Middleware-managed high availability for cloud applications

Ali Kanso
IBM T.J. Watson Research, USA
Email: akanso@us.ibm.com

Abdelouahed Gherbi*
Department of Software and IT Engineering,
Ecole de Technologie Superieure (ETS),
Montreal, Canada
Email: abdelouahed.gherbi@etsmtl.ca
*Corresponding author

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

Abstract: High availability is a key non-functional requirement that software and telecom service providers strive to achieve. With the ongoing shift to cloud computing, the challenges of satisfying the high availability requirement become more arduous, as the cloud introduces additional features, such as on-demand access, scalability, virtualisation, which add to the complexity of the high availability solution. In this article, we target the issue of achieving high availability for the software applications running in the cloud. We first benchmark two of the most prominent middleware implementations for HA. We then build on the most responsive one to present our solution to address the complexity issue of applications high availability deployed in the cloud. Finally, we discuss a quantitative and qualitative assessment of the overhead of our proposed solution.

Keywords: high availability; cloud platform; HA middleware; state-aware applications; runtime integration; high availability as a service; REST architecture.


Biographical notes: Ali Kanso is Cloud Software Engineer at IBM. He was previously a Senior Researcher at Ericsson working on cloud technologies and highly available systems. He earned his MSc and his PhD in Computer Engineering in 2008 and 2012, respectively, from Concordia University in Montreal, Canada. He has several patents granted/pending in the field of distributed highly available (HA) system design and automation, as well as a number of publications in conference proceedings, journals and book
chapters. His research interests include software modelling and design, middleware-based HA integration, cloud computing and autonomous self-managing systems.

Abdelouahed Gherbi is an Associate Professor at the Software and IT Engineering Department in the École de Technologie Supérieur (ETS) in Montreal, Canada. He received his PhD in Computer Engineering from Concordia University, Canada. He has spent one year as Visiting Researcher at the Defense Research and Development Canada DRDC in Valcartier, Quebec. His research interests include mainly model-driven software engineering, modelling and analysis techniques for real-time and critical software systems, software performance, high availability and security.

Yves Lemieux is a Research Engineer at Ericsson. His main interests are in 3GPP-based end-to-end quality of service and virtualisation for cloud computing. He received his MS in Computer Engineering from École Polytechnique de Montreal. He has several patents and publications in the fields of cellular system synchronisation selection, network resiliency, and long-term evolution core network congestion control, among others.

This paper is a revised and expanded version of a paper entitled ‘Achieving high availability at the application level in the cloud’ presented at IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, 28 June–3 July 2013.

1 Introduction

Cloud computing is based on the notion of offering compute, network and storage resources as services offered on-demand (Lehman and Vajpayee, 2011). As such, the cycle from development to staging to production can be significantly simplified, when the effort is focused on the development of the applications and services instead of the deployment environment and maintenance. The high availability (HA) of the applications running in the cloud is not only a factor of the robustness of the applications, but also a factor of the reliability of the cloud infrastructure itself. Therefore the question of whether the cloud is ready to host critical applications was raised (Leavitt, 2009; Gagnaire et al., 2012), since in order to assure the HA of the applications, we need to handle failures at both the application layer and the cloud infrastructure layer.

HA is achieved when the system is capable of providing its intended functionality 99.999% of the time (also known as five nines) (Gray and Siewiorek, 1991). HA is mainly based on four pillars:

1. Monitoring: without monitoring, failures will remain undetected, and thus no recovery will be triggered.
2. Redundancy: failures are bound to occur on the software, hardware and network level, therefore redundancy is a must in order to avoid single points of failure.
3. Availability management: the system must have the intelligence to coordinate the recovery after a failure is detected. Moreover, when the recovery fails, an escalation policy must be enforced to ensure that the system regains its functional status.
4. Checkpointing: stateful applications should not lose their state in case of failure, i.e., after the recovery, the application must resume executing from its last healthy state.
Any HA solution must incorporate these four pillars, whereby it should support the distributed deployment across a cluster of nodes that intercommunicate to maintain the system’s HA.

The user’s applications can be deployed in the cloud using two deployment models; the first one is based on using the platform as a service (PaaS) (e.g., Severance, 2009), where the cloud providers offer an isolated deployment of the application within a sandbox, i.e., an execution environment with bounded scope. Applications running in the sandbox have limited visibility outside the sandbox, and can only communicate with the outside world using the PaaS communication services. In case of failure, the entire sandbox can be failed over or restarted. However this deployment model is not suitable for all applications; it is intended for the applications developed within the cloud provider PaaS and using its application programming interfaces (APIs). The second deployment model is based on using the infrastructure as a service (IaaS) (e.g., Barr et al., 2011), this model is generic, and supports the deployment of a wide range of applications, without requiring changes in the application code. However, cloud provider offering IaaS are not concerned with HA at the application level, instead they focus on the HA of the virtual machines (VMs) hosting the client’s applications, in other terms, the health and the state of the applications are not managed by the IaaS provider.

In this paper, we describe our approach to achieve HA at the application level in the context of a cloud deployment. Our solution is based on using agents that can abstract the integration of HA into the applications. Our approach can be integrated at both: the IaaS and the PaaS levels. In particular, we extend further our previous work with the following contributions. We extensively benchmark the performance of the existing HA solution namely Pacemaker and OpenSAF. We extend our agent-based proposed HA solution with the checkpointing agent to enable maintaining the state of state-aware applications. Finally, we provide the details of the implementation and discuss the results of the testing and evaluation of our proposed solution.

This paper is organised as follows: in Section 2, we present the background of our work illustrating the different approaches for achieving HA with their advantages and limitations. In Section 3, we benchmark the performance of two HA solutions for clustered applications, namely Pacemaker and OpenSAF, in order to compare the responsiveness. In Section 4, we present the details of our approach to achieve HA at the application level. In Section 5, we outline our solution implementation and present the testing results. We discuss further the perspectives of adoption of our solution in Section 6. In Section 7, we survey the existing literature and finally we conclude our paper in Section 8.

2 Background

HA is a non-functional property that proves to be challenging for application developers to implement. In fact the developers are expected to focus on the logic of the applications and not on its availability. Instead, this task can be achieved by employing specialised solutions to handle the systems availability. HA solutions can basically be divided into two categories. The first one is based on virtualisation whereby a distributed hypervisor is used (VMwareFT, 2013; Chan and Chieu, 2012). The second category relies on the use of specialised middleware which are responsible for maintaining the HA of the applications and their states (OpenSAF, 2013; Pacemaker, 2016).
2.1 HA solutions

Virtualisation-based HA solutions assume that the applications are running in virtualised execution environments (e.g., a VM). Using the capabilities of the hypervisor, the VMs are monitored and their state is replicated such that in case of failure the VM can resume executing from the same state (VMWareFT, 2013). Such solutions require no modifications to the application’s code or design; however they suffer from several limitations. First, they offer protection against hardware and VM failures while failures at the software applications level can remain undetected. 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. Moreover, such solution must be deployed on identical hardware (among the active and the standby) with a specific set of supported CPUs. In short, using visualisation 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 9s of availability for the user’s applications in their service level agreement (SLA) (Calzolari, 2009). Specialised middleware for HA offer 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 provisioning. Several proprietary solutions (e.g., Jews and Ahmad, 2008; Oracle, 2013) 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 (SA Forum) (SAF-AIS, 2013) defined a set of specifications for a platform-independent, standardised middleware solution for HA. An open-source implementation of these standards such as OpenSAF (2013) can deploy and manage the applications and maintain the availability of their services. Pacemaker (2016) is another Linux-based project for maintaining the HA of the applications across a cluster of machines.

2.2 The Pacemaker middleware

Pacemaker is a cluster resource manager, capable of performing failure detection and synchronise the recovery across a set of machines (be it physical or virtual machines). Pacemaker relies on a stack of other components to achieve its functionality, including:

- Corosync (2016): a group communication system with virtual synchrony guarantees for replicated state machines (note that heartbeat coupled with CCM could be used as an alternative to Corosync).

- Haas (2012): are specialised scripts that allow Pacemaker to manage services in a generic way without knowing the details of the services. A service can be any software component of a system configuration that needs to be executed. Resource agents are implemented according to the Linux standard base, and must expose basic
management functionalities to Pacemaker, such as starting, stopping, and monitoring a specific service. There is a wide range of existing resource agents that manage a wide range of services (such as major databases, and http servers).

Internally, Pacemaker uses the cluster information base (CIB) to store the cluster configuration. This configuration is kept in sync across the cluster using Corosync. On each cluster node, and local Pacemaker resource management daemon is running in order to enforce the policy engine instructions forwarded by the cluster resource manager. The policy engine periodically verifies the cluster configuration against the cluster status and initiates the required actions. Pacemaker supports various redundancy models including active/active, active/passive and N+1. Pacemaker does not include a framework for the HA upgrade of the software resources under its control, nor does it natively support a checkpointing mechanism.

2.3 SA Forum middleware

The SA Forum 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. The availability management framework (AMF) (SAF-AMF, 2013) constitutes the core of the SA Forum 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/standby 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 takeover in case of failure. 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 characterised 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 modelling language (UML) class diagram specified in SAF-AMF (2013). The configuration objects must be described according to a standardised machine-readable extensible markup language (XML) schema (SAF-IMM, 2013). 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. The checkpoint service (SAF-CKPT, 2013) 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 synchronisations between the replicated checkpoint objects (e.g., synchronous and asynchronous) [software management framework (SMF)]. Software systems are
constantly evolving, especially with development processes like agile (Pressman, 2005) where new features and functionality are constantly being added. Interrupting the services for the purpose of upgrade and maintenance is not acceptable in HA systems, where the allowed service downtime distributed over the year is roughly five minutes. SMF (SAF-SMF, 2013) defines a framework that enables the software upgrade in a structured manner with minimum service interruption. The upgrade is based on a standardised 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 minimise 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 standardised XML schema (SAF-SMF, 2013). 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 rollback to the original configuration. SMF also offers administrative operations to execute, suspend, rollback and commit an upgrade.

Figure 1 AMF configuration example (see online version for colours)
The SA Forum middleware is designed for massive data centres where it would handle a large number of software/hardware components. The information model of such systems tends to be significantly complex, with all the information about the AMF configuration, upgrade campaigns, checkpoints and others. To maintain a consistent view and ease the manageability of this model, the IMM middleware service is defined. IMM provides the needed interface for the other middleware services/frameworks to act on the information model, and at the same time it allows the system administrator to monitor the system state, and perform various administrative operations (Figure 3). IMM exposes two types of interfaces, the object-manager interface, and the object implementer. For instance when upgrading software component ‘comp1’, SMF cannot lock the component and shift its workload to its standby counterpart, as AMF is the object implementer responsible for these actions, from this perspective SMF is the object manager, which communicates with AMF through IMM to lock the component.

Figure 2  Upgrade-campaign structure (see online version for colours)

Figure 3  The IMM middleware service (see online version for colours)
3 Benchmarking existing HA middleware solutions

Both Pacemaker and OpenSAF are capable of managing the HA of a clustered application. In order to compare the responsiveness of both solutions, we have benchmarked their performance in managing a stateful video streaming application, namely the Video LAN Client (VLC) (2016). We used the active/standby redundancy model in Pacemaker, and its 2N counterpart with OpenSAF. We performed two types of testing by:

1. inducing a failure with a local restart recovery option
2. inducing a failure with a failover recovery option.

For Pacemaker, we used distributed replicated block device (DRBD) to maintain the state of the application. With OpenSAF, we tested the application in two modes:

1. without integrating any of the SAF APIs (application programming interface) into its code
2. with slight modifications to the application code to include the SAF libraries, also known as the service availability aware (SA Aware) approach.

Pacemaker offers a configuration attribute that allows us to change the monitoring frequency; hence we experimented with different monitoring intervals varying from one second to one millisecond.

Figure 4 The restart recovery comparison (see online version for colours)

We performed our experiments in a virtual environment running virtual box on servers with Intel Core i7 and 16 GB of RAM. Each experiment is the average of 25 samples. Figure 4 illustrates the comparison results of the restart recovery, we distinguish between the reaction time, i.e., the time needed for the middleware to detect the failure, consult its configuration (or policy) and issue the proper commands to start the recovery. And the recovery time, which is the time needed to successfully execute the recovery, i.e., cleaning up the faulty process and re-instantiating it locally (or failing it over, as in the next experiment). OpenSAF proved to be highly efficient with SA Aware
applications. However, with non-SA Aware applications where the operating system facilities are used to detect the processed failure, the recovery times were significantly higher. As for Pacemaker, lowering the monitoring interval below the 1-second margin, we improved the recovery duration. It should be noted that the Pacemaker documentation recommends setting the monitoring interval above 10 seconds; however, we believe for the sake of the fairness of the comparison, we had to lower this interval.

In the following experiment shown in Figure 5, we have compared the failover recovery durations (with the minimum monitoring interval of Pacemaker). A failover can be a preferred recovery if the restart of the application takes longer than switching to the standby, assuming the state is already in sync between the active and standby components. However, in case of the node (or host) failure, failover is the only recovery option that can ensure minimum service interruption. OpenSAF significantly outperformed Pacemaker with SA Aware applications and was slightly better even with the non-SA Aware applications.

Our final experiment compares the CPU resource consumption of both middleware. Pacemaker resource consumption significantly increases with smaller monitoring intervals.

**Figure 5** The failover recovery comparison (see online version for colours)

With OpenSAF the CPU consumption remains moderate throughout all the experiments as illustrated in Figure 6.

In our benchmarking exercise, we chose generic settings and configurations for both middleware, with the exception of significantly lowering the monitoring interval of Pacemaker. Our experiments are not exhaustive, as there are still other redundancy models to be compared such as active/active, and different mechanisms to be tested such as the escalation policies in case of the successive failures of the recovery. Our main objective was to demonstrate that both middleware technologies are perfectly capable of providing HA, with a moderate amount of resource consumption. In conclusion, when applications implement the SAF APIs, OpenSAF can easily outperform Pacemaker. However implementing the SAF APIs would require a significant amount of effort and expertise as we will elaborate in Section 4. Overall, OpenSAF is a more comprehensive middleware for HA. It implements checkpointing, reliable messaging and software management mechanisms for rolling upgrade that Pacemaker lacks. Moreover it is based
on well-defined specifications by the SA Forum. Hence we chose the SAF middleware for our proposed HA solution in the cloud.

**Figure 6** The CPU utilisation comparison (see online version for colours)

![CPU utilisation comparison diagram](image)

### 4 Achieving HA at the application level

Our solution for achieving HA at the application is based on deploying the SA Forum middleware in the cloud. Nevertheless, the SA Forum 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.

#### 4.1 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 organisation of resources available on demand (Figure 7). 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 virtualisation layer (e.g., Xen, 2013), or an operating system-based virtualisation as in KVM (2013). 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 virtualisation 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.

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 (2013). 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 highly available, 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 7 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.

Figura 7 Proposed cloud architecture (see online version for colours)

4.2 Integrating the applications with the middleware

There are three different approaches to integrate applications with the SA Forum middleware (SAF-AMF, 2013):

- **Non-SA Aware**: this approach is 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.

- **SA Aware**: this is potentially a more intrusive approach that requires the application to implement the API required by the middleware to enable the bidirectional 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-managed high availability for cloud applications

The middleware cannot control the lifecycle 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 with minimum interruption. Expecting cloud users to invest the time and effort to acquire the needed domain knowledge and define these middleware artefacts (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.

4.3 HA integration solution

In order to address the complexity issues of the HA integration, we define three agents (Figure 8):

1. The HA agent which accepts the cloud user input and translates it into information to be passed to the other agents
2. The integration agent that can dynamically modify the system information model to let the middleware be aware of the newly added applications
3. The checkpoint agent that will interact with the state-full applications in order to checkpoint their state.

Figure 8 The HA integration solution (see online version for colours)

4.3.1 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 (since the current state of execution for the application would be lost). 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 behaviour as follows: upon instantiation, the state-aware application will try to acquire its last saved state from a checkpoint 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 9. 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.

4.3.2 HA agent

The cloud user interacts directly with the HA 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. The HA agent will transform this information into an input that is readable by the other agents, and orchestrate the HA integration into the user’s applications.

Figure 9 The behaviour of a state-aware application (see online version for colours)
4.3.3 Integration agent

The integration agent receives its input from the HA agent. This information is fed to two generators:

- The configuration generator which is an extension to the one we defined in our previous work. It takes the user input and automatically generates the configuration XML file. We extended the features of generator by allowing it to select the needed Checkpoint-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. That is, upon instantiation, the state-aware application will be passed the information that identifies its checkpoint agent, from which it will acquire its state, and thereafter checkpoint its state.

- The upgrade-campaign generator 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 10 illustrates the tool chain in the HA agent and integration agent showing the workflow starting with the user requirements.

Figure 10  Integration-agent workflow (see online version for colours)
4.3.4 Checkpoint agent

The checkpoint 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 checkpoint agent is then a software component that implements two interfaces:

1. the interface that allows it to interact with the middleware, as such, it is fully integrated with the middleware
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 (Fielding, 2000), 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 checkpoint 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 checkpoint agents across the cluster, and the use of virtual addressing solutions (e.g., virtual IP) to access the agents. By this, we tolerate the failure of the checkpoint agent, since other agents can resume the same task. Figure 11 illustrates the interfaces of the checkpoint 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 checkpoint 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 checkpoint agent does not need to have any information concerning the applications for which it is managing the state.

Figure 11 Checkpoint-agent interfaces (see online version for colours)

The checkpoint service of the middleware offers several options to save the checkpoint according to different preferences. Such preferences include the replication mode of a checkpoint object (co-located/non-co-located with the application), the synchronisation
mode of the checkpoint replicas (synchronous/asynchronous), and the retention time of
the checkpoint object before being discarded after not being opened by any process.
While not every application developer may have the interest or the knowledge to use
these features, we do not want our solution to impose limitations on which features to be
used. Therefore we defined two levels for our REST resources:

1. Using the first level the client application will send a simple request to the
checkpointer including only the content of the state that needs to be
checkpointed. The checkpoint agent will take this state and checkpoint it with the
checkpoint service using default preferences. This approach favours simplicity and is
aimed for applications that require basic checkpointing features.

2. For the applications that require specific checkpoint features offered by the
middleware, we implemented the checkpoint agent in such a way that can parse
requests that, in addition to the content of the state, include the preferences according
to which the checkpoint object (storing the state) must be configured.

Figure 12  Interactions between the various components (see online version for colours)
4.3.5 Summary of the HA solution

The presented HA solution can be summarised 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 checkpoint 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 store 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 12, 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 behaviour’ and try to acquire its state with its unique ID (based on which the relevant URIs are defined). This time, the checkpoint agent will find a previous state corresponding to this ID, and return it to the application.

5 Implementation and testing

The integration agent and the HA agent have been implemented using a model-driven approach, where the middleware configuration is modelled as a UML class diagram. Based on this model the input files, the upgrade campaign files and the XML configuration file are automatically generated. We used Eclipse modelling framework (EMF) (Steinberg et al., 2008) as our modelling infrastructure.

As for the checkpoint 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 (OpenSAF, 2013) is a lightweight implementation that consumes moderate amounts of memory (roughly 15 MB of RAM) and CPU. As for the agents, the integration agent and the HA agent are 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 Checkpoint agent is continuously processing checkpoint saving/retrieving requests. To increase the capacity of handling the checkpointing request, additional checkpoint 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 checkpoint agent. Using a load balancer, the checkpoint 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 checkpoint agents are identical, they can be added and removed on demand in response to the increasing/decreasing traffic.

In order to evaluate the overhead (in terms of processing delay) imposed by using the rest interface, we compared the read/write time of checkpoint object with and without the use of our checkpoint agent. In order to include the state size of wide range of applications we varied the checkpoint size between 1 kb and 2 MB. In our experiment,
the agents and our state-aware application are deployed in VMs with 2 GB of RAM and 2 CPU cores at a speed of 2.9 GHz. Our case study is a video streaming application implemented in Java, interacting with a checkpoint agent implemented in the C language. Note that the results shown below can vary in different contexts with different case studies; the objective here is to get a general idea of the overhead of using the checkpoint agent.

Our test results show, as depicted for example in Figure 13 that the reading overhead is in the magnitude of a few milliseconds. Note that this is the overhead causing an extra outage during the recovery period, since the application will fetch its state after the failover. Since the checkpointing should not be blocking the flow of execution of an application (e.g., in our case study, checkpointing is done in an independent thread of execution), the writing overhead should not be noticeable to the application. In summary, resource consumption, the communication and processing of requests, is still negligible compared to virtualised 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 SA Forum 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 checkpoint agents does not impact the operation of the other applications (using these agents), if the agents are behind a load balancer, the failure will be unnoticed to the application, otherwise the failure simply means that the checkpoint service is temporary unavailable, until the agent is repaired.

Figure 13  Checkpoint reading overhead (see online version for colours)
6 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 (SAF-AIS, 2013), with an open-source implementation (OpenSAF, 2013). 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/failover using the hypervisor capabilities. The communication between the HA middleware and the hypervisor is done using standard APIs, like Libvirt (2013).

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
2. at the PaaS 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 data centre) 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 analysing the stock market based on data mining typically perform the analysis after the market trading hours, and therefore only need to be instantiated and remain highly available during specific hours, while other applications, may need HA at different hours. The question is can we offer this HA on-demand? The answer is yes, we simply remove HA the same way we add it, through another upgrade campaign that automatically removes the application from the middleware information model (without necessarily uninstalling the application, unless it is needed to save resources), and instructing the checkpoint-agent to stop accepting requests from this application. 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 based on open-source implementations and does not dependent on any cloud technology; therefore it can be ported to any cloud provider, and by that avoiding any vendor-lock-in.

7 Related work

The related work on HA can be divided into two categories, the one based on virtualisation, 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 (Haddad, 2003), is a Linux-based open source solution for HA. It is based on three main components:
Middleware-managed high availability for cloud applications

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
3. Data synchronisation, 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 synchronisation).

Thanakornworakij et al. (2012) proposed using this technology in the cloud. Pacemaker (2016) is a cluster resource manager that evolved out of the Linux-HA project. Pacemaker is composed of mainly four components:

1. The CIB that uses XML to represent both the cluster configuration and the runtime state of all the resources in the cluster; a resource can be any cluster service, e.g., an Ethernet network card IP address or a software application
2. The policy engine which uses the information managed by the CIB to compute the ideal state of the cluster and how it should be enforced
3. The cluster resource management daemon, execute the list of instructions generated by the policy engine
4. Finally, pacemaker supports the shoot the other node in the head as a fencing mechanism to clean up and isolate failures.

The main shortcomings of these solutions is that it does not support the runtime upgrade of the system in an HA manner, and they are not based on standardised and platform-independent specifications and information models that facilitate the large-scale system management as in OpenSAF (2013). Hence, it lacks the manageability aspect which is one of the main aspects in cloud computing.

Anand (2012) proposes 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 re-assigned 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, applications 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, Singh et al. (2012) propose using integrated checkpointing algorithms; nevertheless, the work suffers from the same limitations.

In Section 2, we discussed the limitations of virtualisation-based solutions for HA. Here we survey relevant techniques for achieving HA using virtualisation. Commercial solutions, such as VMWareHA (2013), 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, VMWareFT (2013) is presented as a solution that supports fault-tolerance. With
VMWareFT 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 in case of application failure. To remedy this issue, Symantec (2013) 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 Chan and Chieu (2012), the authors present an approach to achieve HA also based on virtualisation. Unlike VMWareFT (2013), 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 VMWareFT (2013), it still consumes a significant amount of resources (since the state of the entire VM and not just the application is saved). 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.

8 Conclusions and future work

In this paper, we presented our approach for achieving HA at the application level using a specialised middleware for HA. In order to adapt the middleware to the more agile and flexible environment of cloud computing, we defined software 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. As future work we will target the issue of inter-data centres HA management and thus stepping into different availability zones/regions, and the effect this has on service availability and other QoS such as the ‘greenness’ of our solution. As such additional constraints will be added to maintain a greener cloud with reduced resources and energy consumption without violating the HA constraints. This introduces a multi-objective constraint satisfaction problem that we intend to study and solve.

References


