Adaptive and Scalable High Availability for Infrastructure Clouds

Stefan Brenner, Benjamin Garbers, Rüdiger Kapitza

Adaptive and Scalable High Availability for Infrastructure Clouds

Stefan Brenner, Benjamin Garbers, and Rüdiger Kapitza

TU Braunschweig, Germany
brenner@ibr.cs.tu-bs.de, b.garbers@tu-braunschweig.de,
rrkapitz@ibr.cs.tu-bs.de

Abstract. These days Infrastructure-as-a-Service (IaaS) clouds attract more and more customers for their flexibility and scalability. Even critical data and applications are considered for cloud deployments. However, this demands for resilient cloud infrastructures.
Hence, this paper approaches adaptive and scalable high availability for IaaS clouds. First, we provide a detailed failure analysis of OpenStack showing the impact of failing services to the cloud infrastructure. Second, we analyse existing approaches for making OpenStack highly available, and pinpoint several weaknesses of current best practice. Finally, we propose and evaluate improvements to automate failover mechanisms.

Keywords: Reliable Cloud Computing

1 Introduction

Cloud computing is a big paradigm shift affecting almost all kinds of IT-provided services. Both, for private and public cloud settings, people increasingly consider shifting their applications to clouds. This naturally includes even critical applications, and thus, critical data [2,10]. As a result, reliability and availability of cloud services become more and more important. Besides cost efficiency and scalability, these properties are significant reasons for offloading applications to clouds for many cloud customers.

Recent events show, however, even professional clouds still struggle with outages and failures [15,11]. Hence, support for high availability (HA) of Infrastructure-as-a-Service (IaaS) clouds has to be built from scratch, deeply integrated in the cloud software stack.

In order to handle the aforementioned needs for reliability and availability in infrastructure clouds, we present adaptive and scalable HA for the OpenStack[1] infrastructure services. We consider typical failures like host, network, disk, and process crashes.

Focus is set on HA of infrastructure services, so the actual virtual machines (VMs) running on compute nodes are not covered. If this is necessary, there has been research that combines live migration with failover techniques for

[1] https://www.openstack.org

single VMs [8] and VM replication [1], or as a cloud-integrated service for highly available services [19].

In this work, first, we analyse the generic IaaS cloud anatomy, specialise on OpenStack later on, and show the impact and consequences of failing services and components. Second, we outline existing approaches and best practices for HA and identify their weaknesses and open problems. Third, we propose several approaches for the improvement of these mechanisms, tackling the discovered problems. Our approaches are based on replication of services and load balancing. Besides HA, they also improve the scalability of stateless infrastructure services when the load increases. Similarities between several infrastructure clouds, in combination with our generic approaches, allow universal application of our proposed principles.

There are existing approaches using passive replicas [5]. In that case, passive service endpoints are made active on failures of an active replica. In contrast to that, there are approaches using active replicas only, balancing the request load across all of them. This allows failure masking, because requests are only forwarded to correct replicas. At most one request being in flight while a failure occurs can be lost in this setup. Hence, we base our approach on this mechanism as well. Our contribution improves on existing ones by automatic failover procedures that do not require manual intervention and modifications that allow to handle more fine-grained failures. Furthermore, conflicts that may happen on failures when existing approaches are used, are solved and prevented with our improvements.

In our failure analysis, we identified that OpenStack's networking component *Quantum*[2], that manages and maintains virtual networks interconnecting VMs, runs into conflicts when only some processes of Quantum fail while other components keep running. To solve this problem, we provide a monitoring-and-cleanup solution that prevents the arising conflicts and implements automatic failover that does not require any manual intervention by an administrator.

This paper starts with an architecture overview of OpenStack in Section 2, with special focus on the Quantum architecture, because it is most relevant for the contribution of this work. Then, we provide a detailed failure analysis of OpenStack in Section 3. Next, we show current best practices making OpenStack highly available and their shortcomings in Section 4, followed by Section 5 that describes the improvements of our approach. In Section 6 we measure the performance of our approach compared to existing approaches and show the recovery times on component's failure. Finally, we conclude our work in Section 7.

2 Anatomy of an OpenStack-based IaaS-Cloud

In the following section, we describe the anatomy of an IaaS cloud using OpenStack as an example. OpenStack is only one open source implementation of an IaaS cloud system, and other implementations like Eucalyptus [14] and OpenNebula [12] are quite similar in terms of architectural aspects [16]. Hence, our

[2] Will be renamed *Neutron* in the future

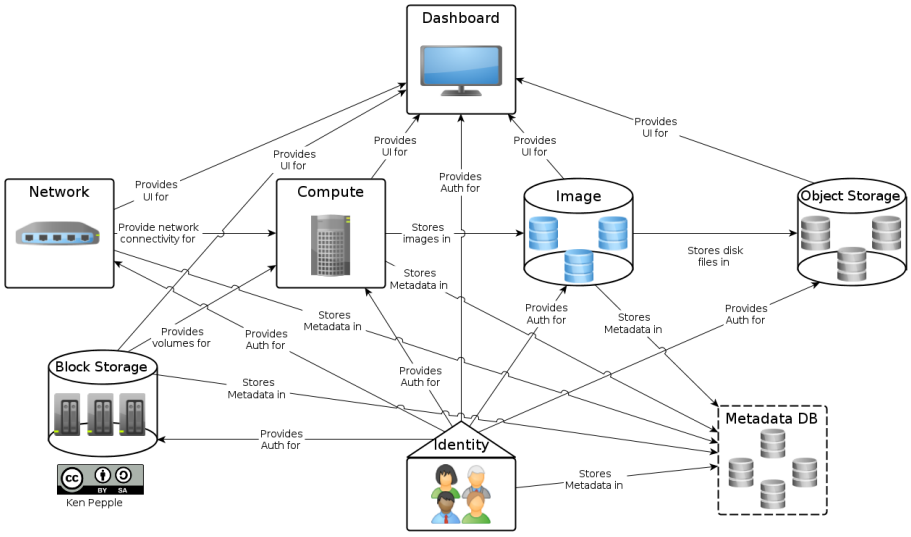


Fig. 1: OpenStack Architecture Overview based on [3]

generic approaches solving the HA questions, can be applied to them with small adaptions as well (see Section 5).

OpenStack-based IaaS clouds usually consist of several loosely coupled services, i.e. OpenStack components, that cover various cloud infrastructure building blocks like identity management, storage, networking, as well as the actual virtualisation fabric and the user interface. Figure 1 illustrates how the OpenStack services are interconnected and interacting with each other. All components interact with each other via a database and a messaging bus system. The database is mostly used for storage of relevant meta information about VMs and networks and can be implemented by a relational database system. As part of the default configuration MySQL is utilised here. The responsibilities of the message bus, cover the exchange of state information about compute nodes and scheduling tasks for example. Usually, the message bus is implemented using RabbitMQ[3], an AMPQ-compliant messaging system.

While OpenStack components interact with each other using a central database and the message bus, in order to provide functionality to the users, REST APIs are provided by the individual components. These APIs are widely compatible to the well-known Amazon Web Services[4].

Two important components of OpenStack are *Nova*, that provides the actual cloud computing fabric controller, and *Swift*, which provides object storage. These two components have been part of OpenStack from the very beginning of the development. Apart from *Cinder*, that provides block storage to VMs,

[3] `https://www.rabbitmq.com`
[4] `https://aws.amazon.com/`

further components are *Keystone* for identity and user management, *Glance* for virtual machine image management, and Quantum that maintains the (oftentimes completely virtual) network infrastructure interconnecting the VMs and real networks.

Most OpenStack components are divided up into several individual services working tightly together. Nova for example consists of the Nova scheduler service, the Nova compute service and several others. Similar to this, many components including Nova contain an API service. Clearly we have to consider all services of a component when making the OpenStack infrastructure highly available. Many of these services are stateless, which simplifies replication and makes them suitable for load balancing.

One of the most sophisticated OpenStack components is Quantum. It contains several services which are partly not stateless, and thus, sophisticated to make highly available because of consistency problems. The Quantum server is the main subcomponent and provides the management interface. It also controls the other subcomponents of Quantum, namely the DHCP-Agent, the L3-Agent and the Plugin-Agent. The agents in turn interact with system mechanisms for providing DHCP services to VMs, maintaining layer three networking capabilities, and managing firewalling.

Quantum supports several deployment modes. The simplest one is called *flat network*, bridging the VMs directly to an external network. Next, there is the mode *provider router with private networks* that isolates the VMs of one tenant in individual networks routed by an OpenStack networking router. The most flexible mode is called *per-tenant routers with private networks*. In this mode, each tenant can create virtual networks and routers and interconnect them. Virtual networks are isolated using Linux network namespaces, such that tenant's private networks could even have overlapping IP namespaces. In this work, we focus on the OpenvSwitch[5]-based variant of this deployment mode, that uses a OpenvSwitch Plugin-Agent running on the network node, that manages the virtual networking.

3 Vulnerability Analysis

The focus of this work lies on the analysis of the general cloud infrastructure services and its possible failure scenarios. HA for actual tenant VMs is out of scope for this work and has to be considered separately, for example by use of VM replication [8]. Special attention is payed to database failures and failures of virtual network components involved in stateful Quantum. Other stateless services can be grouped together and approached the same.

We further divide this section up into failures of building blocks the OpenStack services rely on, such as the relational database and the messaging system. Apart from that, we consider failures of the actual OpenStack services itself, for example the identity management service *Keystone*.

[5] `http://openvswitch.org/`

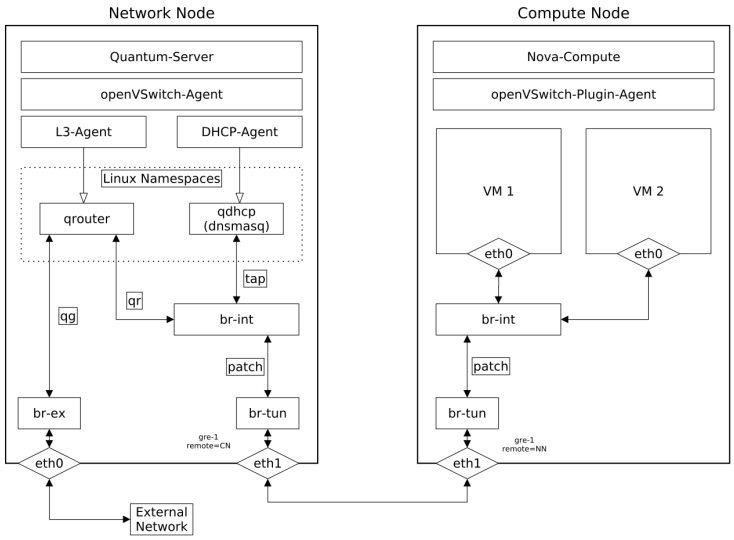


Fig. 2: Quantum with OpenvSwitch

In this work we consider crash-stop failures, complete host failures or individual process failures. If complete hosts fail, OpenStack loses its connections to all services running on that host, and has to offload these tasks to other hosts. More fine-grained are individual process failures, because only the specific failed process has to be handled by the failover procedure.

In our analysis we distinguish three different failure severity categories: *low*, *medium* and *high*. Failures, that render the whole infrastructure unusable are considered highly severe. Low severity of failures means the failure does not affect big parts of the infrastructure but mainly influences some VMs on a single physical machine. In between there are medium severe failures with major impact on the cloud infrastructure and a reduction of cloud functionality. Table 1 gives a quick overview over our analysis that is detailed in the remainder of this section.

3.1 Failures of OpenStack Building Blocks

Database failures: The relational database is one of the core parts of an OpenStack cloud, as it stores the configuration and metadata. Most services directly rely on a relational storage backend in order to function properly. A failure of the database is considered highly severe, as it will cascade to the other OpenStack services.

Messaging System: The messaging system in OpenStack is another vital core component. It is used for internal communication of services directly or indirectly responsible for configuring and running VMs. Remote procedure calls (RPCs) are used for direct communication between OpenStack components, whereas tasks like scheduling requests are distributed within one component following a

push-pull pattern. Periodic resource usage reports and events are also collected using the messaging system. Messaging system failures break inter-component communication which prevents the cloud from serving requests like VM creation, and thus, is considered higly severe.

3.2 Failures of OpenStack Services

Keystone: Keystone provides authentication and authorisation services for the remaining OpenStack components. If this service fails, requests can no longer be verified and are rejected by the services. The actual behaviour depends on the used authentication method of either unique IDs or a public key infrastructure. In the latter case, request validation is theoretically still possible without Keystone, but is prone to identity theft, as the revoked certificates are no longer updated. As a result Keystone failures are classified as highly severe.
Nova: Nova consists of several services. The API service handles internal and external requests and a failure will break all communication with Nova. The scheduler distributes VM requests among all compute hosts. Without it, new requests will remain in a pending state. So, scheduler failures will not render the whole cloud useless, but at the same time are not limited to a single host, so they are considered medium severe. The conductor service mediates all database accesses by compute hosts, mainly updating their current resource usage. A failure may lead to the scheduler making decisions based on outdated metadata. A failed compute service is of low importance, as just the VMs running on this host are no longer manageable but continue to run. Scheduling requests will ignore the affected host and no new machines are placed on it. The failure of the underlying hypervisor of a host (e.g. KVM) leads to the loss of all ephemeral data of the VMs. Resultant from our assumptions about the classification, we consider it being of low severity. The remaining services, Cert, Consoleauth and noVNC, are not always required or only provide additional functionality, which is not critical for general infrastructure operation.
Glance: Glance consists of an API service and a registry for storing metadata about VM images. A failure of either one will render the component useless, as incoming requests can not be served without the image's metadata (owner, visibility etc.). Because this is not limited to a single host, it is classified as medium severe.
Cinder: The Cinder block storage service of OpenStack is mostly out of scope in this work, because it can be implemented using renowned highly available block storage systems as backend providers. In this paper, we only need to consider the OpenStack-related parts of this service. The API service and the scheduler equals other component's API services and schedulers when considering HA. The severity of the volume and backup services is considered medium, since failures do not affect the cloud as a whole but also do not only affect single VMs on one host but all VMs using block storage.
Swift: Since Swift's architecture has been designed for HA and fault tolerance. Thus, the severity of failing storage nodes only depends on the Swifth configuration parameters like the deployed cluster size and replication count. In this paper we

only need to consider OpenStack-related parts. One of them, the Swift proxy service, can be handled like all stateless services.

Quantum: The Quantum component, that provides the network functionality, is divided into several parts: The Quantum server is responsible for management and includes the public API and scheduling facilities, that behave like the Nova API and scheduler. The DHCP-Agent handles automatic configuration of new VMs and manages the underlying Dnsmasq processes. A failure will break the network and routing configuration of VMs. The core parts of Quantum are the L3-Agent and the Plugin-Agents. Both are an abstraction for the actually used vendor-specific software switch, for example OpenvSwitch. In the evaluated setup, the L3-Agent handles the routing and bridging configurations for all deployed virtual networks. Traffic from tenant VMs is routed through a host running this service and then bridged to an external network. The Plugin-Agents are the respective counterparts, running on the compute nodes. They manage the connection of VMs to their networks and the tunnelling of traffic to the nodes running the L3-Agent. While Plugin-Agent failures only affect single hosts, L3-Agent failures may affect multiple virtual networks across many hosts. The networks continue to operate with the last known configuration until the L3-Agent is restored, but may experience inconsistencies due to the missing configuration updates. A failure of the L3-Agent is therefore medium severe, whereas the Plugin-Agents are of lower importance.

Horizon: Horizon is a web-based frontend for OpenStack and uses the API services of all other components in order to provide its services. A failure will not affect the regular operation of the infrastructure and therefore is classified low severe.

4 Best Practice for Providing Highly Available IaaS

Since the reliability of IaaS clouds is important, there have already been efforts dealing with HA. In the following, we summarise the current best practices.

The current most widely used approach for providing highly available OpenStack clouds is the Pacemaker cluster stack [4]. It consists of a collection of software tools to provide replication and failover capabilities for generic services. Corosync is used as messaging and membership management layer in order to connect all participating nodes of a cluster [9]. The communication between all nodes follows the virtual synchrony pattern and is realised using the Totem single-ring ordering and membership protocol [6]. Corosync can use multiple physical connections in a cluster to be resilient against link failures or routing errors, replicating all messages either in an active or passive fashion.

The Pacemaker cluster manager is responsible for monitoring and managing the cluster. All services are configured using *resource agents*, small scripts, which map the service specific status variables and init calls to standard Pacemaker calls. The internal policy engine can then query the services for their state and enforce failovers in the event of a failure. The affected resources will then be relocated to another host in the cluster. To guarantee a consistent cluster state,

Table 1: The table shows the criticality classification of the individual components, their usage of the database, the message queue, and whether they are stateful.

Component	**low**	**med**	**high**	**database**	**message-queue**	**stateful**
Database			×	×		×
RabbitMQ			×		×	×
Keystone			×	×		
API Services		×				
noVNC	×					
Horizon	×			×		×
Hypervisor	×					×
Nova-Scheduler		×			×	
Nova-Cert	×				×	
Nova-Consoleauth	×				×	
Nova-Conductor		×		×	×	
Nova-Compute	×			×	×	×
Glance-Registry		×			×	
Cinder-Volume		×				
Cinder-Scheduler		×			×	
Cinder-Backup		×				
Swift-Proxy		×				
Swift-Storage	×					×
Quantum-Server		×		×	×	×
DHCP-Agent		×		×	×	
Metadata-Agent		×		×	×	
L3-Agent		×		×	×	×
Plugin-Agent	×			×	×	×

Pacemaker replicates configuration changes among all participating hosts. Failed hosts can be excluded by using fencing mechanisms to prevent a possible faulty behaviour. Fencing of hosts can be achieved following the shoot the other node in the head (STONITH) approach, where a failed host is separated from the remaining cluster through measures like remote shutdown or even stopping the power supply via manageable power distribution units.

The services of OpenStack are replicated following the primary-backup approach as described by Buhiraja et al. [7] and listen on a virtual IP-address also managed by Pacemaker. Some of the infrastructure services, like MySQL or the RabbitMQ messaging system, additionally require highly available block storage. For these applications, DRBD is used. DRBD stands for *distributed replicated block-device* and replicates write requests for a volume to another standby host in either synchronous or asynchronous fashion.

Pacemaker-based HA solutions can be more complex to implement and maintain compared to a lightweight load-balanced setup, that we use. Following the primary-backup approach, standby resources are only used in the event of a failure and reduce the overall efficiency of the cluster. The features of service and resource replication can often be directly achieved using the scheduling mechanisms in newer versions of OpenStack. The advantages of Pacemaker include its ability to be used for almost any generic service and its integrated support for most cluster related operations, such as monitoring, policy enforcement, and fencing.

Another cluster management software is the *Red Hat Cluster Suite* [17], a set of tools for monitoring and managing distributed resources in a group of hosts. The features and drawbacks are similar to the Pacemaker stack, as it is able to provide HA for generic services.

5 Scalable High Availability for IaaS

This section shows how our approach improves on the existing ones, tackling their shortcomings. At the same time we make use of simple and reliable mechanisms and allow to handle failures more fine-grained. Compared to existing approaches, we do not require any manual intervention on failures.

Our basic intention is to transparently mask service interruptions from OpenStack users, such that the services are always available to them, in order to encourage user's trust in cloud services. We can achieve scalability and availability at the same time involving replication and load balancing techniques. While many components are stateless and easy to replicate, we also have to consider stateful services like the central database and the message bus. Most sophisticated, however, is the Quantum component, consisting of many services and components.

We define an architecture with several roles of nodes. There are nodes belonging to the *persistence cluster* hosting the central database, the message bus and the HA tools Keepalived and HAProxy [4]. Then we have *cluster controller* nodes that host all the important infrastructure services like Keystone, Glance, Nova, Quantum and Horizon for example. Naturally, we additionally have *compute nodes* that run the actual VMs.

We tolerate single host failures and continue provisioning of all required cloud services even when failures occur. With our approach even multiple failing hosts at the same time can be tolerated, as long as they are part of different roles. Additionally, if they are of the same role, the remaining correct replicas only have to be able to handle the increased load to mask the failure.

In our approach we make use of virtual IPs, which allows us to implement automatic failover to other active service endpoint replicas. Virtual IPs allow, based on gratuitous ARP, to use one IP address as a service endpoint independently from the actual machine hosting the service. Hence, we can replicate the service on several machines and dynamically redirect traffic if one of them fails. Since we know which replicas are up and online, we can even apply load-balancing across

the individual replicas in order to distribute the workload evenly. This works well especially for stateless services, for example the API services. Even though it is easily adaptable to any TCP-based services we focus on OpenStack-specific details here. The actual implementation of the above paradigms is done using Keepalived and HAProxy [4].

For stateless services, we can use active replicas and load balancing of requests instead of passive standby replicas. By this a better resource usage and easier failover can be achieved because requests and load are spread across all replicas. On failure we do not need to migrate service state or warm up a standby replica, but the load of the failed replica is directly distributed and adds evenly to the load of all other replicas.

Basically, we always try to minimise the failover times. That is, the time of service unavailability that is noticeable by the clients. Hence, we try to achieve automatic failover that does not require any manual intervention.

Making OpenStack services highly available using Keepalived and HAProxy is a very generic approach. It can be easily adapted to all kinds of services even beyond OpenStack. Basically, applying the approach only requires the service, which has to be highly available, to be stateless. Then it can be deployed on several hosts and requests are balanced evenly across all correct replicas. Stateful services always require a little more effort. However, sometimes there can be worked around the statefulness of a service, as it is described in this work for the DHCP-agents in Section 5.3. State is offloaded by the service to the database, so the DHCP-agent itself is actually stateless, but the database maintains its state. For the database in turn, there has to be a special approach for HA, which in our case is achieved using the Galera framework as described in Section 5.2.

5.1 High Availability for Stateless Services

For stateless services the basic idea involves HAProxy running on several hosts knowing all actual service endpoints. Clients can access any HAProxy instance they want and will be redirected transparently to the actual service. Here, HAProxy tries to involve stickyness of clients to services in order to support caching mechanisms. Since the HAProxy itself might fail as well, Keepalived is used, which applies gratuitous ARP in order to capture incoming packages from clients to a specific virtual IP [13]. This allows to transparently mask failing hosts (HAProxy instances) and to automatically redirect all traffic to another HAProxy instance. From the client perspective, this results in a virtual IP that is *always available*. Clearly, this approach requires the actual service (endpoints) to be replicated as well. This architecture is illustrated in Figure 3 with the dashed line showing one possible request-path. HAProxy is chosen over competitors like Nginx or Pen due to its resource efficiency and stability [18].

5.2 High Availability for Stateful Services

Apart from stateless services, we also have to consider stateful services and building blocks of OpenStack like the MySQL database and the Quantum networking

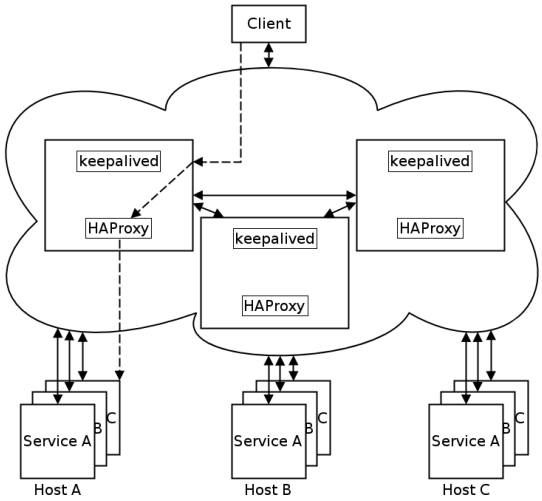


Fig. 3: Load-balancing Cluster with Keepalived and HAProxy

services. The database can be made highly available using MySQL Galera [4], while the Quantum networking services require further considerations covered in Section 5.3.

Another fact that could bring further conflicts is inherently caused by the asynchronous network itself. It's not possible to decide whether services are responding slow or failed. Hence, hosts could misleadingly be considered faulty and later show up correctly again. However, in the HAProxy and Keepalived approach for stateless services this is no problem, because faulty hosts will be excluded from the load balancer until they respond back positively to the health checks. As a result, on failures, future requests by clients will be answered correctly (by another replica). However, the request sent to the service in the very moment of its failure will be lost and could be masked by the client library's timeout mechanisms using request repetition.

5.3 High Availability for Quantum Networking

When analysing Quantum, we discovered special consistency problems when agents fail but the underlying mechanism continues to work properly. This affects the L3-Agent, because OpenvSwitch is still working without the agent, but OpenStack assumes it has failed and reschedules the according networks. For this problem, we show our approach of cleaning up the old virtual infrastructure that has been managed by the failed L3-Agent, such that the newly commissioned replacement will not conflict with any legacy virtual networks. In addition, here we describe our monitoring approach, that is necessary to notice agent failures and trigger the cleanup accordingly.

Cleanup Legacy Virtual Networks A trivial approach for the infrastructure cleanup would be a simple script, checking the affected components in short intervals triggering the needed action when a failure is detected. This has the advantage of low intervals between the checks, but recovery is problematic when the script itself terminates unexpectedly. The cleanup could also be implemented using a reoccurring tasks framework like Cronjob, that is considered well-tested and reliable. As with a simple script, the failure handling is not satisfying and Cronjob's minimum interval of one minute is too high in order to provide efficient failover. The current implementation uses Init systems (upstart or systemd) to manage the lifecycle of the cleanup utility. Like cron, the Init systems are well tested components and assumed to be reliable. They handle the activation and deactivation of the job, as well as failure handling in the event of an unexpected termination. The interval between checks can be chosen as low as in the simple scripted approach.

Rescheduling of Virtual Networks to Another L3-Agent Network nodes have been limited to Master/Slave operation in the past [4]. Starting with OpenStack Grizzly, virtual resources can be scheduled. This allows to operate multiple network nodes as a basic requirement for making the services highly available. However, virtual routers still require manual intervention to be relocated on another network node. We use OpenStack's integrated scheduling mechanism in order to automatically move routers to other hosts. The health of all L3-Agents is periodically monitored and router rescheduling is triggered, when a faulty agent has been detected. Then, the corresponding routers are migrated to other hosts. In our approach this is based on the active scheduling policy, while others involve custom schedulers[6].

The health checks are performed by the periodic task framework of OpenStack, executed in configurable intervals[7]. By reducing the periodic interval, lower failover times can be achieved, but at the cost of additional resource usage. The lower bound is limited by Python's cooperative threading model, where long running tasks block all others inside the same process and might even delay the execution of the next periodic interval. New tasks are only started, after the previous one finished or yielded its execution to the next task.

In addition to highly available routers, this solution also enforces a consistent network configuration when only the L3-Agent fails but the underlying software switch (e.g. OpenvSwitch) continues to run but is no longer manageable by OpenStack. Our policy will remove all virtual routers and their configuration from the affected host, even without a connection to the Quantum server.

The reliability of this solution only depends on the reliability of the used Init system and its capabilities to handle failures. In our evaluation environment, failure handling was successfully tested with Upstart and systemd. The monitoring job will be loaded at boot time and restarted automatically in the event of an unexpected termination.

[6] `http://lists.openstack.org/pipermail/openstack/2013-December/004226.html`, last accessed Feb 3th, 2014

[7] defaults to 40s

If a slow host is considered faulty and comes back online later, with our approach no conflicts arise. This is caused by our cleanup procedure, that prevents the same network to be maintained by more than one agent at the same time. The cleanup script itself will be maintained by the Init system and automatically restarted on failure, so the cleanup is still executed reliably.

Compared to the existing state, with our solution manual intervention is not necessary. In addition, our approach is very fine-grained, because we not only consider complete host failures but even handle failures of the individual processes.

High Availability for the DHCP-Service Providing highly available DHCP services is not a problem in OpenStack Grizzly as multiple DHCP-Agents can be configured to serve a network. Since all agents synchronise their managed instances and configurations indirectly through the relational database, the underlying Dnsmasq configuration needs not to be changed.

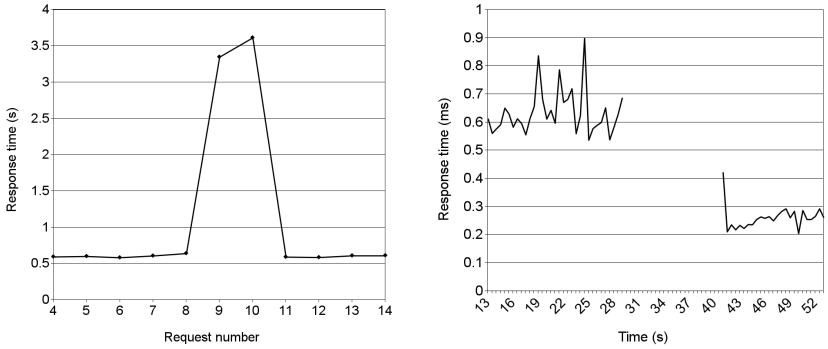
6 Evaluation

In order to evaluate our approach and proof its functionality, we deployed it in a lab environment. Automatic failover allows the system to tolerate and mask any complete host and individual service failure in our testbed.

This evaluation focusses on failover times and special attention is paid to Quantum L3-Agent failures while other stateless services are grouped together and Keystone is evaluated as one arbitrary representative. The approach for this evaluation is to simulate failures by cutting network connections (complete host fail) and killing single processes. Then, we measure the time until clients experience the failed service available again served by another host.

For the Keystone evaluation we issued the `user-list` request repeatedly to the Keystone service. We expect all other stateless services to behave the same, different from Keystone only by a constant factor representing the difference in computing effort. Failures will be detected by the healthchecker from Keepalived which maintains an active replica list. Faulty replicas get removed from this list, and correct ones can be added back again later on. On failover we do not have any bootstrapping times, because the services are already up and running. Usually clients get sticked to one replica, to allow optimal use of caching mechanisms. Figure 4a shows that on failures the response time increases. Caused by the failure, requests time out and are retried by the client library. After the faulty replica is removed from the list, requests get responed by another replica.

The evaluation of L3-Agent failures is done in our testbed environment hosting 54 VMs based on CirrOS basically running idle. We evaluate failures by sending ICMP echo requests (pings) to a VM every 100ms while killing the L3-Agent process. Connectivity is measured by ICMP echo responses (pongs) from one arbitrarily chosen VM. Our cleanup mechanism is triggered on failure which disables all virtual infrastructure components maintained by the failed agent. Another agent notices the failure and cleans up the metadata and reschedules the failed router. The agent that has been selected by the scheduling algorithm



(a) Stateless Services Failover Time (b) L3-Agent Failover Time

Fig. 4: Failover Times for Stateless and Stateful Services

receives a notification via the message bus and recreates the virtual infrastructure. Figure 4b shows a gap of pongs where the agent failed and a recovery of VM connectivity about 11 seconds later. This time period consists of the above parts of our automatic recovery procedure. The measurement shows that the automatic failover procedure is working without manual intervention and VMs are back online after a short period.

7 Conclusion

Since adoption of cloud computing is steadily increasing, the need for reliability is getting more relevant and important for both, providers and customers of IaaS clouds. In this work, we start with a general architecture discussion of IaaS clouds, then take OpenStack as an example and pinpoint shortcomings in existing approaches for HA. We provide a detailed failure analysis and classification describing effects of failing hosts and services to the cloud. In a next step, we solve the identified weaknesses of existing HA approaches. With our approach we want to minimise the time of unavailability of cloud infrastructure services as experienced by the cloud users and provide automatic failover procedures that do not require manual intervention by an cloud administrator.

References

1. vSphere - Replication. `https://www.vmware.com/de/products/vsphere/features-replication`, last accessed Mar 25th, 2014.
2. Critical workloads, private deployments and market opportunities abound in the cloud, 2013. `http://insights.ubuntu.com/resources/article/critical-workloads-private-deployments-and-market-opportunities-abound-in-the-cloud/`, accessed Mar 23th, 2014.

3. OpenStack Grizzly Architecture (revisited), 2013. http://www.solinea.com/blog/openstack-grizzly-architecture-revisited, last accessed Mar 24th, 2014.
4. OpenStack High Availability Guide, 2014. http://docs.openstack.org/high-availability-guide/content/index.html, last accessed Feb 3rd, 2014.
5. OpenStack High Availability Using Active/Passive, 2014. http://docs.openstack.org/high-availability-guide/content/ha-using-active-passive.html, last accessed Mar 23th, 2014.
6. Y. Amir, L. E. Moser, P. M. Melliar Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *Transactions on Computer Systems*, pages 311–342, 1995.
7. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary-Backup Approach. pages 199–216, 1993.
8. B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.
9. S. C. Dake, C. Caulfield, and A. Beekhof. The Corosync Cluster Engine. In *Linux Symposium*, pages 85–100, 2008.
10. M. Greer. Survivability and Information Assurance in the Cloud. In *Proc. of the 4th Workshop on Recent Advances in Intrusion-Tolerant Systems*, 2010.
11. M. Krigsman. Pain and Pleasure in the Cloud, 2011. http://www.zdnet.com/blog/projectfailures/intuit-pain-and-pleasure-in-the-cloud/14880, last accessed Feb 5th, 2014.
12. R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. *Computer*, pages 65–72, 2012.
13. S. Nadas. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798, 2010. https://tools.ietf.org/html/rfc5798.
14. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Symposium on Cluster, Cloud, and Grid Computing*, pages 124–131, 2009.
15. J. Pepitone. Amazon EC2 outage downs Reddit, Quora, 2011. http://money.cnn.com/2011/04/21/technology/amazon_server_outage/, last accessed Feb 5th, 2014.
16. P. Sempolinski and D. Thain. A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In *Cloud Computing Technology and Science*, pages 417–426, 2010.
17. K. Shiraz. Red Hat Enterprise Linux Cluster Suite, 2007. http://www.linuxjournal.com/article/9759, last accessed Mar 25th, 2014.
18. W. Tarreau. New benchmark of HAProxy at 10 Gbps using Myricom's 10GbE NICs (Myri-10G PCI-Express), April 2009. http://haproxy.1wt.eu/10g.html.
19. T. Thanakornworakij, R. Sharma, B. Scroggs, C. B. Leangsuksun, Z. D. Greenwood, P. Riteau, and C. Morin. High Availability on Cloud with HA-OSCAR. In *Proc. of the 2011 International Conference on Parallel Processing - Volume 2*, pages 292–301, 2012.