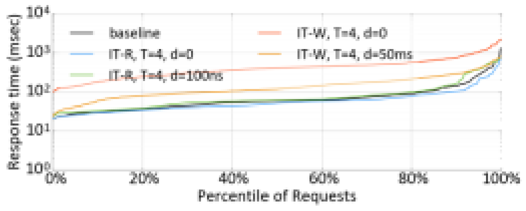
(c) Apache / ‘StoriesOfTheDay’



(d) Apache / ‘StoreComment’



(e) MySQL / ‘StoriesOfTheDay’

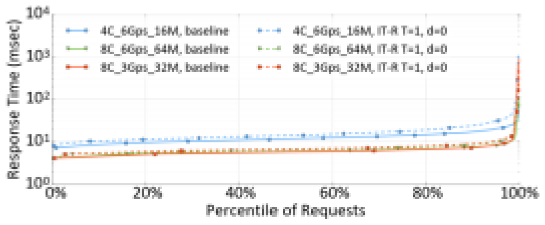


(f) MySQL / ‘StoreComment’

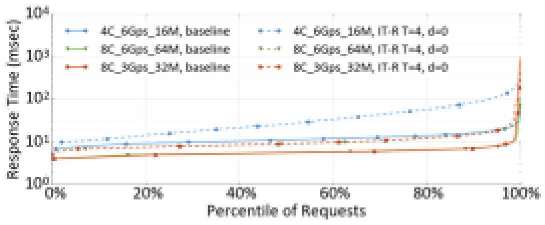
Figure 38 – Response time by percentile of requests in a multiple host scenario (host with interference/page).

We now analyze the impact on the response time when the application is deployed in hosts with heterogeneous hardware characteristics. We study deployments of the application in three different hosts. We use the single-host setup in Figure 36a, using hosts B, E and F. Hosts E and F have twice the number of cores and different disk performance (Table IV). Host E’s disk has the same speed and better cache than host B. Host F has a slower disk and slightly better cache than host B.

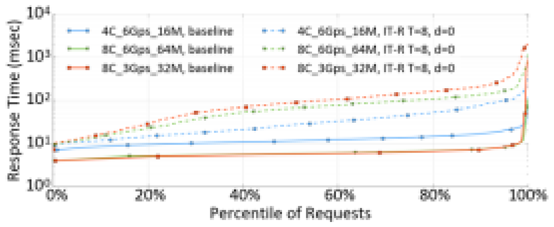
In Figure 39, we represent the response time for the page ‘StoriesOfTheDay’ when we trigger IT-R interferences using different number of threads (rows 1-4, 23-25 and 30-32 of Table V). The performance of the two hosts with 8 cores (E and F) is very similar except when we use 8 threads for the interference. The app in the host F (slower disk) has response time two times higher than host E. CPU overload is typically the main cause for performance degradation in the ‘StoriesOfTheDay’ page. However, extreme interferences (8-threads) can move the bottleneck to Disk IO.



(a) T=1



(b) T=4



(c) T=8

Figure 39 – Response time by percentile of requests in a single host scenario for different hosts: Page ‘StoriesOfTheDay’, IT-R.

Consider the impact of Disk Write interference using a single thread in the ‘StoreComment’ page displayed in Figure 40 (rows 5-8, 26-28 and 33-35 of Table V). Hosts with better disk perform better. Naturally, host E still outperforms host B as it has higher CPU speed. However, the resulting response time is not significantly different (ranges between 5 and 26%). Thus, the impact of the interference is also highly dependent on the host’s **hardware characteristics**.

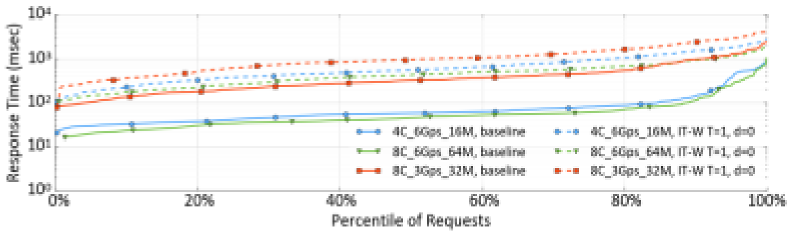


Figure 40 – Response time by percentile of requests in a single host scenario for different hosts: Page ‘StoriesOfTheDay’, IT-R, T=1.

We demonstrated that the impact of interferences depends on factors like the nature of application, type of affected components and host’s hardware characteristics. The true impact of the interference can only be assessed by observing the entire app’s ecosystem. We designed LAMA to provide app agents with all the tools and metrics to implement customized and effective diagnosis algorithms.

5.4.2. Custom Detection and Diagnosis

This section describes an application of our tool to improve effectiveness of detection and diagnosis. As seen in Section Table V we can define monitoring strategies to customize detection of events and diagnosis. We will use this feature to build customized decision trees to diagnose problems in the RUBBoS benchmark. As we have seen we need to define inputs and trigger metrics, algorithm and output actions.

To define the components of the decision tree, we start by analyzing application resource usage under different levels of IT-R and IT-W interferences. We deploy RUBBoS and generate twelve different interferences with different values of delay.

Figure 41 displays resource usage at the host, while Figure 43 display the response time of the two reference pages. These interferences are marked with lighter vertical bands.

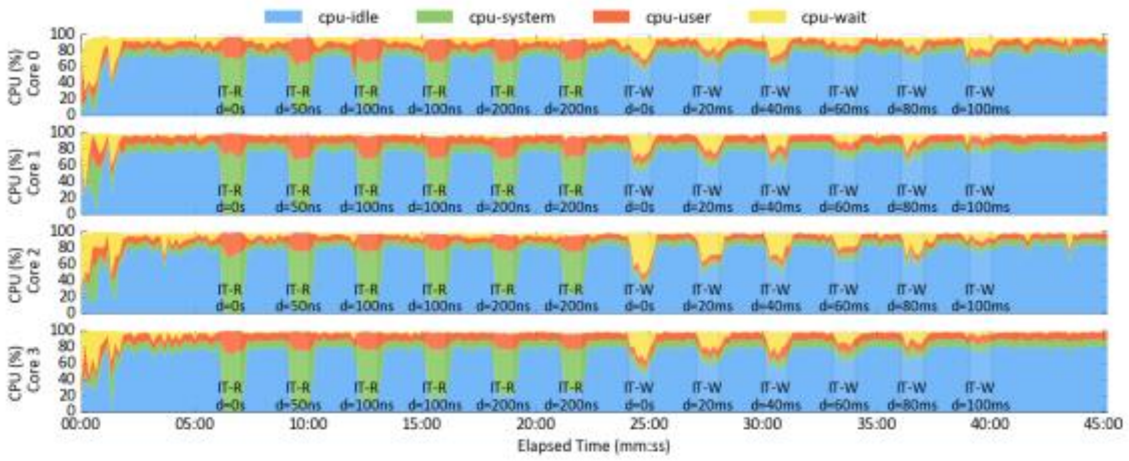
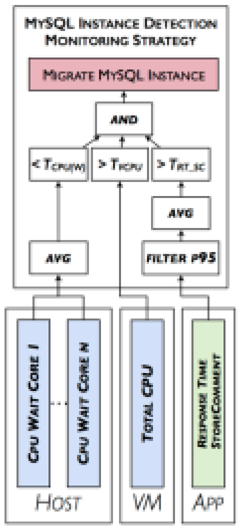
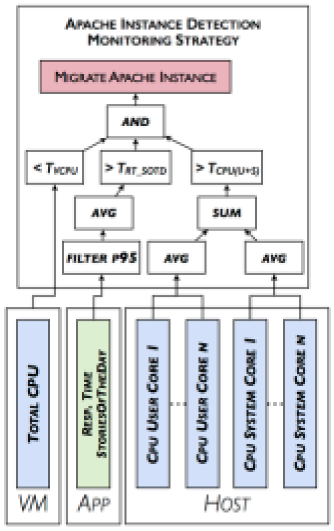


Figure 41 – Performance diagnosis: CPU usage detail by core. Bands indicate each interference period with varying delay.



(a) Single host setup



(b) Multiple Host Setup.

Figure 42 – Decision strategies for migrating MySQL and Apache Instances due to host interference.

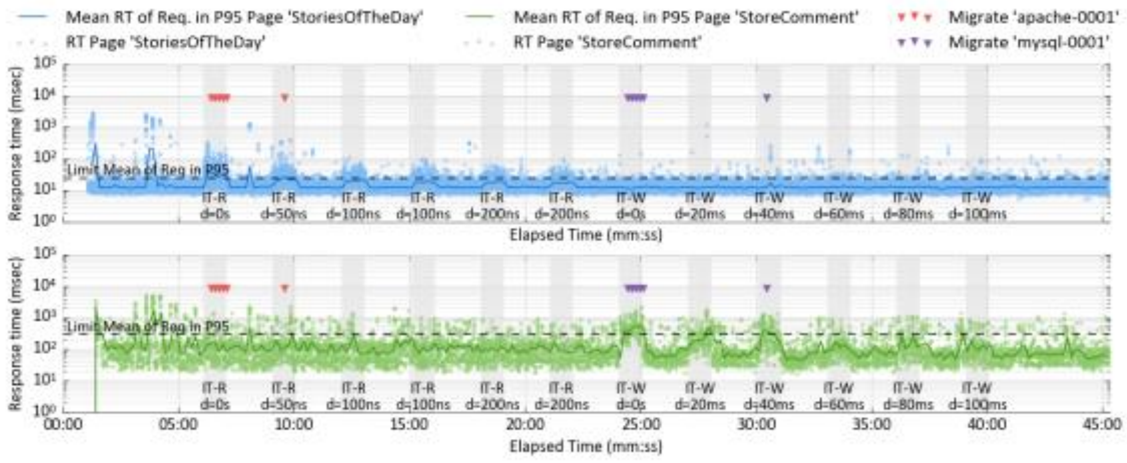


Figure 43 – Performance diagnosis: Response time and Action. Bands indicate each interference period with varying delay.

We observe:

(1) IT-R interferences impact the time the CPU spends in user and system tasks while IT-W impacts the time spent waiting for disk operations. Additionally, we observed during the experiment that resource usage for the virtual instances remains stable;

(2) From the response time chart, we can see the impact that these metrics have on the response time of each of the two reference pages. Just as we had seen in previous experiments, IT-R interferences do not

significantly affect the performance of the 'StoreComment' page while IT-W interferences do not significantly affect the performance of the 'StoriesOfTheDay' page;

(3) We had seen from Figure 38 that the 'StoriesOfTheDay' page is mostly affected by IT-R interferences on the Apache server and 'StoreComment' is mostly affect by IT-W interferences in the MySQL server;

(4) Response time presents isolated peaks that can affect the effectiveness of the diagnosis. To smooth this metric, we define an aggregation statistic that takes the average of the response times of requests in the 95th-percentile (represented as solid lines in Figure 43).

Using these observations, we can now design the three components of our monitoring strategy. We define two decision trees (shown in Figure 42): one to diagnose problems in a server hosting a MySQL instance that affect the 'StoreComment' page and the second to diagnose problems in a server hosting an Apache instance that affect the 'StoriesOfTheDay' page.

The MySQL instance decision tree suggests migrating the MySQL instance when response time of 'StoreComment' is high (aggregate response time higher than T_{RT_SC}), host CPU wait is high (above threshold $T_{CPU(Wait)}$) and the instance resource usage is low (below threshold, T_{VCPU}).

The Apache instance decision tree suggests migrating the Apache instance when response time of 'StoriesOfTheDay' is high (aggregate response time higher than T_{RT_SOTD}), host CPU wait is high (above threshold $T_{CPU(U+S)}$) and the instance resource usage is low (below threshold, T_{VCPU}).

Diagnosis is triggered by the arrival of new values of response time. The results of the diagnosis tree are displayed in Figure 43. The triangles indicate when the algorithm signals that instances should be migrated due to interference in the host, while the dashed black lines represent the response time thresholds. We used the following parameters: $T_{VCPU} = 70\%$, $T_{CPU(Wait)} = 20\%$, $T_{RT\_SC} = 300ms$, $T_{CPU(U+S)} = 70\%$ and $T_{RT\_SOFD} > 25ms$.

The response time statistic and threshold values are highly dependent on user goals for the application's performance. This is an example of how to design an effective diagnosis algorithm by leveraging information from the entire application ecosystem (hosts, instances and application).

5.5. Overhead vs. Granularity for Specific Detection Strategy

One of the advantages of using distributed architectures is reduced network overhead. In centralized solutions, all monitoring data is aggregated to a central system, which creates a network bottleneck. To assess the overhead reduction, we deploy as many applications and instances as possible in our datacenter cluster. Each application has four services (client, load balancer, web server and database). Each service is composed by a single instance. The monitoring granularity is set at 10 seconds between each data point for all servers. We deploy 5 applications (with a total of 20 instances) at a time and measure the average overhead for 5 minutes before deploying a new batch of applications. For a centralized system, we deploy all our agents in the same server, so that all monitoring data is sent to the same host. In LAMA, app agents are assigned to a server in a random round-robin fashion. For each host, we collect detailed CPU consumption per core, memory usage, disk usage for all local disks, and network traffic statistics per interface. For each instance, we collect total CPU usage per core, memory, disk and network usage.

Figure 44 shows the average received traffic at the bottleneck link (i.e. the link with most traffic in the datacenter). The overhead for the centralized system grows linearly with the number of applications or instances deployed in the cluster. The overhead of the distributed system reaches a plateau that depends on the number of applications per host. Ensuring that app agents are distributed evenly in the cluster ensures optimal overhead distribution. As we are operating a small datacenter cluster, the total traffic is not significant. This can quickly increase in large datacenters, with hundreds of thousands of servers. Applications might need more metrics or finer granularity. Another advantage of the LAMA's multi-agent framework is that it scales organically with the datacenter. A centralized system for processing monitoring data needs to be maintained and scaled individually.

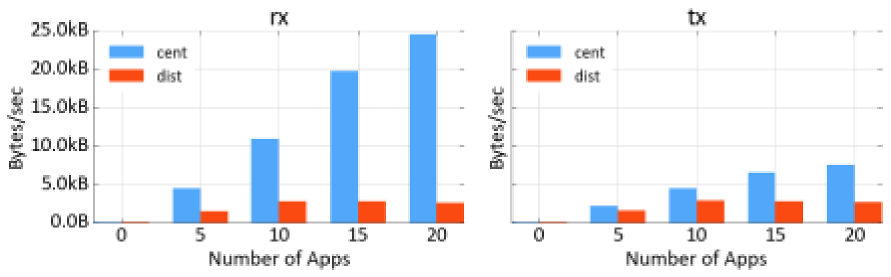
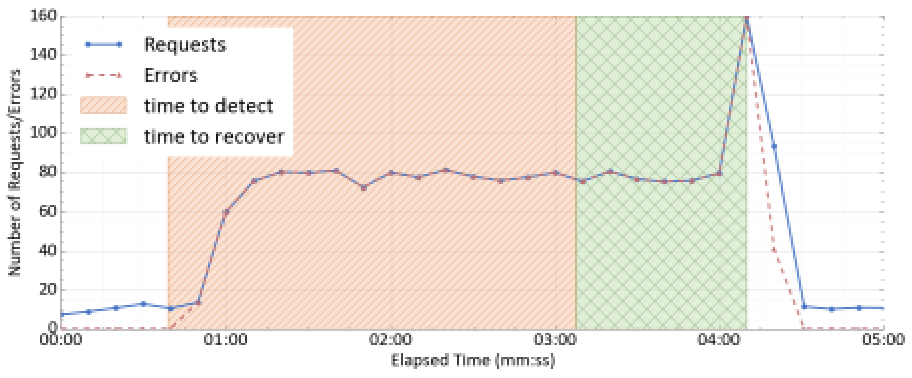


Figure 44 – Monitoring traffic overhead at the bottleneck: centralized vs. distributed setup, period=10sec.

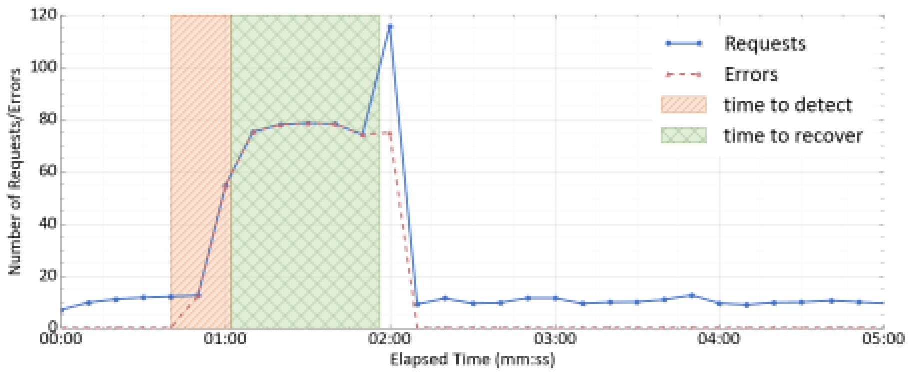
5.6. Impact of Granularity on Detection and Recovery Time

We analyze the impact of monitoring granularity on the detection time of failures in applications. We start with a deployment of RUBBoS app. Twenty seconds after the deployment is complete, we start a RUBBoS client with a workload that does not overload the application. The number of simultaneous users for the benchmark is set to 400. Sixty seconds after the deployment, we crash the MySQL instance. The monitoring strategy uses a simple rule that signals a failure when no monitoring data is received for three consecutive periods.

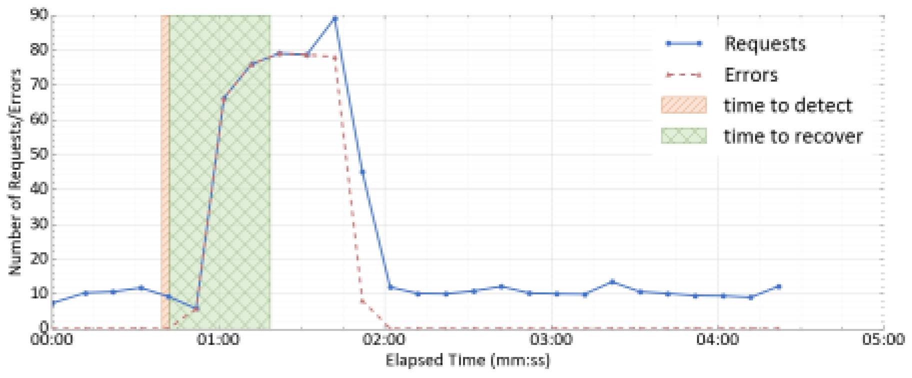
LAMA handles the recovery automatically by allocating resources for the service. The provider agent copies a new disk image for the chosen host and launches a new MySQL instance. LAMA publishes notifications to the user. The notifications include failure detection and deployment of new instances. We measure detection and recovery time from these notifications. We run this experiment for monitoring periods of 1s to 60s. Figure 45 shows the number of requests and errors for two experimental runs with different monitoring granularity (1s, 10s and 60s). The orange area with a diagonal pattern represents the time to detect the failure. The failure detection time is measured from the moment the failure happens until LAMA flags it. The green area with a cross-hatch pattern represents the time to recover. The recovery time is the time since the failure is flagged by LAMA until the replacement instance is active and configured.



(a) 60-sec monitoring granularity.



(b) 10-sec monitoring granularity.



(c) 1-sec monitoring granularity.

Figure 45 – Time-to-detect and time-to-recover from a crash failure.

After we trigger the failure, the number of errors due to timeouts increases to the same number of requests. The number of requests increases as we set a short time out (a few seconds) for each request. Clients retry loading the same page. As we change the monitoring granularity, we can see how the bottleneck changes from the detection to the recovery procedure. The failure detection time ranges from 2 to 3 minutes when the monitoring interval is 60s. The failure detection is negligible compared to the recovery time when the monitoring interval is 1s. The recovery time remains constant. We address improvements of recovery time in other modules of the LAMA framework. Management strategies can make use of warm and hot instances to decrease the recovery time. Figure 46 shows how the detection time varies with the monitoring periods. The monitoring granularity is linearly proportional to the detection time, as expected. This experiment shows the importance of finer monitoring and diagnosing granularities. Current commercial frameworks are limited to several minutes of detection time. Our system can definitely decrease the detection time to few seconds.

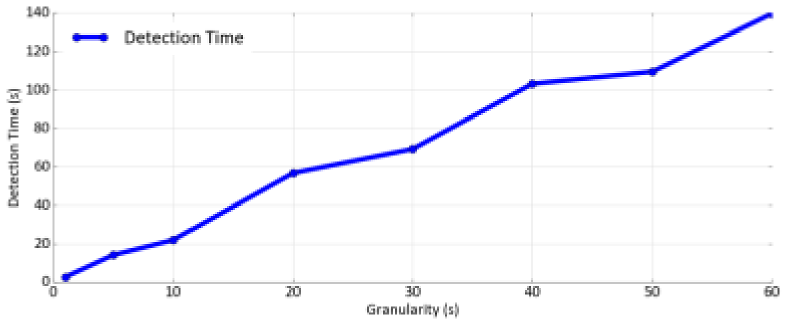


Figure 46 – Time-to-detect with respect to granularity (or monitoring period).

5.7. Assumptions

Assumption 5. Monitoring is not affected by failures.

We do not consider failures that can affect the monitoring infrastructure. Delays in the LAMA's monitoring subsystem could cause false positives. The recovery from this false positive events could cause disruptions to application performance.

In the future, we intend to take into account failures that can affect LAMA's framework. On one hand, application agents should use monitoring as an indication of the health of the hosts and provider agents. This allows applications agents to not utilize hosts that it considers unstable or affected by congestion. As we have seen in Section 2.7, we plan to make application agent divisible. This enables us to implement a monitoring strategy between application agents of the same application. Backup application agents will monitor the active application agent health. Through a leader election scheme, backup agents will be able to take over when the performance of an app agent deteriorates.

5.8. Summary

We propose a monitoring and diagnosing framework that aims to significantly reduce detection and increase effectiveness of problem diagnosis. Autonomous distributed app agents aggregate monitoring data and make diagnose decisions per application. The app agents are also environment-aware. They have access to monitoring data from all the entities that affect its application performance. We deploy LAMA in our datacenter and run extensive experiments to validate our design. We demonstrated how an integrated approach, that allows app agents access to the state of app's ecosystem, enables us to construct a customized diagnostic decision tree that increases effectiveness of failure detection and diagnosis. We showed how our monitoring and diagnosing architecture can improve load distribution in the network and eliminate bottlenecks. In LAMA, monitoring traffic in a link depends on the number of instances per application, while in centralized systems the traffic in the bottleneck link varies linearly with the total number of hosts and instances. Finally, we saw how the finer granularities achievable by our platform can decrease failure detection time from minutes to a few seconds.

Chapter 6

Dynamic Management Strategies for Multi-Tier Applications

LAMA is a generic platform that eases the deployment of custom management algorithms for any type of application. LAMA is especially effective for applications with complex and multiple tiers. For instance, management strategies can leverage information of the applications' environment to improve performance of the application and enable advanced coordinated planning for failures. Additionally, LAMA prioritizes applications during recovery periods and accelerate overall recovery.

Using LAMA, we develop a dynamic management strategy to control a multi-tier web application. The cloud user needs to manage the performance of multiple components that can affect the application in different ways. The strategy is driven by user-configured service-level agreement (SLA) parameters and by the application graph. It leverages information about the infrastructure in order to determine performance of the application and to pre-plan for failures that affect application performance. Our goal is to develop a generic strategy for multi-tier application management. However, some formulas and/or parameters can and should be customized to a specific application. LAMA framework facilitates the deployment of custom management algorithms to better fit apps needs. Additionally, LAMA provides visually rich interfaces that help cloud users to get a better sense of the state of the application and how it will respond to potential future events.

6.1. SLA Definition

In our strategy, a continuous SLA is defined as a desired performance level over a fraction of time within a sliding time window. It is completely defined by three parameters, $(\eta_{SLA}, T_{SLA}, A_{SLA})$:

- **Target Performance Index** (η_{SLA}): a threshold above which a key performance metric represents a desired level of performance for the application.
- **Window Size** (T_{SLA}): the size of a sliding-time-window used to evaluate SLA performance metrics;
- **Availability** (A_{SLA}): the fraction of time the application is available above the defined performance-level.

The goal of our strategy is two-fold. First, the strategy ensures that the application has enough resources to achieve performance level η. Second, the strategy pre-plans allocations using warm, hot or extra active instances as necessary. It ensures that, upon a failure, the application returns to the desired performance within the time available for recovery. Typically, the word *downtime* is used to refer to a period of time when the application is unavailable. In our strategy, instead of downtime, we use the term *degraded time* to refer to periods when the application is performing below the requested performance index.

6.2. Performance Metrics

An application may consist of multiple diverse services. The state of each service is evaluated using three metrics:

- **Performance Index** (η): Measures the performance of the application. In the current LAMA implementation, resource usage is used as a metric. Performance depends on the available resources from the original allocation. This index is used to determine scaling needs.
- **Resilience Index** (ψ): Measures the performance of the application in the event of a failure. This index is used to determine additional active instances that need to be allocated if the SLA application does not allow degraded time.
- **Recovery Time** (r): Measures the expected time to recover from a failure, during which the application might experience degraded performance following a failure. The recovery time includes the time the framework takes to detect a failure and to provision the necessary instances to bring the application to desired performance levels.

The management strategy makes all decisions based on these metrics.

6.3. Recovery Time Estimation

App agents control every step of instance provisioning. They know:

- The monitoring strategies applied. Thus, agents know the maximum time it takes to detect each failure;
- Where instances are allocated and images are stored;
- The time taken by each phase of the provisioning procedure for previous instances.

Thus, the app agent estimates the time it takes to deploy new service instances. By comparing this time estimate to the time available to recover from failures at a given time, the app agent decides what kind of recovery instances needs to be provisioned.

In order to estimate recovery time per instance type, we estimate the time to detect the failure and measure the time of four recovery steps: (1) resource allocation; (2) image deployment; (3) instance bootstrapping (for Cold and Warm instances); (4) instance resumption (for Hot instances).

In the current LAMA implementation, we use a simple prediction mechanism of $\mu + k \cdot \sigma^2$ for each of the phases. μ and σ are the mean and variance of the observed values for the duration of each phase respectively and k is a configurable parameter that regulates the estimation error.

Failure detection time has a significant impact on downtime. Recovery procedures can only start once the failure is detected. Thus, when the available time to recover is below the estimated maximum detection time, the only resilient option is to provide extra active instances. The default load-balancer redirects requests to alternative active instances when it suspects that one of the instances is down.

6.4. Resilience Graph

Using the application graph in Section 2.3, we can estimate the performance metrics of the application. The performance of the nodes on each layer is computed using the dependent nodes on the layer below. To compute the performance metrics defined in Section 6.2, we build a *resilience graph* (see Figure 47). The resilience graph is a directed graph that maps the structure of the application. In the resilience graph, edges between the nodes in different layers contain information about the failures that can affect the relationship between nodes. The edge between a service node and an instance node includes a crash failure of the instance. For each failure, we define a *penalty* factor, $p \in [0,1]$, that represents the magnitude of the impact. We also add a root node that represents the overall performance of the application.

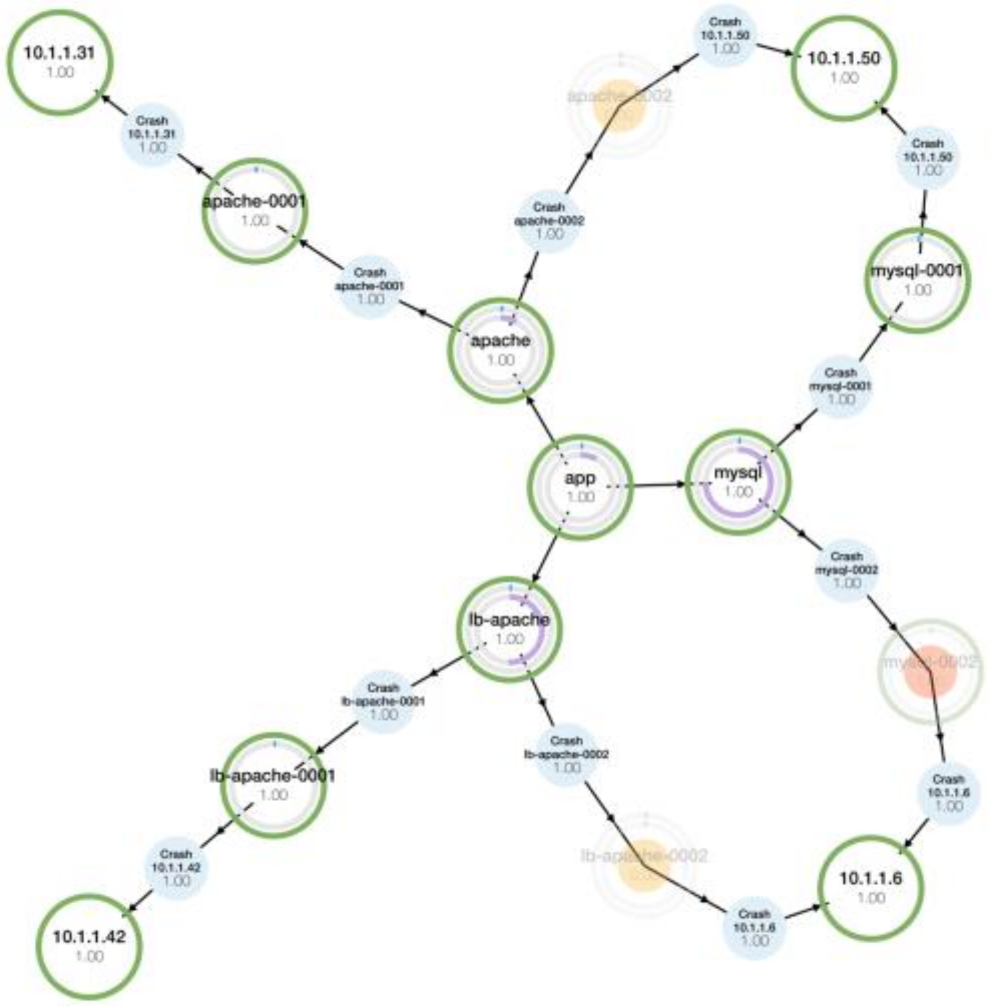


Figure 47 – Sample Resilience graph.

Each node in the graph includes the name of the node (e.g. service name, instance name or host IP address) and information about the node’s performance. Transparent nodes represent latent instances. Nodes with a red center represent hot instances, while nodes with an orange center represent warm instances.

The nodes in the graph are represented using up to three donut charts. Each of the charts indicates one of the key performance metrics. The performance and resilience indices are represented using green and blue charts. A full circumference indicates a value of 1 for the respective index. Consider the service represented on Figure 48. The almost-full green circle indicates that the node is currently performing well. An almost-full blue circle indicates that the node will continue to perform well if any of the failures below the node in the resilience graph is triggered.

The recovery time index, in purple, has a different meaning. The full circumference represents the total time to recover as indicated in the SLA. The purple section, indicates how much time will be left after recovery from the worst-case failure. The failure that imposes the longest recovery time is considered the worst-case scenario. Thus, the recovery circumference indicates the margin of time available for recovery once the worst-case failure occurs.

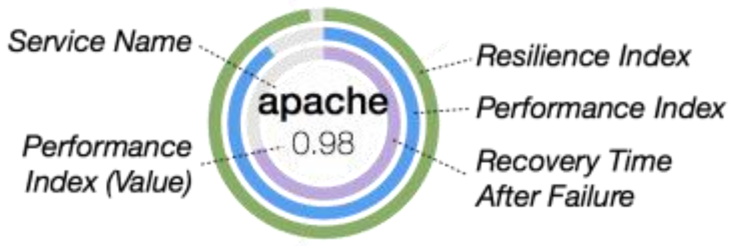


Figure 48 - Sample service entity node.

In Figure 47, the circles in the edges between each pair of entities represent the failures considered. The number inside represents the penalty factor associated with each failure. For instance, between each instance and its host, we represent a host crash failure. Between each service and each instance, we represent an instance crash failure. Failures closer to leafs will have an impact on the metrics all the way to the root of the graph.

To evaluate the performance and resilience of the application, we traverse the resilience graph using a post-order depth-first search. During evaluation of the graph, information about failures is propagated to the root of the tree. Thus, each node's resilience is computed with respect to all failures below it. After evaluation is complete, the app agent has information about the resilience per failure for the entire application. The app agent uses the resilience graph to deploy additional active or latent instances to meet the SLA.

6.4.1. Failure model

We consider instance and host crash failure. Crash failures imply a penalty factor of $p = 1$, i.e. the failure causes total loss of the affected resources.

6.4.2. Recovery model

Recovery is done at the service level. The app agent may deploy extra active, hot or warm instances according to the target response time or resilience index to failures. The actual recovery depends on each service's characteristics. For horizontally scalable services, the app agent can increase the number of active instances indefinitely. For non-scalable services, it can only allocate backup instances to be activated in case of failure. The initial recovery time available (r_{SLA}) is computed from the SLA parameters:

$$r_{SLA} = (1 - A_{SLA}) \cdot T_{SLA} \tag{1}$$

6.4.3. Performance Evaluation

For each node, we compute an instant performance index that represents the expected performance of application requests. The actual impact on the final application performance depends on the structure of the graph, for instance, based on the number of instances per service.

6.4.3.1. Physical nodes

In the absence of over-allocation, the performance of physical servers would be irrelevant. However, even without over-allocation, VM interferences can have an impact on the applications. Cloud providers can also allow use over-allocation to increase efficiency of resources. The performance index of physical nodes indicates how the state of a server can affect the instances. The same server state can affect different applications differently. Thus, the computation of the performance index for server nodes should be customized for each application.

User requests in web applications requests can typically be split into read (e.g. request a page or resource) and write requests (e.g. update state, write to database). The server uses various resources and contention among these resources are inevitable. As seen in Chapter 5, CPU spent running user space and CPU spent on kernel processes indicate possible contention for CPU time. CPU spent waiting for IO operations indicates contention for disk operations. We define the performance index of physical nodes (η_H) as a function that varies exponentially with the resource usage:

$$\eta_H(u) = 1 - e^{\alpha_{cpu} \cdot (\beta_{cpu} \cdot (u_{user} + u_{sys}) - 1)} \cdot e^{\alpha_{io} \cdot (\beta_{io} \cdot u_{io} - 1)} \tag{2}$$

The parameters α and β are determined empirically by studying application behavior and response to interference. We developed a monitoring strategy that estimates these values by observing the response time of the application and correlating it with host and instance resource usage.

6.4.3.2. Virtual Nodes

We define the performance of an instance (η_I) based on the resource usage of the instance ($\eta_{I|\emptyset}$) and on the performance index of its physical host ($\eta_{H|I}$). We compute the performance index as a function of the total CPU usage of the virtual instance.

$$\eta_{I|\emptyset} = 1 - e^{\alpha\cdot(\beta\cdot u_{virt}-1)} \tag{3}$$

We estimate the performance index of the instance as the minimum between the performance of the instance and the performance of the server hosting the instance.

$$\eta_I(u_{virt}) = min\{\eta_{I|\emptyset}, \eta_{H|I}\} \tag{4}$$

6.4.3.3. Logical nodes

The performance of a scalable service depends on the performance of all its instances. The impact of an instance depends on the fraction of workload that instance handles. Thus, the performance of a service is computed by the weighted average of the performance of their active instances.

$$\eta_S = \sum_{i}^{N_I} \omega_i \cdot \eta_I^{(i)} \tag{5}$$

The weights ω_i correspond to the fraction of requests that are distributed to each instance by the predecessor load balancer service. If a service is not horizontally scalable, the performance of the service is the same as its active instance. In this case, the only possible strategy for performance improvement is to use vertical scaling, i.e. an instance with more resources.

6.4.3.4. App node

The performance of the app node (η_{App}) depends on the performance of its services. We consider that the app performs as well as their worst service.

$$\eta_{App} = \min\left\{\eta_S^{(1)}, \dots, \eta_S^{(N_S)}\right\} \tag{6}$$

6.4.4. Resilience Index

The resilience index indicates the app's ability to continue working in the event of a failure. We consider the worst-case failure. The resilience for each node depends on available recovery mechanisms. We provision replicas per service in order to protect from crash failures in the instances and hosts. Only logical nodes that correspond to services of the app can be resilient to failures.

Physical nodes correspond to leafs in our resilience graph. As such, from the perspective of our strategy they are not replaceable and are not resilient. In this work, we do not consider live migration, which could be used to recovery a specific virtual instance. Thus, we only compute the resilience index at the service and app levels. However, failures that are associated at lower nodes still need to be considered. For instance, host crash failures affect availability of instances.

6.4.4.1. Logical nodes

For a service, the resilience index per failure corresponds to the service's estimated performance during the failure. For each failure, we consider the impact of redistributing the workload among the instances not affected by that failure. Consider $F = \left\{f^{(1)}, \dots f^{(K)}\right\}$ as the set of failures that can affect instances of the service. Consider $I_{f^{(k)}}$ as the set of instances affected by the failure $f^{(k)}$. The workload increase on the remaining active instances is given by:

$$\rho_{f_k} = \left(1 + \sum_{i \in I_{f(k)}} p_i \cdot \omega_i\right) \tag{7}$$

In our implementation, workload is distributed evenly among instances and, $\omega_i = 1/N_I$. The resilience index per failure indicates the performance index when the failure triggers the loss of instances:

$$\psi_{S|f^{(k)}} = \min_{i \notin I_{f(k)}} \left\{\eta_I^{(i)} \left[u_i \cdot \left(1 + \rho_{f^{(k)}}\right)\right]\right\} \tag{8}$$

Finally, we compute the resilience index per service, ψ_S:

$$\psi_S = \min_{k} \left\{\psi_{S|f^{(k)}}\right\} \tag{9}$$

6.4.4.2. App node

The resilience of an app is computed by aggregation of the resilience of its services. Just as in the case of the performance index, the app resilience matches the minimum resilience index among its services:

$$\psi_{App} = \min\left\{\psi_S^{(1)}, \dots, \psi_S^{(N_S)}\right\} \tag{10}$$

6.4.5. Recovery Time

Given the availability, A, defined in the SLA we can determine the time available to recover from failures for any given time window of size T_{SLA}. The available recovery time depends on the on the SLA-defined availability and the total degraded time (D) observed in the interval $[t, t - T_{SLA}]$:

$$r(t) = (1 - A_{SLA}) \cdot T_{SLA} - D(t, t - T_{SLA}) \tag{11}$$

6.5. Pre-planning for Scaling and Failures

6.5.1. Scaling

In order to scale application services, we need to know workload fluctuations that an active instance can support. We define a user-configurable parameter, $\delta_W = W_t/W_{t-1}$. It indicates the relative workload increase supported at time t. Assuming that resource usage changes linearly with the workload, we compute the number of active instances need to accommodate the expected fluctuations. From Equation (3) we compute the maximum resource usage that maintains the instance below the SLA, u'_{virt}. Given δ_W:

$$u_{virt}^{(i)} \leq \frac{u_{virt}^{\prime(i)}}{\delta_W \cdot \omega_i} \tag{12}$$

We scale up the service by adding an extra instance when this value is surpassed for any of the instances. We use a similar formula to scale down the service.

$$u_{virt}^{(n-1)} \geq \frac{u_{virt}^{\prime(n-1)}}{\delta_W} \cdot \frac{1-\omega_n}{\omega_{n-1}} \tag{13}$$

In order to avoid frequent changes, we scale up or down when the conditions (12) and (13) are violated for T_S seconds (a user-configurable parameter). The scaling mechanism outputs the required number of active instances, n, to handle current and future workloads. The actual number might increase after evaluation when the application resilience is considered. Thus, the app agent marks the oldest n instances as active to differentiate them from recovery active instances.

6.5.2. Pre-planning for Failures

To be resilient, the application services require additional active or latent instances. We take a greedy approach by allocating the minimum number of instances that will maintain:

(1) the performance index above the SLA target value;

(2) the predicted recovery time for the current window under the available recovery time (Equation (11)).

The procedure to determine which recovery instances to deploy per service is as follows:

1. Inputs:
   1.a. The scaling procedure determines the number of required active instances, n;
   1.b. List of recovery instance states, S_R^0; states are represented by integers, 0: None, 1: Cold, 2: Warm, 3: Hot, 4: Active.
   1.c. Current target recovery time (r) from equation (11)
2. Use the recovery estimator module to determine the lowest instance state (S_T) that has recovery time lower than r;
3. Initialize:
   3.a. List of target states S_R with size $|S_R^0|$ and elements set to 0.
   3.b. List of constraints C_R with size 0;

```
    3.c. List of triggers $T_R$ with size $f^{(k)}$ and elements equal to an empty set.
4.  For each failure $f^{(k)}$ associated with the service:
    4.a. Determine the number of non-recovery active instances affected by the failure, $n_{f^{(k)}}$. This denotes
         the number of instances that need to be recovered;
    4.b. Determine a mask $M$ of recovery instances that are not dependent on the failure. $M$ is a binary array
         of size $|S_T|$, where 1 indicates dependency of the instance on the failure.
    4.c. for $i = 0, \ldots, |S_R|$:
         -  if $M[i] = 0$: set $S_R[i] = S_T$, add $i$ to $T_R[k]$, and $n_{f^{(k)}} = n_{f^{(k)}} - 1$;
         -  if $n_{f^{(k)}} = 0$: break for loop;
    4.d. if $n_{f^{(k)}} > 0$, for $i = 0, \ldots, n_{f^{(k)}} - 1$
         -  append $S_T$ to list $S_R$, append set $\{f^{(k)}\}$ to $C_R[i]$, add $|S_R|$ to $T_R[f^{(k)}]$;
5.  Output:
    5.a. List of desired states, $S_R$;
    5.b. List of constraints for new instances, $C_R$;
    5.c. List of triggers for each failure, $T_R$.
```

Now the app agent can adjust the instances:

```
1.  for $i = 0, \ldots, |S_R^0|$:
    1.a. if $S_R = 0$, remove instance
    1.b. if $S_R[i] \neq S_R^0[i]$, change instance from $S_R^0[i]$ to $S_R[i]$;
2.  for $i = 0, \ldots, |C_R|$:
    2.a. Request new instance with constraints $C_R[i]$ to the provider agent.
```

The provider agent tries to find allocations that satisfy all the constraints. If it is unable to do so, the app agent splits the constraints into multiple instances. Additionally, latent allocation requests carry information about the failures that trigger their activation. The provider agent determines if there are latent allocations that depend on the same failure. This procedure avoids over-allocation when many applications use the same host to recover from the same failure.

The app agent uses the computed triggers (T_R), to quickly activate the respective latent instances in case of failure.

6.5.3. Complexity and optimization

The complexity of evaluating the entire resilience graph depends on the number of instances and the number of failure classes, $|C_F|$. For instance, as illustrated in Figure 47, we consider two failure classes: instance and host crash failures. In the worst case, we evaluate the resilience graph in $O(N \cdot |C_F|)$ time. If a user is using LAMA's graphical interface, we need to periodically recompute the entire resilience graph. However, for changing the application deployment, the resilience graph can be updated on key events: when failures are detected, or when resource usage crosses key thresholds. In this case, we only need to re-evaluate the graph from the affect node to the root of the graph.

6.6. Monitoring Strategies

Monitoring strategies are used to detect relevant events that require adjustment of the application deployment. There are two types of relevant values. When resource usage crosses thresholds defined in (12) and (13), we recompute the application's performance and re-evaluate the resilience graph. When an instance or host failure occurs, we adjust application deployment using the resilience graph. Our implementation defines three management strategies:

- **Instance crash detection:** Signals a failure of an instance when the agent stops receiving CPU usage metrics for the instance after three periods.
- **Host crash detection:** Signals a failure of a host when the agent stops receiving CPU usage metrics for the host after a period three periods.
- **Resilience graph re-evaluation:** Rule-based monitoring strategy that triggers recomputation of resilience graph and recovery mechanisms, when conditions (12) and (13) are violated.

The monitoring period is defined in the configuration of the low-level monitoring module (collectd). The app agent automatically detects the period as it receives new metrics. For finer granularities, the monitoring strategies can use a minimum period of time instead of number of periods.

6.7. Coordinated Recovery

As seen in Section 6.5.2, a latent instance covers for a set of failures. We call this set, the latent allocation's failure dependencies. A new resource reservation request for latent instances carries information about its failure dependencies. Provider agents only reserves the latent resources, if there is no over-allocation for of the failure dependencies.

When the provider agent receives a new active allocation or a request to change an instance from latent to active, it checks the current latent allocations. If there is an over-allocation, the latent allocation is deleted and the app agent is notified. The app agent removes the latent instance from the application graph and re-computes the resilience graph.

6.8. Experiments

We analyze the advantages of LAMA with multi-tier online web applications in a very dynamic environment. We demonstrate that:

(1) LAMA's monitoring and provisioning features can dramatically reduce time to detect and recover from failures;

(2) LAMA enables dynamic management strategies that adapt given application's SLA and past history of degraded time;

(3) Distributed pre-plan for recoveries enables differentiated recovery per application and accelerates total recovery.

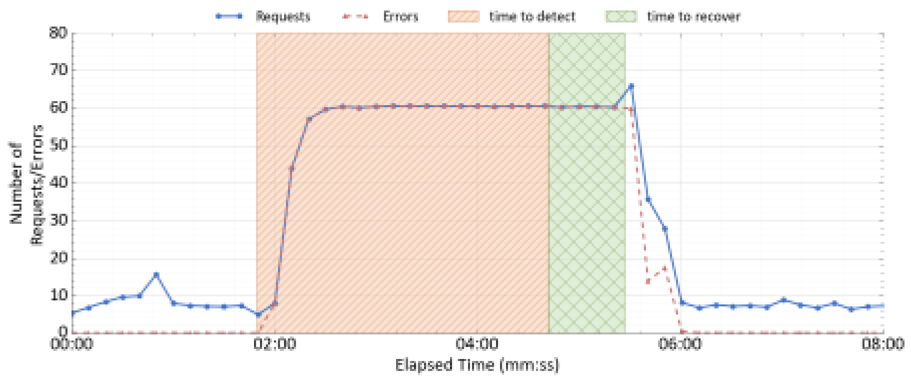
6.8.1. Testbed and Experiment Setup

We implement and deploy LAMA in our datacenter with 24 interconnected heterogeneous servers. The available resources amount to a total of 84 logical cores and 200GB of RAM. Our experiments focus on online multi-tier transaction applications. We use LAMA to automatically deploy a online transaction benchmark, RUBBoS [52] that emulates a bulletin board. LAMA reads a logical specification and provisions the application's services (Apache and MySQL). As described in Section 2.3.2, LAMA also deploys load-balancers before scalable services.

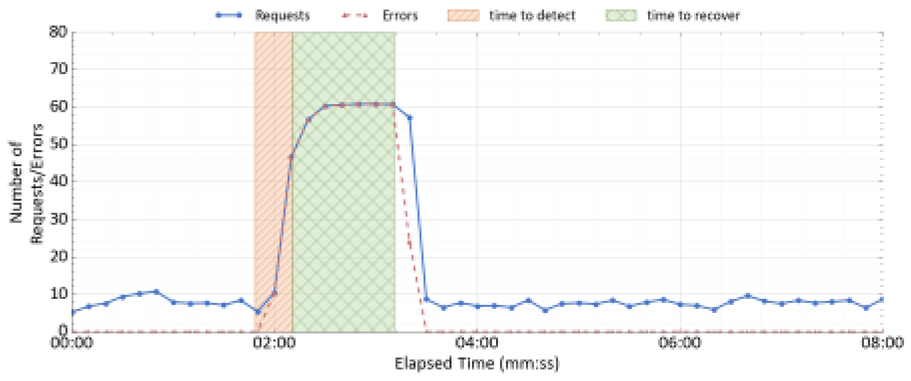
6.8.2. Reducing Total Recovery Time

We analyze the impact of monitoring granularity and the recovery time from a failure. We create a RUBBoS deployment with two services Apache and MySQL. The application is deployed automatically by LAMA. Once the application is deployed, we launch the client generator with 300 simultaneous users. Approximately two minutes after launching the client, we create a crash failure for the Apache instance. The app agent detects the failure, and triggers recovery using the resilience graph. We measure the total time taken by the framework to detect and to recover from the failure. We test the performance under five scenarios by varying the monitoring granularity and SLA parameters. The SLA parameters are chosen to induce the app agent to create recovery instances in different states for each scenario. The monitoring strategies for instance crash detection wait for three periods without metrics before signaling a failure.

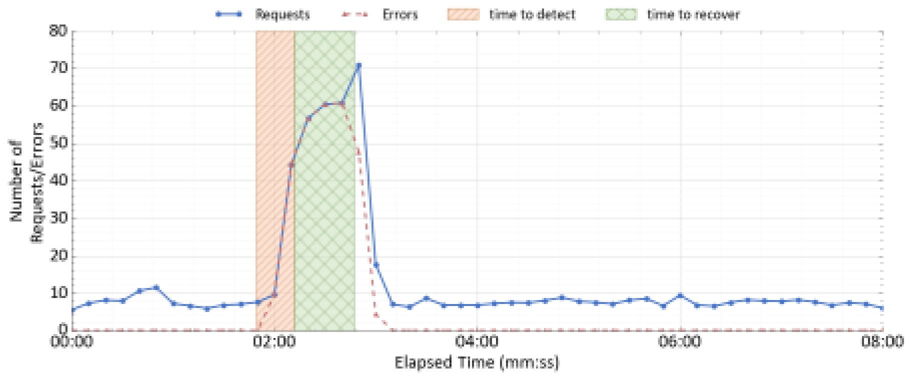
Figure 49 shows the results for each experiment. We plot the number of requests and number of errors observed during the period of the experiment. We also mark the time to detect a failure (orange area with a diagonal pattern) and the time to recover from the failure (green area with a cross-hatch pattern). In Figure 49a, we use a monitoring granularity of 60 seconds and $A_{SLA} = 0$. An availability value of zero forces the app agent to only use *cold* recovery instances. In Figure 49b, we decrease the monitoring period to 10s. Given our failure detection strategy, the detection time is drastically reduced in proportion to the reduction of period size. In this particular example, the reduction is more significant, from 192 to 22 seconds. Larger periods also induce larger deviations in the detection time. In the Figure 49c, we set $T_{SLA} = 500s$ and $A_{SLA} = 0.9$. The app agent creates *warm* recovery instance instead of cold. This behavior depends on the service and time it takes to deploy a new instance. The recovery time is reduced from 60 to 35 seconds. It corresponds to the time it takes to copy the image. In the Figure 49d, we further tighten the SLA parameters by changing the availability, A_{SLA}, to 0.95. The app agent deploys an instance in the *hot* state. In this case, the recovery time was reduced from 35 to 1.5 seconds. This difference corresponds to the time it took to configure the instance. In all experiments, we observe a period of downtime due to slow failure detection time. To further improve the performance of the application, we increase the target availability, A_{SLA}, to 0.99. In Figure 49e, the app agent deploys two active instances. The detection and recovery time, consists of the time it takes to return to a setup with two active instances. We can observe that the application experiences almost no errors during the failure. LAMA automatically adapts the management strategy to meet the SLA requirements defined for each application. The end user does not have to know the availability of host infrastructure.



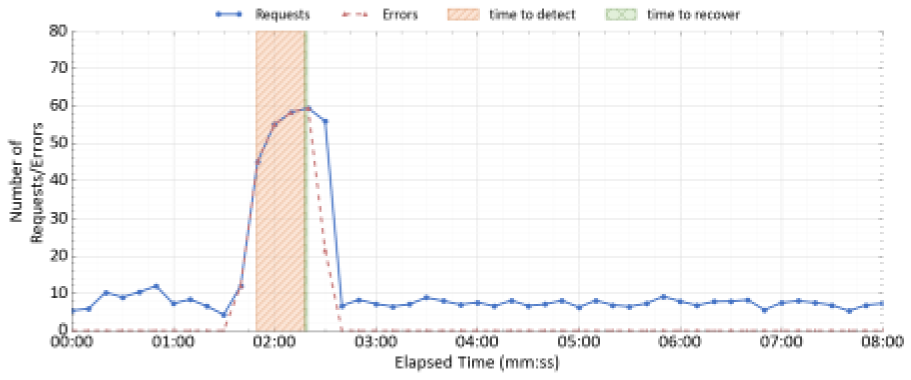
(a) Instance state: *cold*, monitoring period: *60s*.



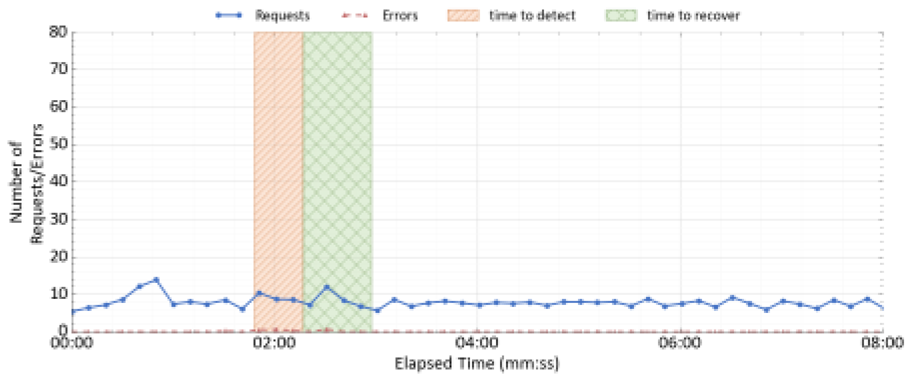
(b) Instance state: *cold*, monitoring period: *10s*.



(c) Instance state: *warm*, monitoring period: *10s*.



(d) Instance state: *hot*, monitoring period: *60s*.



(e) Instance state: *active*, monitoring period: *60s*.

Figure 49 – Time to detect and recover from an instance crash failure and impact on application errors.

6.8.3. Adaptive Recovery Planning

We demonstrate how the app agent adapts its application deployment given the history of failures. As failures occur, the SLA availability becomes harder to meet. We start with a base deployment of RUBBoS with a load-balancer, Apache and MySQL services. We set SLA parameters to $T_{SLA} = 2000sec$ and $A_{SLA} = 0.9$. During the experiment, we trigger five failures at different times. We observe how the app agent responds to each failure and the impact in application availability.

Figure 50 plots the variation of key metrics during the experiment: (1) the time available to recover from failures, $r(t)$, given past downtime period in the sliding window $[t, t - T_{SLA}]$ and (2) the resilience index of the application at time t. Figure 51 plots the variation of the total number of instances of each type (Active, Hot, Warm) during the experiment. Black dashed lines represent the moment that failures occur in the application. We mark periods when application is unavailable using gray bands.

The application starts with a minimal deployment. The app agent provisions one instance for each of the three services. According to the SLA metrics (11), there are *200* seconds available to recover from failures. At *t=2m54s* the first failure occurs. The app agent has no warm or latent instances, so recovery is done from a cold instance. After a period of downtime, $r(t)$ is down to 135 seconds. The available time is still enough to launch a cold instance.

When the second failure is triggered, a new period of downtime follows. When the $r(t)$ drops below the time it takes to provision a cold instance, the agent starts planning additional recovery instances. Initially, it deploys a warm instance. As $r(t)$ drops further, the app agent decides to double the number of active instances per service. At this point, the application is resilient to all the failures considered in the model, $\psi \approx 1$. When the other three failures occur, the application remains available.

At $t = t_{F1} + T_{SLA}$, $r(t)$ starts increasing. At this point, the first period of downtime is gradually leaving the sliding-window considered in the SLA. We observe that, at $t = 40m34s$, $r(t)$ is large enough that the app agent decides that it no longer needs additional active instances. It changes the states of three active instances to hot (two instances) and warm (one instance). The final latent state depends on the estimated recovery time for each service. As the second period of downtime gradually leaves the sliding window, $r(t)$ becomes large enough that the system can again recover from cold instances.

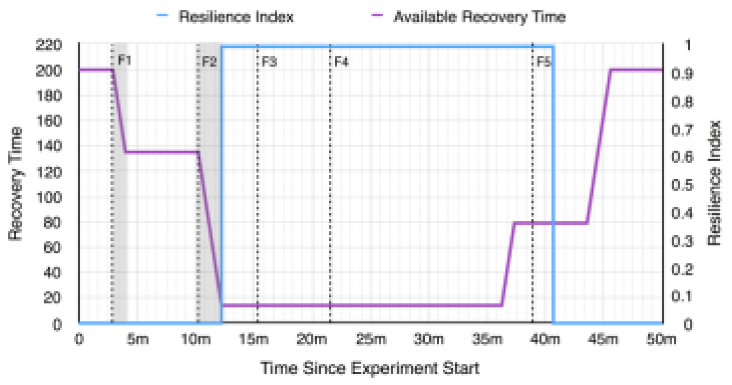


Figure 50 – Adaptive recovery planning: resilience index and time available to recover from failures. Dashed lines represent failures and gray band periods of application downtime.

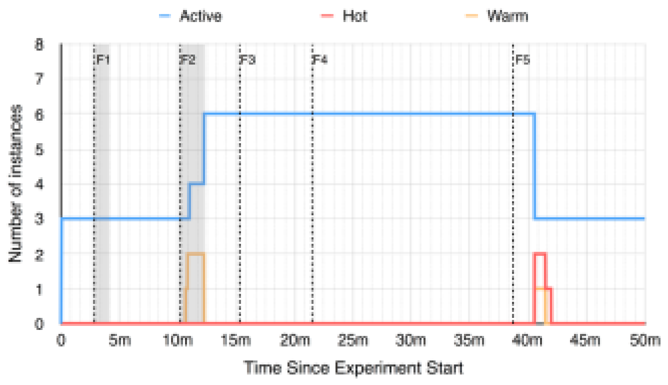


Figure 51 – Adaptive recovery planning: adaptation of the number of active and latent instances provisioned by the app agent. Dashed lines represent failures and gray band periods of application downtime.

6.8.4. Differentiated Application Recovery

In this experiment, we analyze how our strategy provides differentiated recovery performance given application SLA. Our dynamic strategy continuously adjusts the deployment plan based on the time window and availability defined by the SLA. Failures that affect latent instances have no impact on application performance. However, the app agent still needs to adjust its deployment. In this experiment, we take a static approach, i.e. the agent has no memory of past failures. Therefore, for the same SLA, recovery time per instance should be similar.

We start by deploying six RUBBoS applications with identical SLA. We trigger a sequence of n semi-random host crash failures separate by a period p. For each failure, we select a random server that is hosting at least one active instance. We determine, for each application: the number of failures that caused downtime and statistics on the time to recovery per failure. Figure 52 displays the results obtained for $n = 40$, $p = 120s$, $T_{SLA} = 500s$ and $A_{SLA} = 0.9$. The red-shaded region represents recovery time available given by Equation (11).

We observe that the average recovery time per failure is very similar for all applications. However, there are few cases where the recovery time surpasses the SLA threshold. The main factor for deviations is hardware diversity. Some servers with slower disk and CPU, take longer to provision an instance.

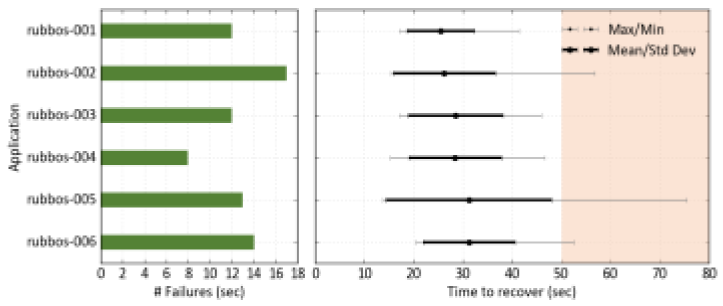


Figure 52 – Recovery time for multiple applications with the same SLA. The left chart represents the number of failures that caused downtime. The right chart represents the total time to recover (minimum, maximum, mean and standard deviation). The red shaded area represents SLA violations.

We now analyze recovery as we deploy applications with different SLAs. We define three SLA classes: (SLA 1) $T_{SLA} = 200s$ and $A_{SLA} = 0.9$, (SLA 2) $T_{SLA} = 800s$ and $A_{SLA} = 0.9$ and (SLA 3) $T_{SLA} = 2000s$ and $A_{SLA} = 0.9$. Figure 53 displays the results obtained for $n = 40$ and $p = 180s$. The SLA violation regions are now different for each SLA class. As we can see, average recovery times depend on the SLA define per application. For class SLA 1, the applications did not experience any downtime. The app agent ensured that there were extra active instances deployed. For the other two classes, the SLA adjusted the deployment to the required latent instances that could be recovered within the SLA-based window. As we are using a static approach, performance of the class SLA 2 is the worst-case scenario. Instances are always recovered from cold latent instances.

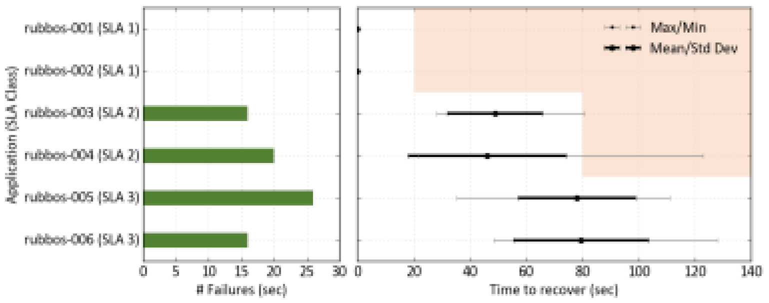


Figure 53 – Recovery time for multiple applications with the diverse SLAs. The left chart represents the number of failures that caused downtime. The right chart represents the total time to recover (minimum, maximum, mean and standard deviation). The red shaded area represents SLA violations.

6.9.Assumptions

Assumption 6. LAMA is always available.

We study how LAMA's features enable the deployment of resilient applications. In our example, we study possible failures in the application and how LAMA is able to trigger recovery of instances marked as failed. However, failures can also occur in the LAMA framework itself. We currently constraint the location of the app agent. We ensure that the app agent is not placed in the same host as any of the application instances. This ensures that failures affecting the application will not affect the app agent (assuming independent failures). However, crash (fail-stop) failures in the LAMA framework affect the ability of applications to recover from failures.

Naturally, LAMA needs to be resilient and self-monitored. Agents currently save their state to local storage. This allows use to restart agents if necessary using old state. However, app agents cannot currently be recovered if their host dies. We plan to deploy multiple agents per application with one controller agent and multiple sub-agents. The controller is responsible for running or delegating management tasks and for sending state changes to sub-agents. The sub-agents in turn, would be responsible for monitoring the health and the availability of the controller and take over if necessary. Provider agents can also fail. However, provider agents are directly connected to the host they are deployed in. If the host dies, the corresponding provider agent does not need to be restarted in another host. Application agents in different hosts will be notified of the failure. In this case, they can adapt the strategy to handle the host failure.

Assumption 7. There are no simultaneous independent failures (i.e. single-failure).

The current strategy is a best effort strategy designed for a single failure at a time. No failures occur while the app agent is recovery from the last failure. If more than two failures occur the time to recover defined in the SLA might not be met. The strategy eventually would recover from failures. However, instances need to be deployed from initial phase leading to longer recovery times than predicted.

We plan to develop a probabilistic fault strategy that will take into account the probability of failures occurring. This strategy will then take into account the probability of simultaneous failures.

Assumption 8. Fault model is defined by cloud providers and application-specific monitoring strategies are defined by users.

We consider independent fail-stop failures for instances and hosts. The failures are defined according to user needs and infrastructure configuration. For instance, if two hosts are in the same rack in a DC, they would fail if the top of the rack switch fails. In this case, a new failure should be added to the resilience graph that will represent the failure that creates a dependency between the two hosts. The strategy can adapt its recovery as the hosts and the instances deployed will be dependent on the same failure. The knowledge about what failures can affect each instance and each host is known only to the cloud provider. These failures per host and instance can be configured in the LAMA framework. The application agent uses that configuration to add failures to the resilience graph when adding a new instance or host.

We deploy semi-synchronous monitoring strategies to detect the instance and host fail-stop failures considered. The period of each metric is determined automatically using the interval between data instances of the same metrics. However, timing failures can delay some data metrics. This can deceive the period detection algorithm, and generate false positives. The application agent would react, thus creating an unnecessary disturbance in the application.

Applications often present odd behaviors in the presence of local instance failures. The cloud user is responsible for defining monitoring strategies to detect anomalous application-specific events. An instance exhibiting anomalous behavior can be reported and replaced like a crashed instance.

We plan to include more faults in the resilience graph. The strategy will need to be adapted to handle other types of faults that do not cause simple loss of resources. Examples of these type of failures are intermittent, byzantine or omission failures. Often monitoring and recovery of these failures are application specific. LAMA was built so that custom monitoring and recovery procedures can be deployed per application.

Assumption 9. State management handled by users through management strategies.

LAMA detects and recovers crashed instances and hosts. Recovery is considered completed once the instance is considered active. The application agent checks if an instance is active using the user-provided application-specific connector. The connector should check if the instance has been correctly configured and has all the state needed to execute its function. Recovering state can be a long process for stateful applications. LAMA is not responsible for state replication for applications. Cloud users deploying stateful applications should implement the necessary state replication mechanisms.

We plan to implement an application-agnostic image backup service in LAMA. This feature would provide state protection for some applications. However, a generic solution would not work for all applications and users opting for this feature would still need to handle state synchronization between instances of the same service.

6.10. Summary

In this Chapter, we introduce a new dynamic strategy for multi-tier online applications for LAMA and provides differentiated treatment according to SLA's requirements and the history of failure.

LAMA's provisioning features allow agents to deploy different type of latent instances. Each latent instance type represents a different stage of instance provisioning. By using estimating recovery times for latent instance type, the agent can provide diverse guarantees to the application. We have shown that LAMA's monitoring and provisioning features can dramatically reduce time to detect and recover from failure.

The proposed dynamic strategy adapts application deployment given past history of degraded time according to application's SLA. As failures occurs, SLAs become harder to meet. We show an example of a how the app agent reacts to failures and adapts the application deployment to face changing SLA requirements.

We applied our proposed adaptive strategy to multiple applications sharing our testbed cluster. We demonstrated that a distributed pre-plan for recoveries enables differentiated recovery per application and accelerates total recovery.

Chapter 7

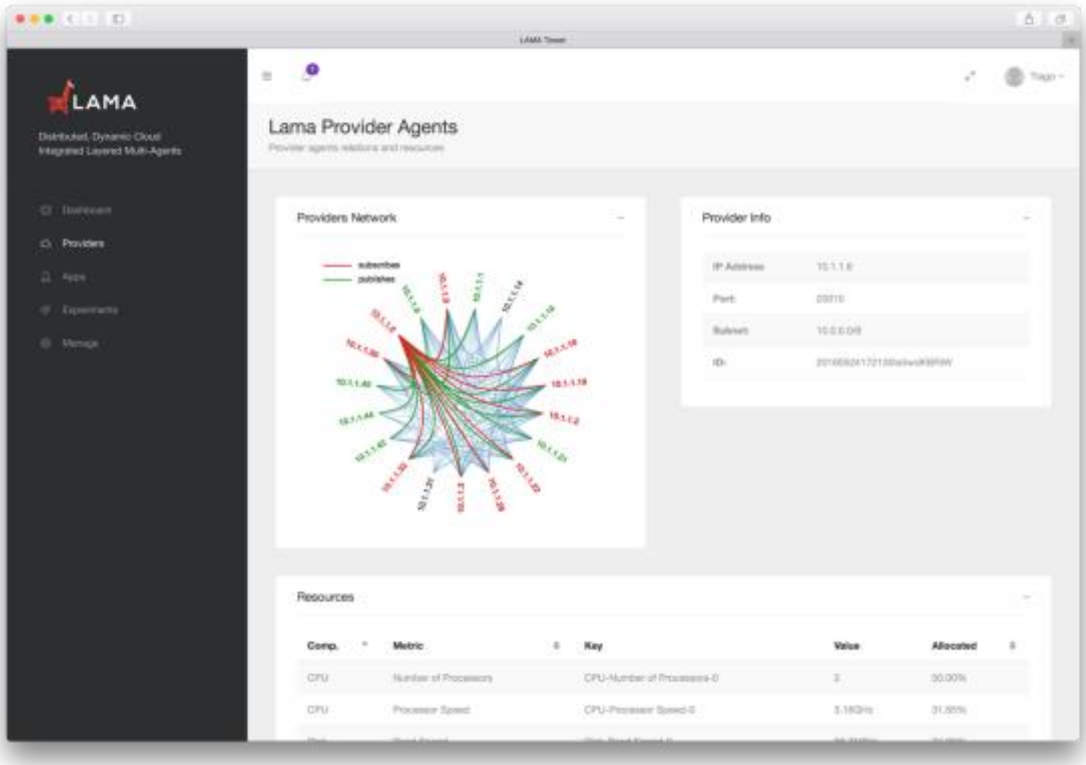
Graphical User Interface

Visualization and management tools are essential for large scale systems. It allows cloud providers and users to quickly assess the state of the infrastructure and applications. These tools are even more relevant in a distributed framework as LAMA. As information is decentralized, users need a tool that is able to find and contact the different agents and aggregate it in a visualization tool.

The web interface needs to adapt to the distributed nature of the LAMA framework. We avoid centralized bottlenecks in all LAMA's management functions. The web interface follows the same paradigm. We build the interface using client-side technology. The web interface uses JavaScript and the recent AngularJS framework. The interface only contacts the centralized dispatcher to retrieve an initial list of providers and to retrieve the location of an application agent. All other requests are directed to the relevant agent. Intensive operations like real-time charts and events or application status data are distributed among all provider agents.

7.1. Provider Agents

The status page for provider agents allows cloud managers to understand the state of the servers and communication between provider agents. Figure 54 displays the provider agents' status page. We can visualize the peer-to-peer relations among providers and the details of each provider: IP address, the server resources, and the percentage of resources available. The radial chart represents the peer-to-peer network of provider agents. Selecting a provider displays the direction of information flow. It highlights, in red, connections and providers from which the provider subscribes resource information. Green lines represent connections to peers to which the selected provider send resource information.



Lama Provider Agents

Provider agents relations and resources

Providers Network

Provider Info

IP Address	10.1.1.6
Port	23010
Subnet	10.0.0.0/8
ID:	[illegible]

Resources

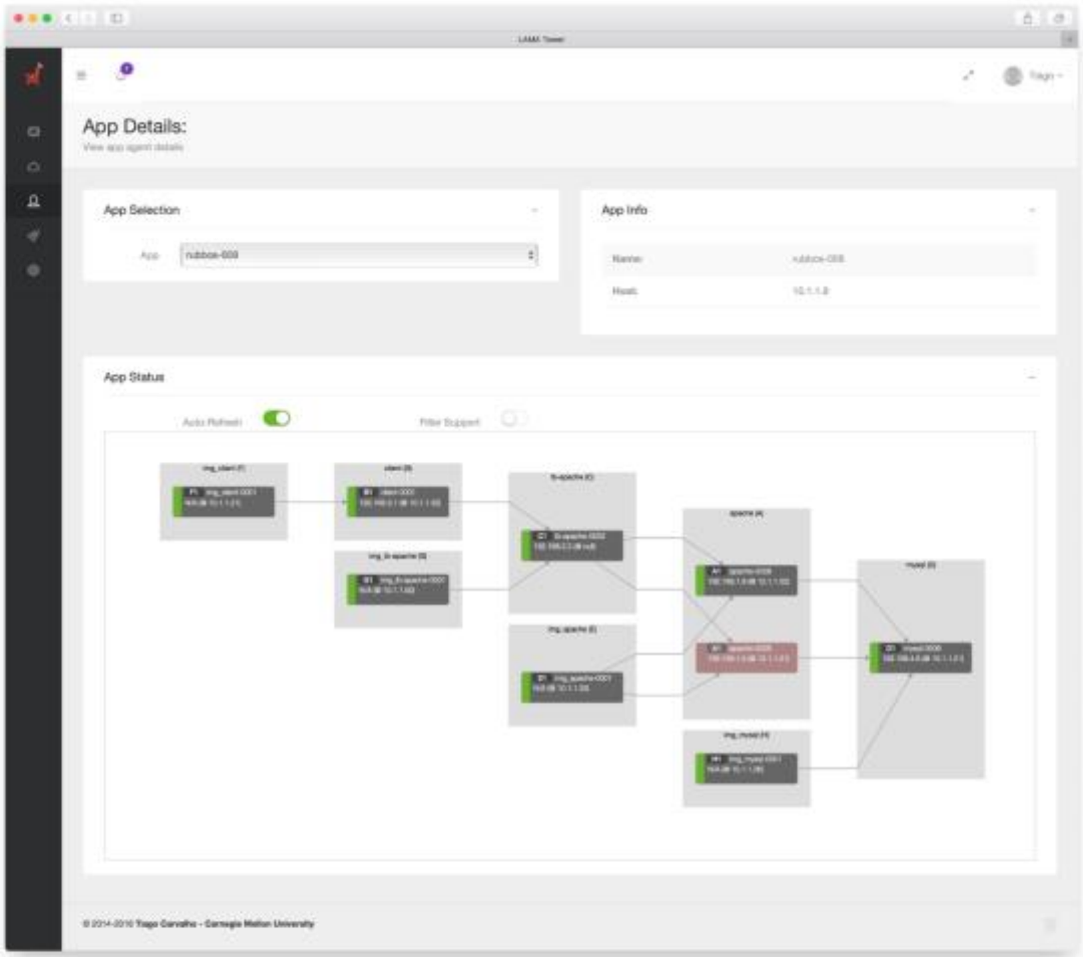
Comp.	Metric	Key	Value	Allocated
CPU	Number of Processors	CPU-Number of Processors-0	2	50.00%
CPU	Processor Speed	CPU-Processor Speed-0	3.16GHz	31.55%

Figure 54 – LAMA web Interface: Provider agents view.

7.2. App Agent and Applications

The status pages for the app agent provide the application owner with information about the agent and the status of the application.

The page in Figure 55 displays information about the application structure. Each node, in dark gray, represents an instance, i.e. a virtual machine or an image. Instances are grouped by logical service inside light gray boxes. Figure 56 presents a detail of the nodes. Each node contains the name of the instance, the virtual address and private address. The state of each instance is indicated on the left side. Three colors are used to indicate performance levels (green for good, yellow for warning and red for bad). The status is defined by a user-configured monitoring strategy. For example, in this screenshot we were using a monitoring strategy that indicates that the instance is performing well when total CPU usage is below 75%. Users can follow the status of the application in real time. When a new node is added, it is displayed with a blinking red color for a few seconds. The users can also filter out support services like image services to improve legibility.



App: rubbos-009

Name:	rubbos-009
Host:	10.1.1.8

© 2014-2016 Tiago Carvalho - Carnegie Mellon University

Figure 55 – LAMA web interface: Application architecture page.

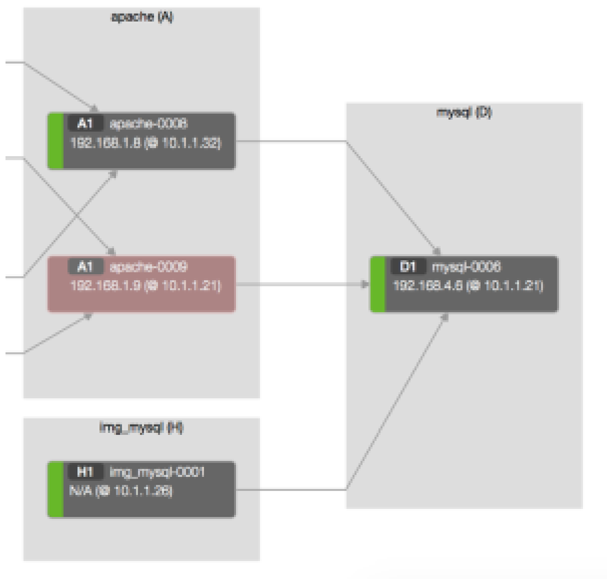


Figure 56 – LAMA web interface: Detail Application Status.

Figure 57 display the web page used to visualize the resilience graph. We described the resilience graph representation in Section 6.4. The resilience graph is updated periodically. The sidebar on the right side indicates the current status metrics for the application node: performance index, resilience index, worst time-to-recover and the current available time to recover. The SLA requirements for the application are also indicated: target performance index, the sliding time-window size and the fraction or time the application should be available.

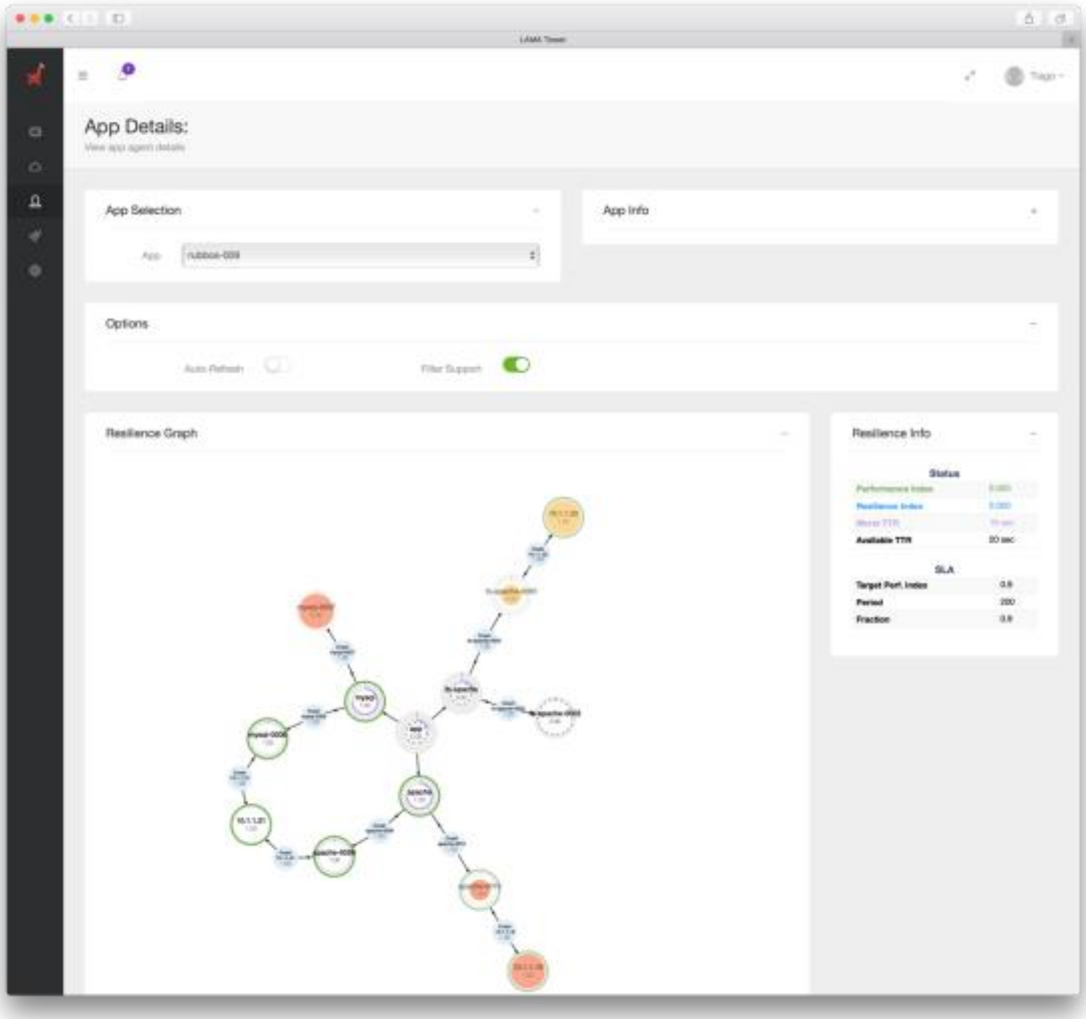
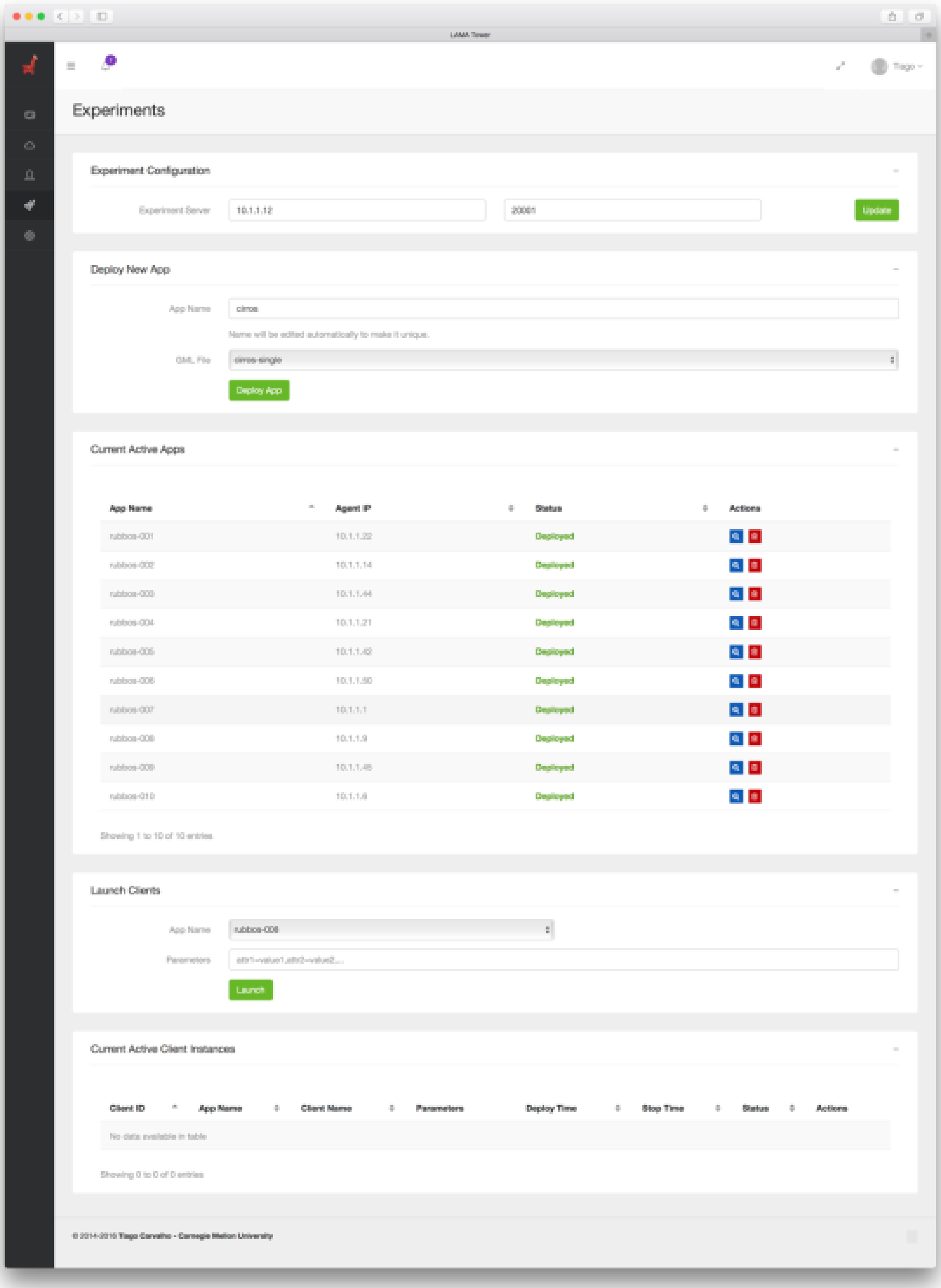


Figure 57 – LAMA web interface: Application resilience status.

7.3. Provisioning and Experiment Control

LAMA also provides an interface to launch new applications. This web page is presented in Figure 58. The user can create a new app by choosing the GraphML file that contains the application spec. The interface includes a list of all the applications currently active in LAMA. As a research framework, LAMA allows users to launch clients to generate synthetic loads to their application. The interface allows the user to configure the parameters of the client generator on launch.


LAMA Tower

Tiago

Experiments

Experiment Configuration

Experiment Server	10.1.1.12	20001

Deploy New App

App Name	cirros
	Name will be edited automatically to make it unique.
GML File	cirros-single

Current Active Apps

App Name	Agent IP	Status	Actions
rubbos-001	10.1.1.22	Deployed	
rubbos-002	10.1.1.14	Deployed	
rubbos-003	10.1.1.44	Deployed	
rubbos-004	10.1.1.21	Deployed	
rubbos-005	10.1.1.42	Deployed	
rubbos-006	10.1.1.50	Deployed	
rubbos-007	10.1.1.1	Deployed	
rubbos-008	10.1.1.9	Deployed	
rubbos-009	10.1.1.45	Deployed	
rubbos-010	10.1.1.6	Deployed	

Showing 1 to 10 of 10 entries

Launch Clients

App Name	rubbos-008
Parameters	attr1=value1,attr2=value2,...

Current Active Client Instances

Client ID	App Name	Client Name	Parameters	Deploy Time	Stop Time	Status	Actions
No data available in table							

Showing 0 to 0 of 0 entries

© 2014-2016 Tiago Carvalho - Carnegie Mellon University

Figure 58 – LAMA web interface: Application and client generator manager.

Automated experiments are triggered through the command line. The user can follow the evolution of the experiment through the timeline displayed in Figure 59. The timeline shows the events published by the agents. The events are represented by color-coded flags that indicate its class (experiment, application or instance). The currently defined events are described in Table VI. A code inside the flag indicates the type of event and contextual tooltip contains extra details. The experiment interface also includes real-time charts of the metrics configured in the experiment.

Table VI – LAMA web interface: Events represented in the experiment timeline.

EVENT CLASS	EVENT CODE	EVENT DESCRIPTION
Experiment	Start	Experiment started
Experiment	End	Experiment ended
Experiment	Fail	Failure was triggered
Experiment	*	Experiment note added by the user
App	AReq	Application requested
App	AAct	Application become active: at least one ready instance for each service
App	ACli	Client generator started
Instance	IAlloc	Instance allocated
Instance	IOn	Instance active (when it is first created by the hypervisor)
Instance	IRdy	Instance ready (when it is accessible by the app agent)
Instance	IDel	Instance removed (if it failed or is no longer needed)

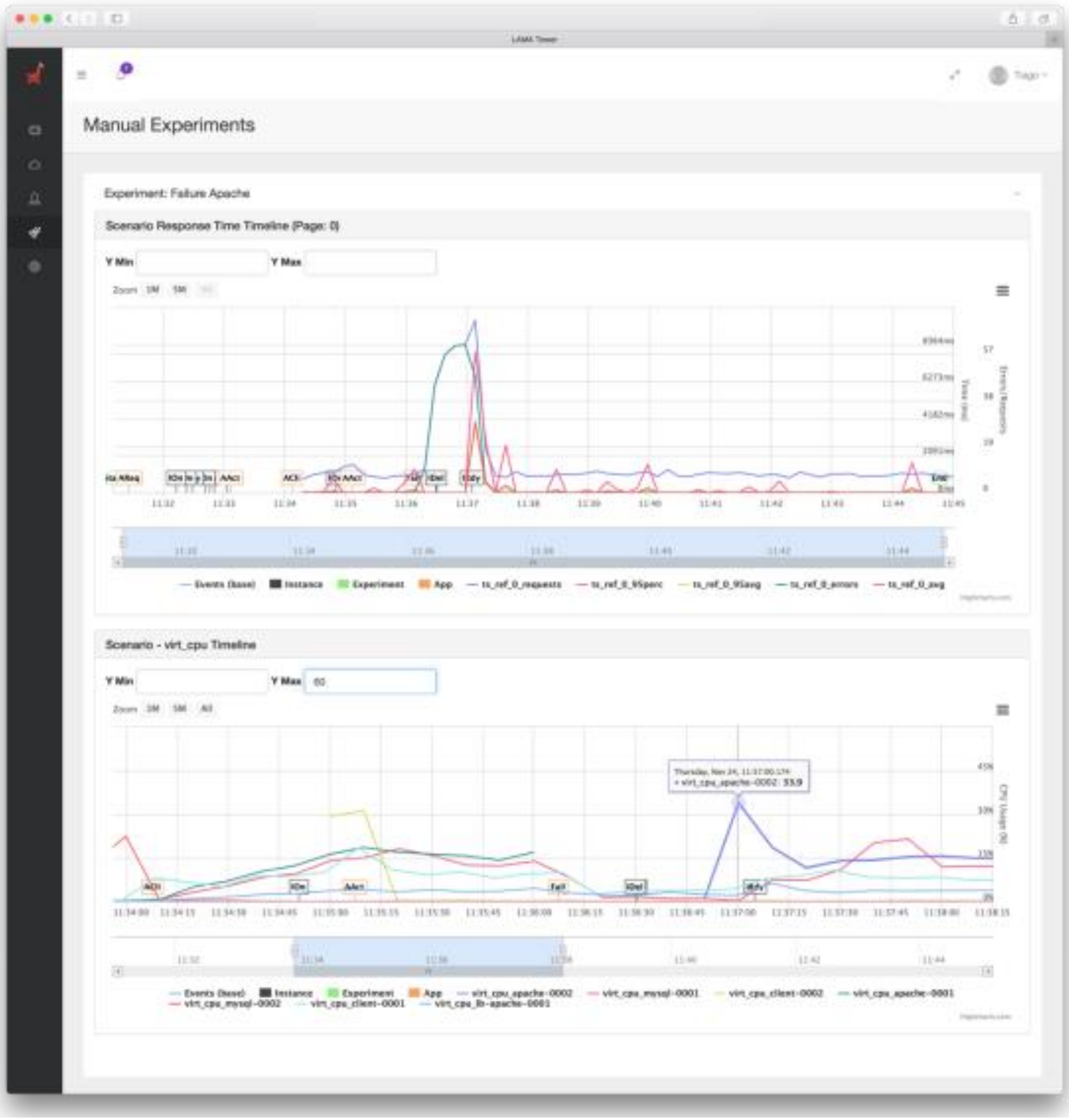


Figure 59 – LAMA web interface: Experiment timeline.

Chapter 8

Conclusion

Management solutions for current and future Infrastructure-as-a-Service (IaaS) Data Centers (DCs) face complex challenges. The large number of application and servers in these infrastructures pose scalability problems. The heterogeneity of application's and hardware characteristics make generic solutions inefficient. These environments are also very dynamic. Applications and hardware evolve. Workloads can vary progressively or in bursts. Frequent failures lead to potential downtime. Current cloud management frameworks are centralized in nature. They provide generic management functions like monitoring, resource allocation and provisioning. There is no differentiation per application. Cloud users manage their applications from outside the cloud using data collected from an external API. The latency leads to slow response to failures. Applications with non-demanding requirements are often over-allocated. On the other hand, application with high availability demands are not able to respond quickly enough to failures.

We propose a novel fully distributed and dynamic management paradigm for highly diverse and volatile DC environments. We develop LAMA, a novel framework for managing large scale cloud infrastructures based on a multi-agent system (MAS). Provider agents collaborate to advertise and manage available resources, while app agents provide integrated and customized application management. Distributing management tasks allows LAMA to scale naturally. Integrated approach improves its efficiency. The proximity to the application and knowledge of the DC environment allow agents to quickly react to changes in performance and to pre-plan for potential failures.

We analyze the performance of LAMA's resource search and allocation. The efficiency of LAMA's resource search depends on the ability to find new available resources using a peer-to-peer network. Resources become harder to find when the DCs has high utilization. We demonstrated that LAMA resource allocation mechanism performs well up to high levels of DC occupancy. Only, when over 80% of the resources are allocated does the search time increases. LAMA also performs well under a high volume of concurrent requests. We observe no increase in resource allocation time with the number of concurrent requests.

We study LAMA's advantages when compared with a state-of-the-art open source framework, OpenStack. LAMA Provisions instances faster as it is able to process more requests in parallel. OpenStack provisioning time per requests grows linearly with the number of concurrent requests. LAMA provisioning time grows only with to local contention, i.e. when several VMs are assigned to the same server at the same. LAMA removes sources of contention in centralized management services like the request API, scheduler and network management. This lead to significant differences in allocation time per instances. For instance, we ran an experiment where we requested to provision 80 virtual instances in a cluster with capacity to 82. LAMA finished the experiment 17 times faster than OpenStack.

We demonstrate the advantages of LAMA monitoring subsystem. LAMA monitoring can achieve finer monitoring granularities while removing centralized bottlenecks. This change has a significant impact in failure detection. Applications can respond to failures in the order of failures. Current cloud providers provide data every few minutes. Any failure detection mechanism would thus take several minutes to detect a failure.

We demonstrate how LAMA dynamic features can be used to further reduce application downtime. We develop a dynamic integrated application management strategy customized for multi-tier online applications. Leveraging knowledge of application's deployment details, SLA requirements and past history of failures, the app agent can determine the latent instances required to respond to failures. We demonstrated that the LAMA coordinated failure planning mechanism allows multiple applications to prepare response to failures. Each application can have differentiated recovery plans according to its SLA.

We also presented LAMA's web interface. The interface allows visualization of the state of provider agents, servers and applications. Users can provision applications, launch client generators and visualize metrics in real-time.

References

[1] M. Armbrust et al., “Above the Clouds: A Berkeley View of Cloud Computing,” (Technical Report). URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf. 2009.

[2] L. M. Vaquero et al., “A Break in the Clouds : Towards a Cloud Definition,” in *SIGCOMM*, 2009.

[3] B. Pfaff, “Extending Networking into the Virtualization Layer,” in *HotNets*, 2009.

[4] Brocade, “The Necessity for Network Modernization,” (Technical Report), URL: https://www.brocade.com/content/dam/common/documents/content-types/whitepaper/brocade-necessity-of-network-modernization-wp.pdf. [Accessed: Oct-2016]. 2013.

[5] R. Nathuji et al., “Exploiting Platform Heterogeneity for Power Efficient Data Centers,” *ICAC*, 2007.

[6] Rogue Wave Software, “CPU Cache Optimization : Does It Matter? Should I Worry? Why?,” (White Paper), URL: http://www.roguewave.com/getattachment/b6524fa0-2f6f-4498-9875-194886ca8def/CPU-Cache-Optimization?sitename=RogueWave. [Accessed: Oct-2016]. 2011.

[7] P. Saab, “Scaling memcached at Facebook,” 2008. URL: https://www.facebook.com/note.php?note_id=39391378919.

[8] “OpenNebula.” URL: http://opennebula.org. [Accessed: Oct-2016].

[9] “OpenStack.” URL: http://www.openstack.org. [Accessed: Oct-2016].

[10] K. Sato et al., “A Model-Based Algorithm for Optimizing I/O Intensive Applications in Clouds Using VM-Based Migration,” in *CCGRID*, 2009.

[11] P. Bodík et al., “Surviving Failures in Bandwidth-Constrained Datacenters,” in *SIGCOMM*, 2012.

[12] N. Bansal et al., “Towards Optimal Resource Allocation in Partial-Fault Tolerant Applications,” in *INFOCOM*, 2008.

[13] K. Amiri et al., “Dynamic Function Placement for Data-intensive Cluster Computing,” in *USENIX*, 2000.

[14] X. Meng et al., “Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement,” in *INFOCOM*, 2010.

[15] D. Breitgand et al., “Improving Consolidation of Virtual Machines with Risk-Aware Bandwidth Oversubscription in Compute Clouds,” in *INFOCOM*, 2012.

[16] U. Sharma et al., “Kingfisher: Cost-aware elasticity in the cloud,” in *INFOCOM*, 2011.

[17] Amazon, “Amazon Web Services : Risk and Compliance Risk and Compliance Overview,” (White Paper). URL: https://aws.amazon.com/whitepapers/overview-of-risk-and-compliance/. 2013.

[18] CA Technologies, "The Avoidable Cost of Downtime," (Technical Report), URL: http://www3.ca.com/~/media/files/articles/avoidable_cost_of_downtime_part_2_ita.aspx. Accessed: Oct-2016]. 2010.

[19] K. Awara, "To 4,000 Compute Nodes and Beyond: Network-aware Vertex Placement in Large-scale Graph," *SIGCOMM*, 2013.

[20] Microsoft, "Disaster Recovery and High Availability for Windows Azure Applications," URL: http://msdn.microsoft.com/en-us/library/windowsazure/dn251004.aspx. [Accessed: Nov-2016].

[21] "GraphML." URL: http://graphml.graphdrawing.org. [Accessed: Nov-2016].

[22] "Open vSwitch." URL: http://openvswitch.org. [Accessed: Dec-2016].

[23] "Collectd." URL: http://collectd.org. [Accessed: Oct-2016].

[24] "Redis." URL: http://redis.io. [Accessed: Oct-2016].

[25] S. Meng et al., "Volley: Violation Likelihood Based State Monitoring for Datacenters," in *ICDCS*, 2013.

[26] B. Hindman et al., "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *NSDI*, 2011.

[27] M. Schwarzkopf et al., "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Eurosys*, 2013.

[28] M. Harchol-Balter et al., "Resource Discovery in Distributed Networks," in *ACM PODC*, 1999.

[29] R. Perlman, "Rbridges : Transparent Routing," in *INFOCOM*, 2004.

[30] L. A. Barroso et al., "The Datacenter as a Computer," Morgan & Claypool, 2009.

[31] D. Abts et al., "High Performance Datacenter Networks," Morgan & Claypool, 2011.

[32] M. Al-Fares et al., "A Scalable, Commodity Data Center Network Architecture," in *SIGCOMM*, 2008.

[33] "NS-3 Official Website," *NS3*. URL: www.nsnam.org. [Accessed: Nov-2016].

[34] Ubuntu Engineering Team, "How we Scaled OpenStack to Laynch 168,000 Cloud Instances," 2014. URL: https://javacruft.wordpress.com/2014/06/18/168k-instances/. [Accessed: Nov-2016].

[35] "Amazon AWS." URL: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring_ec2.html. [Accessed: Oct-2016].

[36] "Microsoft Azure." URL: https://azure.microsoft.com/en-us/documentation/articles/cloud-services-how-to-monitor/. [Accessed: Oct-2016].

[37] "Google Cloud Platform." URL: http://cloud.google.com/. [Accessed: Nov-2016].

[38] OpenStack, "Ceilometer." URL: http://docs.openstack.org/developer/ceilometer/. [Accessed: Oct-

2016].

[39] G. Da Cunha Rodrigues et al., “Monitoring of Cloud Computing Environments: Concepts, Solutions, Trends, and Future Directions,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 378–383.

[40] C. Wang et al., “Performance Troubleshooting in Data Centers: An Annotated Bibliography,” *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 3, pp. 50–62, Nov. 2013.

[41] S. Singh et al., “QoS-Aware Autonomic Resource Management in Cloud Computing: A Systematic Review,” *ACM Comput. Surv.*, vol. 48, no. 3, p. 42:1-42:46, Dec. 2015.

[42] “Nagios.” URL: http://www.nagios.org. [Accessed: Oct-2016].

[43] “Performance Co-Pilot.” URL: http://pcp.io. [Accessed: Oct-2016].

[44] E. Feller et al., “Snooze: A scalable and autonomic virtual machine management framework for private clouds,” *Proc. - 12th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2012*, pp. 482–489, 2012.

[45] C. L. Mendes et al., “Monitoring Large Systems Via Statistical Sampling,” *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 2, pp. 267–277, May 2004.

[46] F. Wuhib et al., “Dynamic resource allocation with management objectives: Implementation for an OpenStack cloud,” in CNSM, 2012, pp. 309-315.

[47] R. Nathuji et al., “Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds,” in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 237–250.

[48] C. Delimitrou et al., “Quasar: Resource-efficient and QoS-aware Cluster Management,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 127–144.

[49] S. Kashyap et al., “Efficient Constraint Monitoring Using Adaptive Thresholds,” in *2008 IEEE 24th International Conference on Data Engineering*, 2008, pp. 526–535.

[50] J. Shao et al., “A Runtime Model Based Monitoring Approach for Cloud,” in *2010 IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 313–320.

[51] S. Meng et al., “State Monitoring in Cloud Datacenters,” *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1328–1344, Sep. 2011.

[52] JMOB, “RUBBoS.” URL: http://jmob.ow2.org/rubbos.html. [Accessed: Oct-2016].

[53] F. Shahar, “btest.” URL: http://btest.sourceforge.net. [Accessed: Oct-2016].