Achieving High Availability at the Application Level in the Cloud

Ali Kanso
Ericsson Research
Ericsson Inc.
Montreal, Canada
Ali.kanso@ericsson.com

Yves Lemieux
Ericsson Research
Ericsson Inc.
Montreal, Canada
Yves.lemieux@ericsson.com

Abstract—Cloud computing is an emerging paradigm that is gaining more attention by the day. Even with the increased number of applications that are being deployed in the Cloud, the question remains, is the Cloud ready to host applications adhering to telecommunication-grade requirements? In this paper we target the issue of the high-availability (HA) requirement in the Cloud from an application perspective. We present an approach that enables the dynamic incorporation of HA features into the deployed applications, which raises the discussion about the feasibility of having HA-as-a-Service, per application, in the Cloud.

Keywords- High-availability (HA), Cloud middleware, runtime upgrade, configuration generation, HA as a service, REST architecture, state-aware applications.

I. INTRODUCTION

Cloud computing offers a business model that abstracts computing resources as commodities that can be offered on demand. Many enterprises are still reluctant to adopt this model, due to the fact that the Cloud paradigm is still evolving, and has not reached the maturity level to host critical applications [1][2]. The dependability issues such as service availability are a major concern for both Cloud users and providers [3]. Cloud providers offering Infrastructure-as-a-Service (IaaS), e.g. [5], are agnostic of the type of applications being deployed in their Cloud, and therefore the protection mechanisms they provide against failures are not at the application level. This is understandable since the outages caused by failures affect stakeholders differently: the IaaS providers are not affected when a software application running in a virtual-machine (VM) fails as long as the VM is healthy, since the VM constitutes the service they are providing. In fact, this type of failure will probably remain undetected by the Cloud middleware. On the other hand, the Cloud users running revenue generating applications in the Cloud are directly impacted by this type of failure, since it is interrupting their services, and as a result, they are concerned with having HA solutions at the application level and not only at the VM level.

Service downtime is not only caused by failures, in reality, planned outages due to software upgrade and maintenance account for a significant portion of the downtime [6]. A full-fledged HA solution should take this factor into account, and enable the applications running in the cloud to continue providing their services even during their upgrade.

In this paper we propose an HA solution for the applications running in the Cloud, and the Cloud middleware itself.

The paper is organized as follows: in Section II we present the background of our work, in Section III we present our solution for HA. In Section IV we discuss the applicability of our solution and in Section V we survey the literature of related works. We conclude and discuss the future work in Section VI.

II. BACKGROUND

A. High availability solutions

The advancement in the information technology is changing the way services are being delivered, and introducing new business models based on broadband internet access, e.g. VoIP (voice over the internet protocol). The services provided by the applications are considered highly available if they are accessible 99.999% of the time (also known as five 9’s). Application developers are expected to focus on the logic of the applications and not on the availability of the application. For this purpose other technics based on virtualization (e.g. [6]) and specialized middleware (e.g. [9]) have been employed. Virtualization-based HA solutions require no modifications to the application’s code or design; however they suffer from several limitations. First, they offer protection against hardware failures while failures at the software level can remain unnoticed. Second, they are based on full duplication, where the standby VM mimics the active VM in a lock-step fashion, which can overload both the computing and the network resources. Moreover, this approach does not provide protection against malicious attacks, as the attack is replicated on the standby VM. Furthermore, having the standby as an exact replica eliminates the option of using N-
version programming to protect against faults in the application or the operating system. In short, using virtualization as a generic availability solution is not suitable to achieve HA at the application level, and this is the main reason why Cloud providers, who are already using such HA solutions, do not guarantee the five 9’s of availability for the user’s applications in their service-level-agreement (SLA).

Specialized middleware for HA offers an alternative solution based on monitoring the applications, and reacting to their failure by isolating their faulty components, and failing over the services to redundant replicas that can resume the service provision. Several proprietary solutions (e.g. [7][8]) are based on this approach. Nevertheless these solutions are platform dependent, and applications developed to run on these platforms will lack the portability features and suffer from vendor lock-in. To remedy these issues, the service availability forum (SAForum) [9] defined a set of specifications for a platform-independent, standardized middleware solution for HA. An open-source implementation of these standards such as OpenSAF [10] can deploy and manage the applications and maintain the availability of their services.

B. SAForum middleware

The SAForum middleware specifications describe a general-purpose HA solution, regardless of the type of the application being managed. The middleware offers a set of complimentary services/frameworks that are essential for maintaining the availability. In this section, we only present the ones that are relevant to our solution.

1) The Availability Management Framework (AMF):

AMF [11] constitutes the core of the SAForum middleware. It is responsible for maintaining the service availability by detecting and reacting to failures. AMF maintains this availability according to a model that represents a logical classification of the system resources. This classification distinguishes the service from the service provider component. This decoupling allows the middleware to support a variety of redundancy models that other HA solutions do not support. The service can then be protected using various models of redundancy such as Active/Active, Active/Standyby and Active/Spare.

Figure 1 illustrates a simplified example of an AMF configuration. In this configuration we have one active component providing a service, and one standby component for that service, ready to take-over. AMF dispatches the active/standby assignments according to the configuration. Components typically abstract software resources. They collaborate to provide and protect a service in the context of a service-group. Each service-group is characterized by a redundancy model that defines how the active/standby assignments can be distributed by AMF among components. All of the elements shown in Figure 1 must be represented by objects in the AMF configuration. The structure of these objects has to comply with a unified-modeling-language (UML) class diagram specified in [11]. The configuration objects must be described according to a standardized machine-readable eXtensible Markup Language (XML) schema [12]. Figure 1 shows some of the component related attributes in their XML format. It is the responsibility of the system integrator to define the AMF configuration.

Figure 1. AMF configuration example

2) Checkpoint service:

The Checkpoint service [13] of the middleware, allows the components at runtime to create checkpoint objects that can store data representing the application state. Once a checkpoint object is created, the checkpoint service will make sure that it is properly replicated within the cluster to avoid losing the state information in case of failure. The checkpoint service offers various modes of synchronizations between the replicated checkpoint objects (e.g. synchronous and asynchronous).

3) Software management Framework (SMF):

Software systems are constantly evolving, especially with development processes like agile where new features and functionality are constantly being added. Interrupting the services in the name of upgrade and maintenance is not acceptable in HA systems, where the allowed service downtime per year is roughly five minutes. SMF [14] defines a framework that enables the software upgrade in a structured manner with minimum service interruption. The upgrade is based on a standardized upgrade-campaign structure.

Figure 2 illustrates the structure of the upgrade-campaign, which is composed of upgrade-procedures. A procedure can be either rolling or single step. In a rolling upgrade the system is upgraded gradually one step at a time, whereby we make use of redundancy to minimize service
outage. A rolling upgrade can have multiple steps. Each step encompasses a set of actions. The number of actions depends on whether the software installation/removal can be performed online (i.e. does not interrupt the system, e.g. by requiring a reboot) or offline. The system integrator designing the upgrade-campaign will decide according to the new software which procedure to follow. Again the upgrade-campaign is specified according to a standardized XML schema [14]. Based on the upgrade-campaign, SMF can automatically take the system from the current state to the desired state. Note that the upgrade is not limited to existing components; we can also use the upgrade to install/remove components. In case of any failures during the upgrade, SMF can automatically fallback or roll-back to the original configuration. SMF also offers administrative operations to execute, suspend, rollback and commit an upgrade.

III. ACHIEVING HA AT THE APPLICATION LEVEL

Our solution for achieving HA at the application is based on deploying the SAForum middleware in the Cloud. Nevertheless, the SAForum solution comprises a level of complexity that is not suitable for the Cloud model, which is based on the notion of simplifying the usability aspect. In this section we first discuss the middleware deployment aspect, and then we present our solution for simplifying the usability aspect.

A. Middleware deployment

In an IaaS setting, we consider the Cloud to be a stack of compute, storage and network elements integrated together to form a unified organization of resources available on demand (Figure 4). At the bottom of the stack we have the network fabric with the networking elements and their interconnections. On the infrastructure level we can have either a bare-metal virtualization layer e.g. [15], or an operating system based virtualization as in [16]. The Cloud middleware is considered as a set of software components collaborating to provide the Cloud services. Those components include the management interface, scheduling, messaging, storage management and others. The VMs, although they run on the virtualization layer, they are considered to be managed by the Cloud middleware (even if this management is indirect). Finally each VM will have its own operating system, and the set of user applications.

Figure 4. Proposed Cloud architecture

In order to achieve HA at the application level, we need to ensure that the entire underlying stack has no single point of failure. For this, we need redundancy at the network layer, in both the data and the management planes. The Cloud middleware components and the VMs need to be highly available. Therefore we introduce an HA middleware
deployment at this level, where the HA middleware will monitor the Cloud middleware components, and repair them in case of failure, and it will monitor the VMs, and restart them somewhere else in case of failure, using the hypervisor capabilities. The communication between the HA middleware and the hypervisor is done using standard APIs, like Libvirt [17]. This middleware deployment will ensure that the Cloud middleware components and the VMs are HA, nonetheless it will not render the applications within the VMs HA, since this deployment will not have access to the content of the VMs. Therefore, another middleware deployment is needed inside the VMs, which can monitor the applications, and react to their failures. Figure 4 illustrates the two deployments. It should be noted here that the two middleware deployments can be completely independent and agnostic of the existence of each other.

B. Integrating the applications with the middleware

There are three different approaches to integrate applications with the SAForum middleware:

- **No integration**: this approach non-intrusive, it does not require any modifications to the application code, the integration is done through the AMF configuration alone, where the life-cycle control (instantiate/terminate) scripts are referenced. The middleware can detect the application failures by using passive monitoring, and react to those failures by restarting or failing over the application (depending on the preferences specified in the configuration). This approach is suitable for stateless applications that do not need to checkpoint their state.

- **Full integration**: this is potentially a more intrusive approach that requires the application to implement the application programming interface (API) required by the middleware to enable the bi-directional communication. Using this approach the applications can benefit from other middleware services such as checkpointing, messaging, etc.

- **Proxied**: in this approach the application interact with the middleware through a dedicated proxy component that mediates the interactions between the middleware and the proxied. The proxy is a fully integrated component that implements the middleware APIs. The middleware cannot control the life-cycle of the proxied without the proxy.

All of the above approaches require that the application description is included in the AMF configuration. Moreover, when this integration needs to be done at runtime, then an upgrade campaign must be defined to enable this on-the-fly integration. Expecting Cloud users to invest the time and effort to acquire the needed domain knowledge and define these middleware artifacts (configuration, upgrade campaign) is not feasible. Moreover, for state-full applications, the developers should not be expected to implement the complex middleware APIs, in order to use the middleware services such as checkpointing.

C. HA Integration Solution

In order to address the complexity issues of the HA integration, we define two agents (Figure 5): (1) the integration-agent that can dynamically modify the system information model to let the middleware be aware of the newly added applications. (2) The HA-agent that will interact with the state-full applications in order to checkpoint their state.

1) State-aware Applications

Stateless applications can be integrated with the middleware using the “No integration” approach. However for state-full applications this kind of integration only satisfies the service availability aspect, but not the service continuity in case of failure. To enable the service continuity while at the same time relieving applications developer from implementing the middleware APIs, we define the notion of state-aware applications. A state-aware application is simply an application that is aware of its state and can save this state as needed. We define the state-aware application behavior as follows: upon instantiation, the state-aware application will try to acquire its last saved state from an HA-agent, if such a state does not exist then the application will start from a default state, otherwise it will continue executing from the last saved state as shown in Figure 6. We believe that application developers know best their application, and what constitutes its state, therefore they can easily define when (synchronous/asynchronous) and what (data type) to save as the state.

![State-aware Application Diagram](image-url)
2) Integration-agent

The Cloud user interacts directly with the integration-agent. The user will provide basic information such as the scripts to instantiate/terminate its application as well as the redundancy model according to which the middleware will protect the application. The user can also specify the installation scripts, and by that fully benefitting from the SMF framework of the middleware that can automatically deploy the application on several nodes without the user intervention. This information is fed to two generators:

- The configuration generator which is an extension to the one we defined in [18]. It takes the user input and automatically generates the configuration XML file. We extend the features of generator defined in [18] by allowing it to select the needed HA-agents to maintain the state of the application, and include their information (e.g. domain name, or address) as instantiate-command-arguments to the state-aware application. I.e. upon instantiation, the state-aware application will be passed the information that identifies its HA-agent, from which it will acquire its state, and thereafter checkpoint its state.

- The upgrade-campaign generator [19] takes as input an AMF configuration file, and based on that generates the upgrade campaign with the required steps and procedures. The upgrade campaign is then fed to the SMF framework of the middleware. Based on the campaign, SMF will automatically install the application and update the system information model using IMM. In turn, IMM will notify AMF of the changes, and thereafter AMF will instantiate the application and maintain its availability. The Cloud user can use the same approach to perform future upgrades to the application with minimum service interruption.

Figure 7 illustrate the tool-chain in the integration-agent showing the workflow starting with the user requirements.

3) The HA-agent

The HA-agent is responsible for maintaining the state of the state-aware applications. The main purpose of its existence is to abstract the complex middleware APIs that are needed for checkpointing that state. The HA-agent is then a software component that implements two interfaces: (1) the interface that allows it to interact with the middleware, from this perspective it is fully integrated with the middleware, and (2) the interface through which the state is saved and retrieved by the state-aware application. For the later interface, we chose the Representational State Transfer (REST) interface [20], since it does not define an API library, but rather it is an architectural style that defines a client-server communication. From this perspective the state-aware component would be the client and the HA-agent would be the server. The choice of REST is driven by the fact that it is widely known by programmers, and already implemented by the majority of web-based applications. REST is agnostic of the programming language used to develop the application. Thus, it allows the server-client communication in a generic way since it uses unified resource identifiers (URIs), by that, we overcome the current limitation of the middleware that can only interface with a limited set of programming languages. Moreover, REST allows location transparency, where the Client does not need to know the physical address of the server which allows the replication of the HA-agents across the cluster, and use virtual addressing solutions (e.g. virtual IP) to access the agents. By this, we tolerate the failure of the HA-agent, since other agents can resume the same task. Figure 8 illustrates the interfaces of the HA-agent.

When using REST in the context of HTTP, a handful of methods (Get, Put, Delete, etc.) can be used for the communication between the client and the server. REST does not enforce a specific implementation of such methods, thus giving maximum flexibility to the application developers. Again on the server side, REST does not restrict the methods’ implementation. In short, the HA-agent is a generic component that accepts the application requests in a generic manner, processes them, and answers them, again in a generic manner, where the HA-agent does not need to have any information concerning the applications for which it is managing the state.

4) Summary of the HA solution

The presented HA solution can be summarized as follows: the user will provide basic information about the application and its availability requirements. The integration-agent component will process this information and produce an upgrade campaign, which is then fed to the middleware. The middleware will – at runtime – deploy the application and instantiate it with the proper arguments identifying its HA-agent. Once instantiated, the application will try to acquire its last saved state, using a generic REST interface. If
this is an initial instantiation, then such a state does not exist, and therefore the application will start executing from its default state. Subsequently, the application will start checkpointing its state, again using the REST interface. The HA-agent will checkpoint this state using the middleware checkpoint service. In case of failure, the middleware will detect it, and react to it. In the sequence diagram example shown in Figure 9, the recovery is a failover to a cold spare.

The spare will be instantiated on a different cluster node. Again, once instantiated the application will exhibit the same “state-aware behavior” and try to acquire its state with its unique ID (based on which the relevant URIs are defined). This time, the HA-agent will find a previous state corresponding to this ID, and return it to the application.

Figure 9. Interactions between the various components

IV. IMPLEMENTATION

The integration-agent we presented is based on the concepts defined in [18][19]. It implements algorithms with polynomial time/space complexity, and consumes negligible amount of resources. As for the HA-agent, we implemented it as a REST server application that implements a RESTful API that handles HTTP requests, and transforms them into checkpointing request based on the middleware APIs.

Our solution imposes a small performance footprint, as the middleware implementation itself [10] is a lightweight implementation that consumes moderate amounts of memory (roughly 15 MB of RAM) and CPU. As for the agents, the integration-agent is activated per request for adding (or removing) applications. Such requests are anticipated to arrive in low frequency and thus the integration-agent will typically be idle for most of the time. On the other hand the HA agent is continuously processing checkpoint saving/retrieving requests. To increase the capacity of handling the checkpointing request, additional HA-agents can be deployed. This deployment is cluster wide, as the communication is based on the address (IP:port, or domain name when using a local DNS) of the HA-agent. Using a load balancer, the HA-agent address is abstracted by a virtual IP, which is then converted by the load balancer to the real IP address of the agent. Since all the HA-agents are identical, they can be added and removed on demand in response to the increasing/decreasing traffic. Although this communication and processing of requests consumes resources, it is still negligible compared to virtualized fault tolerance solutions, where for each active VM, a dedicated standby is executing the same instructions.

The availability of the agents is also managed by the SAForum middleware as we included them in the AMF
configuration. Hence they are treated as applications that are monitored by the middleware, which will react to their failure and recover them. It should also be noted that the failure of the HA-agents does not impact the operation of the other applications (using these agents), it simply means that the checkpoint service is temporary unavailable.

V. DISCUSSION

In this paper we introduced an HA solution for the applications running in the Cloud. This solution is based on a general purpose, open-standard, platform-independent middleware [9], with an open-source implementation [10].

We consider the Cloud Middleware itself, a set of management applications that in turn can be made highly available using the same approach. As for the VMs, their availability can be handled by most of the current hypervisors (e.g., KVM, ESXi, etc.). Nonetheless, to maintain only one availability manager in the system, the HA middleware will also handle the VMs as stateless applications, that can be restarted/failed-over using the hypervisor capabilities. The communication between the HA middleware and the hypervisor is done using standard APIs, like Libvirt [17].

Cloud providers can deploy our HA solution at two levels: (1) at the Cloud infrastructure level, to maintain the availability of the Cloud middleware itself, and (2) at the PaaS (platform as a service) level, where the Cloud users are allocated VMs that already have the middleware and the agents deployed (or accessible). The Cloud users then only need to provide the integration agent with the needed basic information, and their applications can be automatically deployed in the cluster (or virtual datacenter) and made highly available.

This opens the discussion for offering HA-on-demand for the Cloud users. For instance not all applications in the Cloud need to be HA around the clock, certain applications perform critical computation at certain hours, e.g., applications analyzing the stock market based on data-mining typically perform the analysis after the market trading hours, and therefore only need to be instantiated and HA during specific hours, while other applications, potentially running on the same VMs, may need HA at different hours. The question is can we offer this high availability on-demand? The answer is yes, we simply retrieve HA the same way we add it, through another upgrade campaign that automatically removes the application from the middleware information model (without necessarily un-installing the application, unless it is needed to save resources). In short, upgrade campaigns can be scheduled in a timely manner, where the Cloud user simply needs to specify the times which he needs his applications to be instantiated and HA.

Finally, our solution is not dependent on any Cloud technology; therefore it can be ported to any Cloud provider, and by that avoiding any vendor-lock-in.

VI. RELATED WORK

The related work on high availability can be divided into two categories, the one based on virtualization, and the one similar to our approach that is based on checkpointing the state at the application level. We start with the latter.

HA-OSCAR [22], is a Linux based open source solution for HA. It is based on three main components: (1) IP monitoring using heartbeat, (2) service monitoring, where in case a service fails, it can be restarted a configurable number of times before failing-over, and (3) data synchronizazion, which is provided by a daemon that monitors the modifications made in particular directory trees and provides a replication service via rsync (Unix-based utility software for data synchronizazation). The authors of [23] proposed using this technology in the Cloud. The main shortcomings of this solution is that it does not include the capabilities for runtime upgrade for the system, and that it is not based on standardized specifications and information models that facilitate the large-scale system management as in [10]. Hence, it lacks the manageability aspect which is one of the main aspects in Cloud computing.

In [24], the authors propose an architecture for the application HA in the Cloud. The idea is based structuring the application into request receiver and request processor nodes. The request receivers will dispatch the requests to the processors. In case a processor fails in servicing a request, it will be restarted by the receiver, and the request can be reassigned to another processor. The architecture is generic, and not based on existing technologies, thus the application needs to implement the needed availability mechanisms, instead of using a more modular approach that benefits from existing HA technologies. Moreover this approach only applies for request based workloads, which is not the case for all applications, for instance, application performing media broadcast do not necessarily function based on user request, but rather on a predefined workload, yet they still need to checkpoint the broadcasted stream position, in order to avoid re-broadcasting the entire stream in case of failure. Furthermore the approach heavily impacts the development process of the applications through explicit structuring of functionalities. In a similar work [25], the authors propose using integrated checkpointing algorithms; nevertheless the work suffers from the same limitations.

In Section II we discussed the limitations of virtualization-based solutions for HA. Here we survey relevant techniques for achieving HA using virtualization.

Commercial solutions such VMware-HA [26] can detect the failure at the VM level and restart it on the same or different machine. This solution does not provide service continuity; thus, VMware-FT [27] is presented as a solution that supports fault-tolerance. With VMware-FT a hot standby of the VM runs in lock-step mode, where it mimics the active VM. Nonetheless, the failure of the applications running in the VM is not detected, from this perspective, the solution provides service-continuity in case of hardware or VM failure (with a hefty price in terms of resource consumption), but not is case of application failure. To remedy this issue, Symantic [28] proposes a solution, based on VMware products, where the applications are monitored, however this commercial solution applies for a specific list of supported applications, and does not support HA in a
generic way. Moreover it does not support the dynamic upgrade in a highly available fashion.

In [29], the authors present an approach to achieve high availability also based on virtualization. Unlike [27], in this approach we do not have hot standby VMs, instead we have cold ones. The solution is based on having: snapshot agents, that, through the hypervisor, periodically take snapshots for the running “active” VMs, and snapshot managers that receive the snapshots collected by the snapshot agents and merge them with their respective parents (latest snapshot) as soon as the snapshots are received. When the active VM fails, the standby is requested to play the last saved snapshot. While this approach causes less overhead than [27], it still consumes a significant amount of resources. Moreover, the same limitations apply, whereas the approach does not target the availability at the application level, but rather at the VM level, therefore the authors do not discuss the failure of the applications in the VMs, and how it is handled.

VII. CONCLUSION AND FUTURE WORK

We presented a solution for achieving HA at the application level using a specialized middleware for HA. In order to adapt the middleware to the more agile and flexible environment of Cloud computing, we defines two agents, that can automate the integration with the middleware. In order to relieve the developer of HA applications from implementing complex APIs, we based our solution on the REST architecture.

The near-future work includes profiling the different states for different stateful applications in terms of size, format, update frequency etc. and experiment with the impact of checkpointing such states on the Quality of Service (QoS).

We will target the issue of intra data-centers HA management and thus stepping into different availability zones/regions, and the effect this has on service availability and other QoS.

ACKNOWLEDGMENT

This work is partially supported by Ericsson Research and the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[26] VMWare vSphere High Availability (HA), URL: http://www.vmware.com

[27] VMWare vSphere Fault Tolerance (FT), URL: http://www.vmware.com


Published in the proceedings of the 2-13 IEEE Sixth International Conference on Cloud Computing, pages 778-785, CLOUD 2013, Santa Clara, California (http://dl.acm.org/citation.cfm?id=2515046)