SPONSORED BY THE

Federal Ministry
of Education
and Research

ASMONIA

Attack analysis and **S**ecurity concepts
for **MO**bile **N**etwork infrastructures,
supported by collaborative **I**nformation exch**A**nge

Design and Implementation of an Intercloud demonstrator

D3.3

Contributors:

Cassidian Systems

ERNW Enno Rey Netzwerke GmbH

Fraunhofer Research Institution for Applied and Integrated Security (AISEC)

Hochschule Augsburg

Nokia Siemens Networks Management International GmbH

RWTH Aachen

Editor: Mark Gall (Fraunhofer AISEC)

Author(s)	**Company**	**E-mail**
Dr. Niels Fallenbeck	Fraunhofer AISEC	niels.fallenbeck@aisec.fraunhofer.de
Mark Gall	Fraunhofer AISEC	mark.gall@aisec.fraunhofer.de
Bernd Jäger	Nokia Siemens Networks GmbH & Co KG	bernd.jaeger@nsn.com
Joachim Lüken	Nokia Siemens Networks GmbH & Co KG	joachim.lueken@nsn.com

About the ASMONIA project

Given their inherent complexity, protecting telecommunication networks from attacks requires the implementation of a multitude of technical and organizational controls. Furthermore, to be fully effective these measures call for the collaboration between different administrative domains such as network operators, manufacturers, service providers, government authorities, and users of the services.

ASMONIA is the acronym for the German name* of a research project that aims to improve the resilience, reliability and security of current and future mobile telecommunication networks. For this purpose the ASMONIA consortium made up of several partners from academia and industry performs a number of research tasks, based on the specific expertise of the individual partners. The project running from September 2011 till May 2013 receives funding from the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF). Various associated partners further contribute on a voluntary basis.

* The full name is "**A**ngriffsanalyse und **S**chutzkonzepte für **MO**bilfunkbasierte **N**etzinfrastrukturen unterstützt durch kooperativen **I**nformations**A**ustausch" (Attack analysis and security concepts for mobile network infrastructures, supported by collaborative information exchange).

Partners:	Cassidian Systems
	ERNW Enno Rey Netzwerke GmbH
	Fraunhofer Research Institution for Applied and Integrated Security (AISEC)
	Hochschule Augsburg
	Nokia Siemens Networks Management International GmbH
	RWTH Aachen
Associated Partners:	Federal Agency for Digital Radio of Security Authorities and Organizations (BDBOS)
	Federal Office for Information Security (BSI)
	Deutsche Telecom AG (DTAG)

For more details about the project please visit www.asmonia.de.

Executive Summary

This document provides an overview of the demonstrator of the ASMONIA Collaborative Architecture that has been developed during the ASMONIA research project. The demonstrator focuses on the Intercloud Cloning of SIP servers in an overload state. One advantage of cloning a virtual machine (VM) hosting a SIP server to another cloud is that in case of an overload situation resulting from an attack of the origin cloud the SIP service can be handled by machines in other clouds even if the origin cloud becomes unreachable. However,

A major challenge of Intercloud cloning across different administrative domains is that a running VM needs to be cloned in the origin cloud and transferred to a different independent cloud system, the target cloud. To enable the cloned VM to run in the target cloud, it not only must be imported into the cloud management system of the target cloud but also needs to be reconfigured on-demand to ensure that the SIP-server will run in the new environment of the target cloud. Additionally, a load balancer needs to be reconfigured on-demand to forward incoming calls to the additional VM.

In this document the design and implementation of the demonstrator is presented as well as a short conclusion about the results of the implementation and testing of the demonstrator. With the proposed solution we were able to live-clone a VM hosting a SIP server and bring up a new VM based on the cloned machine in a remote cloud within 5 minutes. This demonstrator does not include optimizations which would further speed up the live-cloning mechanism but is intended as a proof-of-concept showing that live-cloning can be achieved in complex environments to deal with peak loads caused by attacks and to ensure that services stay available even in situations where particular clouds become unresponsive.

Table of Contents

1 Introduction

In our previous work [1] we presented the ASMONIA Collaborative Cloud Architecture. This architecture based on the concept of Intercloud shows how cloud computing can be integrated into the backend of mobile network providers in order to strengthen the availability of backend services. Several issues in that architecture are in the focus of active research, e.g. service discovery and registry, policy management, live migration, match-making and horizontal federation of clouds — to name only a few. In order to evaluate the design of the architecture and show its feasibility we implemented a demonstrator. But the task of implementing the complete architecture is out of scope of this research project. Thus we need to limit the scope of this demonstrator to focus on a certain number of issues of the architecture, while ignoring others.

An important and central issue in the research area concerning Intercloud is the horizontal federation of clouds, which denotes a resource management across cloud boundaries and different administrative boundaries. The Intercloud Cloning service which has been described in the previous deliverable D3.2 [1] is a good representative for horizontal federation. The demonstrator described in this document focuses on Intercloud Cloning, Auto Scaling, VM Monitoring and Load Balancing.

Intercloud Cloning describes the process of VM cloning between different cloud infrastructures including the process of cloning the VM in the origin cloud, transferring the snapshot of the VM to the target cloud, importing the snapshot into the target cloud system and performing some reconfiguration tasks to make the VM running in the new environment.

The Auto Scaling mechanism ensures that VMs will be automatically scaled up and down depending on the current workload of these VMs. The workload is determined by a monitoring system:

VM monitoring is a flexible monitoring approach allowing to monitor different VMs in a dynamic environment. Because of the fact that different VMs can host different services they need to be monitored in a different manner. It is important to detect overload situations that the monitoring system can deal with different services and monitor the services differently. It is also important that the monitoring system can deal with cloned VMs even if they moved into another cloud infrastructure.

Finally, Load Balancing is needed to distribute incoming service requests like SIP calls to the VMs that handle these events. The load balancing service needs to be reconfigured on-demand if VMs scale up or down.

As these services are part of the ASMONIA Collaborative Cloud Architecture, it is important to note that they will not run on top of a single cloud, but are parts for the solution of the horizontal federation issue between multiple clouds. The demonstrator uses two clouds for the federation.

Design

1.1 Overview

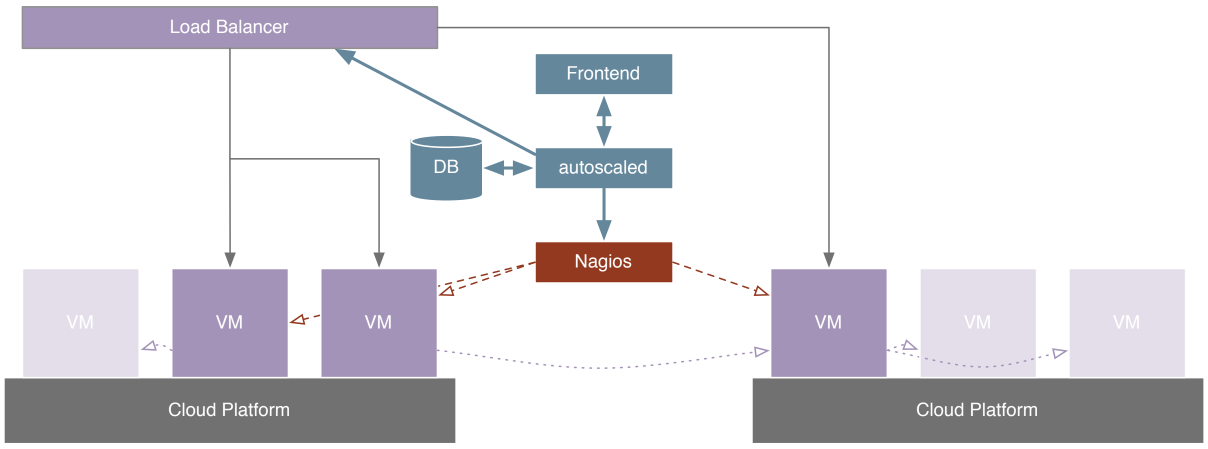


Figure 1: Architecture of the ASMONIA Intercloud Cloning components

Figure 1 shows the architecture of the components used for Intercloud Cloning of the virtual machines. Two clouds are shown running few virtual machines which are monitored by **Nagios** representing the VM Monitoring service defined in the ASMONIA Intercloud Architecture. Nagios is an open source monitoring tool, which can be used to monitor servers and virtual machines and is connected to **autoscaled**, representing the the Auto Scaling service from the ASMONIA Collaborative Architecture. The virtual machines may contain any service, but for the demonstrator we chose a VoIP-Server as the service in the form of an OpenSIPS server. OpenSIPS is an open source voice-over-IP-telephony server. Depending on the virtual machine’s states, autoscaled will decide whether to start the cloning process of a virtual machine or not.

The state of a VM may contain the load of the virtual CPU, the used network bandwidth or the number of users currently logged into a service hosted by the VM. This decision is based on thresholds that can be configured via a **Frontend** by the administrator of the cloud system. When an overload situation is detected, autoscaled decides whether the cloning process will be performed in the cloud in which the overloaded VM resides or if a new instance of this VM will be created in a remote cloud, which is referred to as Intercloud Cloning. For decision-making, several strategies can be implemented ranging from a simple algorithm monitoring the systems’ load to more sophisticated ones taking into account several different metrics in order to decide when and where a new VM will be brought up.

Our demonstrator makes the scheduling decision based on information provided by the VM Monitoring service, i.e. a Nagios monitoring system. Nagios employs a self-developed script to monitor the OpenSIPS voice-over-IP-telephony server and to get accurate information about the server’s state. Using the autoscaled frontend an administrator can specify a time interval after which the cloning process will be triggered, when the VM is in a warning or critical state. The clone process of a VM does not only contain the copy process of the VM’s image but needs also to ensure that the image will be successfully imported into the target cloud. Additionally, services and software installed in the VM need to be reconfigured to be able to run in the new environment. For example, when cloning an OpenSIPS-Server, the configuration needs to be adjusted to the IP addresses assigned to the particular VM in order to successfully start the OpenSIPS service once the VM has been booted. After successfully cloning the VM, the load balancer needs to be reconfigured to ensure that new service requests will be redirected also to the newly created VM. Therefore, autoscaled will connect to the load balancer directly, update its configuration files accordingly and will restart the load balancing service.

1.2 Intercloud Cloning

Intercloud Cloning was described in the previous deliverable [1] as “Live Cloning”. The name was changed in order to direct the focus more on the horizontal federation part of the problem. The Intercloud Cloning does not use prepared images, but creates a snapshot as part of the cloning process. In order to prevent the management of images we create a snapshot of the particular image during the cloning process. Because of the fact that running VMs are cloned on-demand we can ensure that the newly created VM contains security updates and patches applied to the VM that it has been cloned from. An additional advantage is that network bandwidth will only be consumed if a VM needs to be cloned. Our approach prevents resource consumption if a VM image has been updated and needs to be redistributed to all participating clouds even if the images will never be used in the future.

The steps involved in the Intercloud cloning process are:

1. Taking a snapshot of a currently active virtual machine in the source cloud
2. Transferring the snapshot from the image service of the source cloud temporarily to the Intercloud Middleware
3. Transferring the snapshot from the Intercloud Middleware to the destination cloud and registering it at its image service
4. Starting a new virtual machine in the destination cloud using the snapshot as a base image

1.3 VM Monitoring

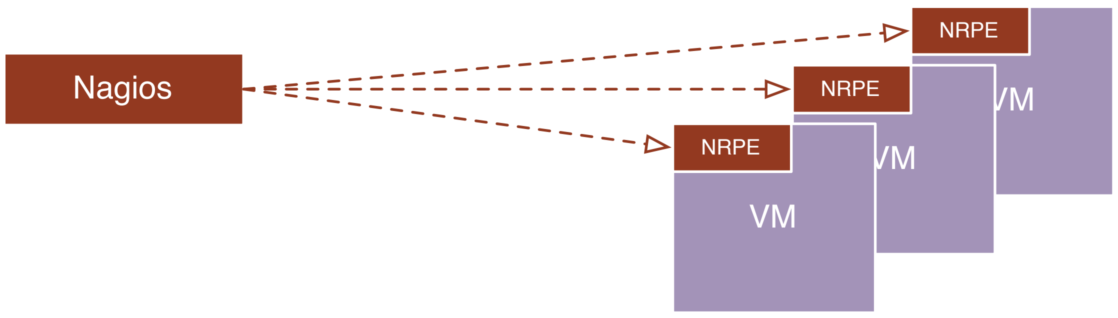


Figure 2: Architecture of the Nagios monitoring system

The monitoring of the virtual machines is done using Nagios. Figure 2 shows the architecture of Nagios how it is used in the demonstrator. The Nagios server depicted on the left hand side of the figure connects to the Nagios Remote Plugin Executor (NRPE) installed on the monitored machines via a secure network connection. The NRPE calls scripts to gather and extract runtime information from the machines. Depending on thresholds configured on the Nagios server the scripts executed by the NRPE will return one of four possible return values:

- OK (Return code 0) if everything is okay and thresholds are not exceeded
- WARNING (Return code 1) if warning threshold has been exceeded
- CRITICAL (Return code 2) if critical threshold has been exceeded or host is not reachable

- UNKNOWN (Return code 3) if an error occurred during the script execution and therefore the system's state is unknown

Each script monitors only one particular service on a physical host or virtual machine and is run by a particular NRPE service for its respective physical host or virtual machine.

To monitor the OpenSIPS voice-over-IP server a Nagios script was created which can be downloaded from github [3]. It uses the OpenSIPS control program opensipsctl, which is provided by OpenSIPS itself. This program can be used to display runtime information about the OpenSIPS instance running on the machine like registered and active users or currently active connections. The administrator configures two thresholds on the Nagios server to specify when a particular machine running OpenSIPS is in a warning or a critical state respectively.

1.3.1 Detecting Overload Situations

For each virtual machine that is monitored by Nagios, a number of metrics is registered that can display the status values OK, WARNING and CRITICAL. What information influences a metric depends on the metric. For Nagios it is important that each metric defines threshold numbers that allow detecting overload situations. Thus Nagios signals that a virtual machine is in an overload situation by displaying the metric status WARNING or CRITICAL.

1.4 Load Balancing

Load balancing, as also shown in Figure 1, automatically distributes incoming application traffic across multiple instances of an application. By scaling the resources in response to incoming application traffic a higher resource availability can be achieved. When "Load Balancing" is used in conjunction with "Intercloud Cloning" it is necessary that the load be distributed between different clouds, i.e. the source cloud and the destination cloud.

1.4.1 Automatic configuration of load balancers

Besides being an important service on its own, "Load Balancing" is a basis on which the "Auto Scaling" Service builds upon. In this case the load balancer in use needs to be automatically configured by the "Auto Scaling" process in order to include the virtual machines that have been newly started when scaling up or excluding those that have been stopped when scaling down.

1.4.2 Specifics for SIP-Servers

In contrast to stateless HTTP load balancing which is easy to achieve by just redirecting incoming requests to a randomly chosen available webserver it gets more complicated when dealing with stateful connections necessary in voice-over-IP scenarios. The reason is that established connections cannot be redirected to another server without interrupting the connection and dropping the voice call. Therefore the load balancer itself needs to track the state of active connections.

1.5 Auto Scaling

The Auto Scaling service joins together "Intercloud Cloning", "VM Monitoring" and "Load Balancing" in order to create a service that is able to deal with overload situations automatically without the need of user interaction.

1.5.1 Configuring the automatic scaling process

A user can configure the scaling process in the following ways:

- For each virtual machine of a user, he can decide to put it under the control of the scaling process or remove a virtual machine from its control by declaring the virtual machine a *virtual machine type*
- For each *virtual machine type* a user can define the number of virtual machines that the scaling process may start (and stop) by defining an upper bound and a lower bound
- For each *virtual machine type* a user can define threshold values for each metric that is registered for the *virtual machine type*. When these thresholds are exceeded the scaling process will activate.

1.5.2 Scaling Up in the Intercloud

The general steps involved when a virtual machine is scaled up differ from the scale down process. These steps are:

1. *Detect load exceeding the configured threshold*

   To detect overload situations the monitoring system, which observes the hosts running the services, is frequently queried and the result is parsed to extract appropriate information for the particular host(s).

2. *Decision Making*

   This step is separated into 2 different tasks. The first task deals with the decision making whether a scaling is needed. Depending on this decision, the target cloud needs to be identified which will host the newly created VM.

   a. Scaling Decision

      To avoid unnecessary network copy operations and a heavy load resulting from permanently booting and destroying instances the autoscale mechanism needs a sophisticated decision making algorithm, which avoids scaling up when detecting short-term peak loads (hysteresis). On the other hand, it must be ensured that service requests can be fulfilled within an appropriate time frame. A straight-forward approach would be to scale up if the threshold is exceeded for a specified time period.

      There are many possibilities to implement a sophisticated approach to optimize the decision making process based on sophisticated data analysis approaches and machine learning processes.

   b. Target Cloud

      In the upscaling scenario, the scheduling decision must include the choice of the cloud which will be used to host the new VM instance. This decision affects the time needed to provide an additional instance because the disk image or snapshot of a particular VM needs to be transferred to the target cloud before the new instance can be launched.

      i. Request additional Resource in the ASMONIA Intercloud

         The autoscaling process contacts the ASMONIA Intercloud middleware and requests additional resources.

      ii. Activation of Intercloud Autoscaling (in combination with Intercloud Monitoring)

         To fulfill the request, the middleware activates the Intercloud autoscaling process knowing the current states of the connected cloud systems from the Intercloud monitoring service provided by the middleware.

iii. Activation of Intercloud Matchmaking

If additional resources need to be requested from a connected Cloud, a matchmaking process is started to find the best suitable Cloud. Several services provided by the Intercloud middleware will be used, e.g., Intercloud Service Publication, Intercloud Service Discovery, and Intercloud Service Subscription.

The choice of which target cloud can be used for scaling up can be controlled by policies. Using policies an administrator can prevent to clone particular services into remote clouds or limit the Intercloud cloning to target clouds which fulfill special requirements.

Additionally, Intercloud Matchmaking also deals with authentication of the connected Cloud systems and ensures that only resources will be acquired that fulfill policies provided with the resource request.

3. *Move VM image to target cloud*

   This step is only needed if a new instance of a given VM will be created in a different cloud, which is referred to as intercloud live-cloning. This will be done be creating a snapshot of the VM which is then transferred to the target cloud.

   A possibility is needed to transfer VM images or snapshots to the remote cloud. OpenStack provides a REST API which promises to provide functionality to import and export VM images and snaphots. This step includes transfer of the image or snapshot, creation of a new instance/snapshot object in the Cloud management system as well as to assign this object to the matching user account. Moreover, configuration settings need to be deployed to the newly created object including credentials, access rules and others. The unique identifier of this imported image needs to be available to the autoscale component.

4. *Create new instance (from a provided snapshot)*

   The new instance of the VM needs to be booted. Technically, this is accomplished by calling an Openstack API function. Once the instance is available, its network address an/or hostname must be made available to the autoscale component. Depending on the Cloud setting, this instance needs to be made available in the intercloud and a floating IP address needs to be assigned to the instance.

5. *Configure the newly created instance*

   After creating the instance of the VM, the OpenSIPS service needs to be reconfigured depending on the VM's current network configuration, before it can be started. It is crucial that the VM's IP address is correctly configured in the OpenSIPS configuration file to ensure that the service will start correctly. Therefore the autoscaled needs to connect to the newly created VM, rewrite the OpenSIPS configuration file with the correct IP address(es) assigned to the newly created virtual machine and (re)start the OpenSIPS service.

6. *Update the load balancer*

   Finally, the load balancer's configuration needs to be updated to incorporate the newly available VM instance. The autoscale components needs to connect to the load balancer and to update its configuration file appropriately according to the information gathered from OpenStack (IP address of the new instance, etc.).

1.5.3 Scaling Down in the Intercloud

When scaling down, the system has to pass fewer stages compared to the scaling up process.

1. *Detect underload situation*

   Detecting an underload situation is comparable of the detection of an overload situation. The administrator needs to configure thresholds in the Nagios system, which are used by the aforementioned Nagios script [3] executed on the OpenSIPS VMs to detect underload situations. An underload situation is characterized by exceeding a predefined threshold i.e. showing that the VM is idle. In the SIP scenario this might be the case if no user is registered at the OpenSIPS server hosted by the particular VM.

   It is noteworthy that underload situations are more difficult to detect than overload situations: It is hard to say if a VM can be shut down if the CPU is idle because there might be other processes e.g. waiting for user input. However, in a SIP scenario the number of active calls handled by the SIP server is adequate metric to detect underload situations.

2. *Decision Making*

   Similar to the upscaling the decision must be made, when to scale down a virtual machine. An administrator can define a minimum number of VMs, which should always be available. When an underload situation is detected autoscaled will not further scale down if this minimum number of VMs has been reached

3. *Reconfigure the load balancer(s)*

   The load balancer(s) need(s) to be reconfigured in order to prevent that new incoming connections are redirected to the particular instance which should be shut down.

4. *Observe state of instance intended to be shut down*

   Instead of shutting down the instance instantly (and close any active connections to this instance) the instance can be monitored to ensure that there are no open connections when shutting down. This information can be obtained by the OpenSIPS load balancing server which needs to track the active OpenSIPS connections as described earlier.

5. *Shut down instance*

   Once the load balancer has been reconfigured to ensure that new incoming connections will not be redirected to the particular VM that should be shut down and when there are no active connections to that VM it can be shutdown directly.

2 Implementation

The implementation is done using python 2.7 and django 1.4 as web framework.

2.1 Intercloud Cloning

The Intercloud Cloning Service is part of the Intercloud Layer of the ASMONIA Collaborative Architecture, which is depicted in Figure 3. But this part of the Intercloud Layer depends on the Intercloud Service Management, which will be discussed in a separate section. The Intercloud Cloning is implemented by creating API calls for the steps introduced in chapter 1.2.

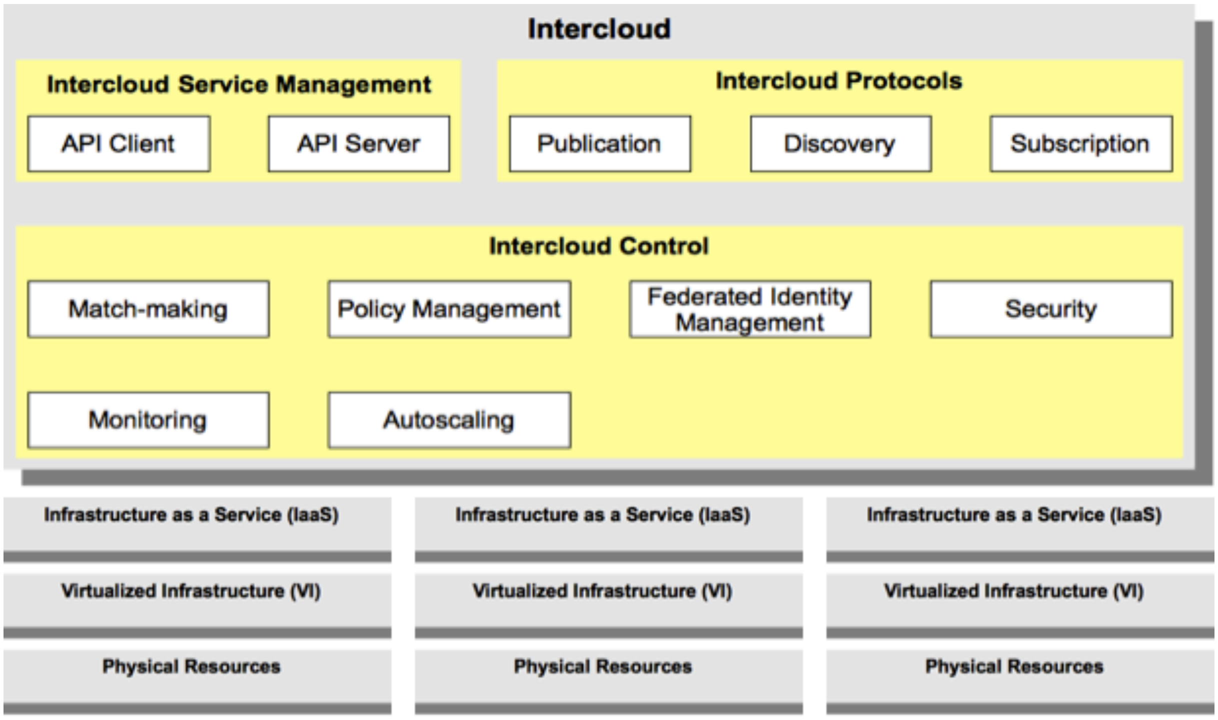


Figure 3: Intercloud Layer of the ASMONIA Collaborative Architecture [1]

2.1.1 Intercloud Service Management

In order to realize the ASMONIA Collaborative Cloud Architecture it is necessary to establish a communication link between separate clouds. This link is established by the Intercloud Service Management (Figure 3) and is used by each cloud to access services of other clouds in the ASMONIA Collaborative Cloud Architecture.

Implementing this link basically means to implement a server-side API that provides a generic cloud service API and translates these API calls into API calls of the underlying member cloud, as well as a client that invokes this API. As we use OpenStack as the cloud software for the underlying member clouds OCCI, which with a very similar interface provides a good candidate for this purpose. But integrating OCCI into the demonstrator would have meant to spend considerable effort on an interface acting as an adapter and thus only becoming useful when heterogeneous cloud software stacks are in use, which is not the case for the demonstrator.

Thus the implementation of the demonstrator simplifies the architecture by joining the originally separate middlewares of each cloud together in one middleware that controls both connected clouds. This way no additional effort for creating a facade for the already existing cloud API is needed. Also no additional effort for the creation of a client is needed, because there already exists a python module from OpenStack for this purpose.

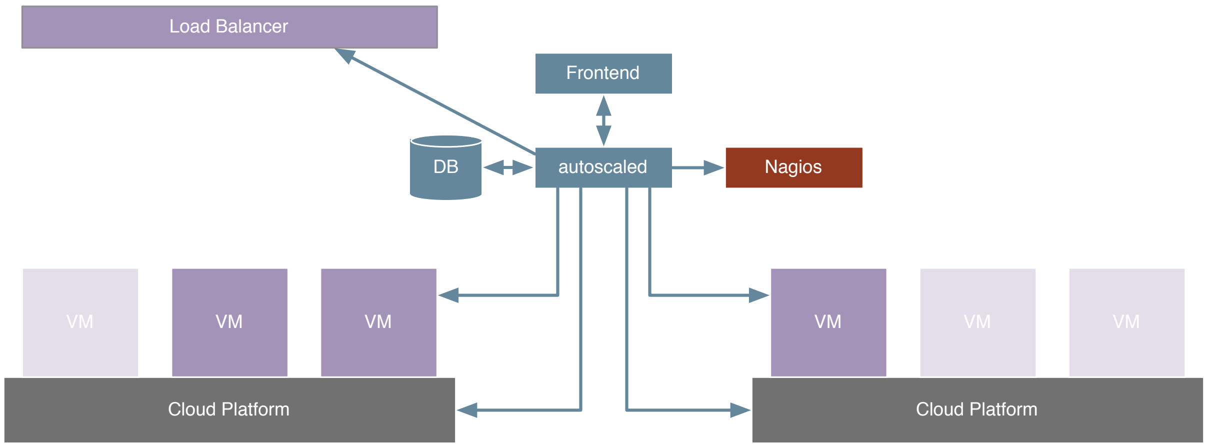


Figure 4: autoscaled connects and reconfigures several components on-demand

As shown in Figure 4, autoscaled is located between all participating clouds. One important fact is that autoscaled is connected to these clouds in order to be able to start and stop VMs as needed. However, this is not sufficient in the ASMONIA Intercloud Cloning scenario. To ensure that service calls will be handled correctly in the Intercloud scenario, autoscaled must also reconfigure the load balancers responsible for distributing incoming requests to the worker nodes. Nagios, the monitoring service, needs also to be reconfigured, because VMs will obtain a different IP address every time they are created. After booting, the particular VM needs to be added to the Nagios server. Different VMs need to be configured in a different manner, e.g., an OpenSIPS VM provides different metrics to be monitored compared to a HTTP webserver. Finally, autoscaled must connect to the VMs if local services need to be (re)configured after the boot process. One example for that is OpenSIPS which needs the VM's IP address in its configuration file.

autoscaled can be seen as the central component which oversees the current state of the relevant components in the Intercloud scenario.

2.2 VM Monitoring

VM Monitoring is realized by having a Nagios server that monitors the virtual machines.

The autoscaled daemon is configured to repeatedly parse the Nagios website and extract the metrics and their corresponding values for each host. The user configures the time interval for the repetition – we use one minute as an interval, because Nagios itself updates at this rate. The monitoring data is stored with a timestamp in the database, so that the Auto Scaling service has access to historic data that it can use as a basis for scaling decisions.

The hosts monitored by Nagios are configured in the Nagios configuration files. Hosts can be grouped in hostgroups referring to a set of hosts, which are monitored in the same manner. In our demonstrator an administrator needs to prepare this hostgroup-configuration, i.e. it must be defined in advance how to monitor hosts running the OpenSIPS service. A hostgroup definition is twofold. First, the hostgroup itself must be defined:

```
define hostgroup{
        hostgroup_name  opensips-servers
        alias           OpenSIPS Servers
}
```

Listing 1: Definition of a hostgroup

Afterwards, a service needs to be defined which specifies the metric to be taken into account for the detection of the status of a remote host:

```
define service{
        use                     generic-service
        hostgroup_name          opensips-servers
        service_description     Registered OpenSIPS Users
        check_command           check_nrpe_1arg!opensips_users
}
```

Listing 2: Defintion of a service

As shown in Listing 2, the check_command check_nrpe_1arg!opensips_users will be executed on all hosts belonging to the hostgroup with the particular hostgroup_name.

On the host side the Nagios Remote Plugin Executor (NRPE) must be installed and executed. In the example shown above, the metric opensips_users will be executed by the NRPE on the host. The NRPE needs also to be configured correctly in order to know what action needs to be performed when a opensips_users command is received from the Nagios server. The configuration on the host side is as follows:

```
command[opensips_users]=/usr/lib/nagios/plugins/opensips -M location-users -w 3 -c 10
```

Listing 3: Configuration of the NRPE service on the monitored host

Whenever the NRPE on the particular host receives an opensips_users command, the Nagios plugin configured in that file will be executed.

This part of the configuration needs to be made manually when setting up the system. Although it looks complicated, these modifications need to be made just once. The configuration tasks on the host side can be performed when OpenSIPS will be installed initially.

When a new VM is created in the cloud, the Nagios server needs the information that a new VM exists with a particular IP address. Therefore, a new host entry is made on the Nagios server itself:

```
define host{
        use             linux-host
        host_name       opensips-1
        alias           OpenSIPS-1 server
        address         138.246.18.151
        hostgroups      opensips-servers
}
```

Listing 4: Definition of a host monitored by Nagios

The host shown in the example in Listing 4 belongs to the OpenSIPS hostgroup, which has been defined earlier (see Listing 1) and can be reached by the specified IP address. Once this host has been defined and the Nagios server has reloaded its configuration file, it will appear in the Nagios monitoring system and its monitoring values can be used by autoscaled. The definition of a host is performed automatically by autoscaled every time a VM is created for scaling reasons. In autoscaled a template is used which defines the information used to update the Nagios server:

```
define host{
       use                          linux-host
       host_name                    $servername
       alias                        OpenSIPS server
       address                      $serveraddress
       hostgroups                   opensips-servers
}
```

Listing 5: Template used by autoscaled to define a Nagios host

As one can see, this template contains 2 variables named $servername and $serveraddress. When a VM is created by autoscaled, these variables are substituted by the runtime information of the particular VM (the hostname as well as the IP address are configured dynamically by autoscaled). Afterwards, the configuration settings will be written to the Nagios configuration file and reloaded.

```
"nagios":{
       "nagios_user": "nagiosadmin",
       "nagios_pw": "nagiospw",
       "nagios_url": "http://10.144.132.130/cgi-bin/nagios3/",
       "ssh_host": "10.144.132.130",
       "ssh_port": "22",
       "ssh_user": "ubuntu",
       "ssh_keyfile": "scaling_daemon/ssh/asmonia",
       "ssh_password": "asmoniaap3",
       "config_hosts": "/etc/nagios3/objects/hosts.cfg",
       "restart_cmd": "service nagios3 reload",
       "templates": {
              "opensips": "scaling_daemon/templates/nagios_opensips_hosts.template"
       }
},
```

Listing 6: Nagios configuration settings in the autoscaled config file

An excerpt of the autoscaled configuration file showing the settings for the Nagios server configuration is shown in Listing 6. Besides the login information provided in this configuration file the template which has to be used for openSIPS VMs is specified as well as the configuration file which holds the hosts configuration of Nagios and the command which needs to be executed to bring Nagios to reload its configuration files. Having a look in the Python source code, Listing 7 shows parts of the method of autoscaled which performs the Nagios server update.

```
def update_config(self):
  # will hold the complete config file later
  config_string = ""

  # load the template files
  for application in self._templates:
    try:
      template = Template(self._get_cfg_template(self._templates[application]))
    except:
      logger.debug('Could not load template file %s' % self._templates[application])
      break

    for hostname, address in self._list_of_hosts[application].iteritems():
      # variables which will be substituded in the template
      config_string += template.substitute(servername=hostname, serveraddress=address)

    logger.debug('Successfully built configuration file:\n%s' % config_string)

    # write to config file
    # need to use sudo if we are not root
    logger.debug('Update file %s on nagios host %s' % (self._config_hosts, self._ip))
    use_sudo = False if self._username == 'root' else True

    self._conn.write_file(self._config_hosts, config_string, sudo=use_sudo)

    # restart service
    self._restart_service()
```

Listing 7: Method in autoscaled which reconfigures Nagios

This listing shows that all templates defined in the autoscaled configuration (see Listing 6) are loaded and the variables servername and serveraddress are substituted and the resulting string is stored in the config_string variable. Subsequently, autoscaled checks if sudo needs to be used. The problem is that Nagios configuration files are only writable for the root user. Recent Linux distributions do not allow login for root for security reasons, so sudo needs to be used when logging in as a non-root user. Afterwards, the Nagios configuration file will be written to the server using the write_file() method provided by _conn which represents the SSHConnection object to the Nagios server. Finally, the _restart_service() method is called which executes the restart command configured in the Nagios-section of autoscaled shown earlier.

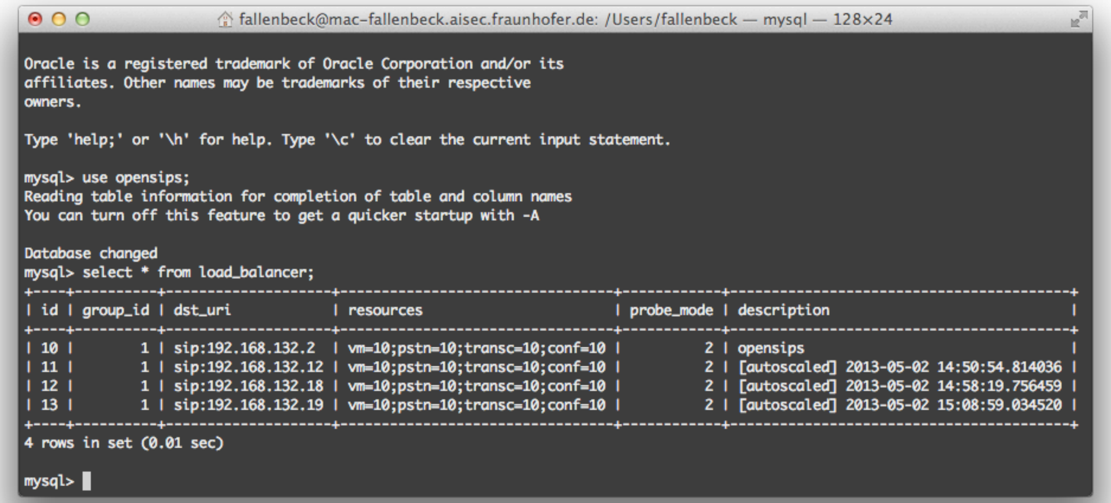
2.3 Load Balancing

For load balancing purposes an OpenSIPS load balancer is used. This load balancer distributes new incoming requests for the OpenSIPS service to the hosts, which are available to process the requests at this time. In general, two different scenarios can be realized.

1. A load balancer machine is always available even if only one single OpenSIPS host is available to process incoming requests
2. If only one OpenSIPS host is available, requests are directly forwarded to this host by the router. If an overload situation appears and the system scales up to at least 2 OpenSIPS VMs an additional load balancer is also booted and the router will deliver incoming packets to the load balancer instead of one OpenSIPS VM directly.

Both variants have their own advantages. While the first alternative is very simple to realize, it has the drawback that the load balancer machine is always running and needs resources even if only one OpenSIPS VM is running and the load balancer always redirects incoming requests to this host. The second alternative does not use as many resources as long as only one OpenSIPS VM is up. When a second OpenSIPS VM is created, one not only needs to reconfigure this VM as stated in Section 2.2, but an additional VM containing the load balancer service needs to be created and configured appropriately. Moreover, one has to make sure that incoming requests will now be redirected to the load balancer itself — instead of the still running OpenSIPS VM — which then redirect the connections to the available VMs. That can be achieved by allocating the IP of the first OpenSIPS VM to the load balancer or by reconfiguring the router, which forwards the IP packets to the OpenSIPS VM. However, because of the fact that the second alternative results in a huge administrative and configuration overhead, the proposed ASMONIA demonstrator focuses on the first approach.

The OpenSIPS service itself can act as load balancer. Therefore, one OpenSIPS instance needs to be configured accordingly. The load balancer configuration is stored in a MySQL database which is also accessible from the OpenSIPS instances which will handle incoming requests afterwards (workers).



```
fallenbeck@mac-fallenbeck.aisec.fraunhofer.de: /Users/fallenbeck — mysql — 128×24

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use opensips;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from load_balancer;
+----+----------+--------------------+---------------------------------+------------+-----------------------------------------+
| id | group_id | dst_uri            | resources                       | probe_mode | description                             |
+----+----------+--------------------+---------------------------------+------------+-----------------------------------------+
| 10 |        1 | sip:192.168.132.2  | vm=10;pstn=10;transc=10;conf=10 |          2 | opensips                                |
| 11 |        1 | sip:192.168.132.12 | vm=10;pstn=10;transc=10;conf=10 |          2 | [autoscaled] 2013-05-02 14:50:54.814036 |
| 12 |        1 | sip:192.168.132.18 | vm=10;pstn=10;transc=10;conf=10 |          2 | [autoscaled] 2013-05-02 14:58:19.756459 |
| 13 |        1 | sip:192.168.132.19 | vm=10;pstn=10;transc=10;conf=10 |          2 | [autoscaled] 2013-05-02 15:08:59.034520 |
+----+----------+--------------------+---------------------------------+------------+-----------------------------------------+
4 rows in set (0.01 sec)

mysql> 
```

Figure 5: Load balancer information stored in the database

The information stored in the database is displayed in *Figure 5*. It contains the URI which is used to connect to the worker as well as the resources each worker is able to provide like the number of slots usable for direct connections or conference calls. The load balancer redirects new incoming call requests to the available workers based on the runtime information and the configuration in the database.

To add a new OpenSIPS host to the load balancer, the information of the particular host needs to be added to the database. Restarting the OpenSIPS service is not necessary.

```
def _add_opensips_host(self, ip, vm=10, pstn=10, transc=10, conf=10, probe_mode=2):
  # use a timestamp as description to show when the entry was made
  host_description = '[autoscaled] %s' % datetime.datetime.today()
  query = "INSERT INTO %s \
    (`group_id`, `dst_uri`, `resources`, `probe_mode`, `description`) \
    VALUES (\'1\', \'sip:%s\', \'vm=%d;pstn=%d;transc=%d;conf=%d\', \'%d\', \'%s\')\
    " % (self._db_table, ip, vm, pstn, transc, conf, probe_mode, host_description)
  self._execute_sql_query(query)

def _del_opensips_host(self, ip):
  query = "DELETE FROM %s WHERE `dst_uri` = \'sip:%s\' LIMIT 1" % (self._db_table, ip)
  self._execute_sql_query(query)

def _execute_sql_query(self, query):
  # build command
  cmd = "%s -u%s -p%s -h %s %s <<'EOF'\n %s" % (self._db_mysql_bin, self._db_user,
self._db_pass, self._db_host, self._db_dbname, query)
  # create subprocess and execute command
  p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
p.wait()
```

Listing 8: Methods to add and remove OpenSIPS host to load balancer configuration

Listing 8 displays the method used to add a particular IP address assigned to an OpenSIPS host to a load balancer configuration. When adding a new host to the configuration, it is also necessary to specify the resources provided by this particular machine:

- vm specifies the number of voicemail channels

- pstn specifies the number of channels to Public Switched Telephone Networks (PSTN)
- transc specifies the number of channels used for transcoding
- conf specifies the number of channels usable for conferences

The probe mode can also be specified which is used to monitor this OpenSIPS machine.

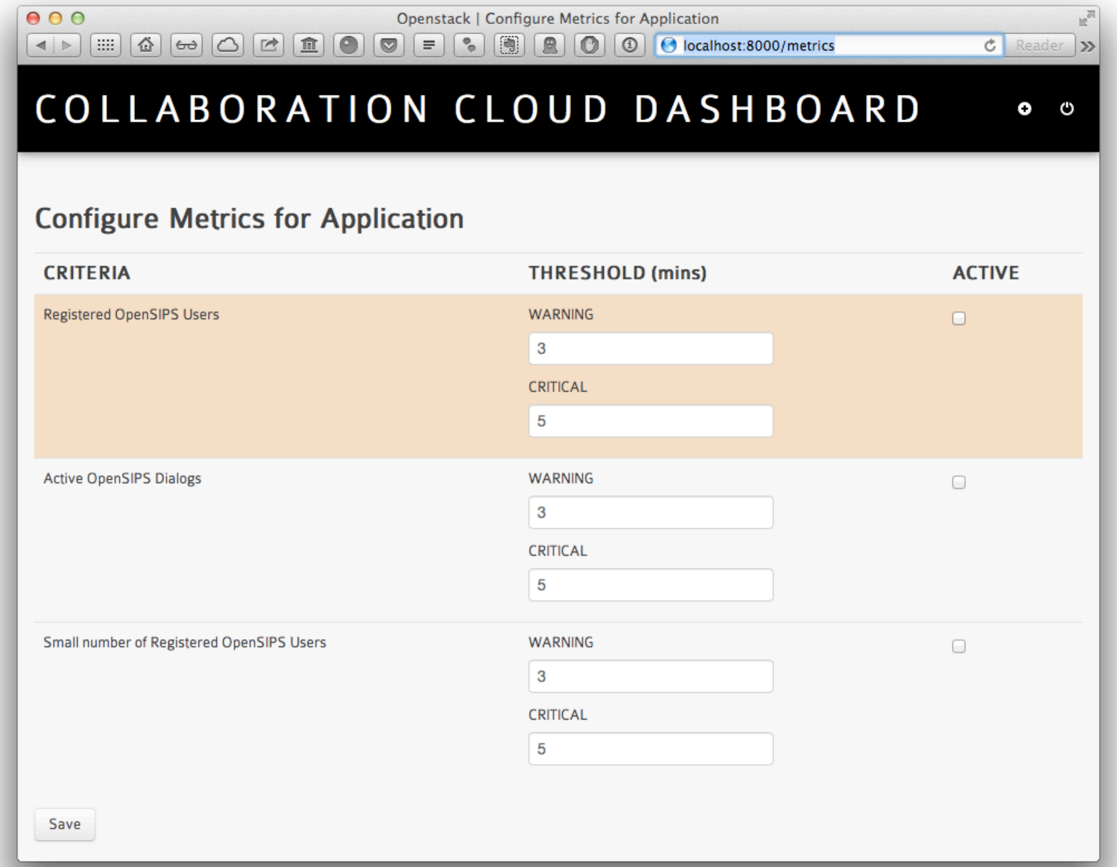
0	Deactivate probing
1	Activate probing the host when it is in disabled mode
2	Always probe the host regardless of it's state

This information is stored in the database by creating and executing an appropriate MySQL query. The `_execute_sql_query()` method will connect to the database server by using the locally installed MySQL client tools, which will be used by the load balancer instance to decide where to redirect incoming calls.

2.4 Auto Scaling

The autoscaled daemon checks repeatedly the values of all metrics. This is done using the same time interval as the VM Monitoring uses to update the monitoring data in the database. For each metric that is registered for a virtual machine type the user can configure, whether the metric should be active, meaning that it is checked by the autoscaled daemon and considered for the auto scaling decision. The user also configures time intervals for the metric values WARNING and CRITICAL separately for each metric, that represent a threshold, which when exceeded signals the autoscaled daemon that it needs to take action. Figure 6 shows the configuration mask in which the administrator can configure the thresholds defining when the scaling process will be initiated.

If the value of a metric is WARNING or CRITICAL than the autoscaled daemon checks how long this value has been registered by checking the historic monitoring data.



COLLABORATION CLOUD DASHBOARD

Configure Metrics for Application

CRITERIA	THRESHOLD (mins)	ACTIVE
Registered OpenSIPS Users	WARNING 3 CRITICAL 5	☐
Active OpenSIPS Dialogs	WARNING 3 CRITICAL 5	☐
Small number of Registered OpenSIPS Users	WARNING 3 CRITICAL 5	☐

Save

Figure 6: Frontend to configure thresholds

The administrator can define a timespan in which a metric of a VM of the particular type must be in a given state until the scale up or scale down process will be triggered. If the timespan exceeds the limit configured by the user, e.g. the value WARNING has been recorded consecutively for the past 4 minutes while the user configured the limit to be 3 minutes, than the autoscaled daemon starts the scaling process after 3 minutes. The states are derived from Nagios which employs 4 states as described in Section 1.3.

For performance reasons, the upscale process can be divided into two phases, a preparation phase and the actual scaling phase. This can be mapped to the threshold values in different ways, e.g. by defining that the WARNING threshold starts the preparation phase and that the critical threshold starts the actual scaling phase. The preparation phase includes the creation of the snapshot of the running VM and its transfer to the remote cloud in which the new machine should be launched. The actual scaling phase can then immediately start a new VM, since the time-consuming tasks like snapshotting and data transfer have been already completed.

Some information regarding nagios.py has been provided in Section 2.2 while aspects of load_balancer.py have already been discussed in Section 2.3. In this section we briefly discuss the decision-making process whether to scale up a virtual machine.

```
values = list(MonitoringData.objects.order_by('-id').filter(vm_id=vm.id,
metric_id=metric.id))[:self.range_of_values]
oks = 0
warnings = 0
criticals = 0
for value in values:
        if value.status == 'O':
                oks += 1
        elif value.status == 'W':
                warnings += 1
        elif value.status == 'C':
        criticals += 1
```

Listing 9: Getting the monitoring values

Listing 9 shows the query of the monitoring data from the database. The data is queried for each VM and for each metric. Only a predefined number of values is used (self.range_of_values) which represents the minimum considered timespan, which is defined in the frontend by the administrator (see Figure 6). Because of the fact that monitoring information is stored as a character representing the state of the VM (O for OK, W for WARNING and C for CRITICAL) we need to count the values to be able to analyze it.

```
if metric.name == self.scaledown_metric:
  logger.debug('%s is the scaledown metric!' % metric.name)
  # are we allowed to add another vm to the scaling process?
  # we've got to check the current and scheduled vms for that
  count_scheduled_vms = [vm_type for (cloud, vm_type) in
InformationManager.vms_to_scale_down.itervalues()].count(vm.vm_type)
  # making sure the threshold is not underrun
  # means to check that the number of existing vms minus the ones
  # that are scheduled for shutdown is still above the threshold
  if vm_type.min_vm_number < len(vms) - count_scheduled_vms:
    logger.debug('Schedule VM %s for shut down' % vm.vm_id)
    InformationManager.vms_to_scale_down[vm.vm_id] = (vm.cloud.name, vm.vm_type)
  else:
    logger.debug('Will not schedule VM %s for shut down because min_vm_number (%d)
reached' %(vm.vm_id, vm_type.min_vm_number))

else:
  logger.debug('%s is a normal metric' % metric.name)
  count_scheduled_vms = [vm_type for (cloud, vm_type) in
InformationManager.vms_to_scale_up.itervalues()].count(vm.vm_type)
  if vm_type.max_vm_number > len(vms) + count_scheduled_vms:
    logger.debug('Schedule VM %s for scaling up' % vm.vm_id)
    InformationManager.vms_to_scale_up[vm.vm_id] = (vm.cloud.name, vm.vm_type)
  else:
    logger.debug('Will not schedule VM %s for scaling up because max_vm_number (%d)
reached' %(vm.vm_id, vm_type.max_vm_number))
```

Listing 10: Mark VMs for upscaling and downscaling

Once the values have been analyzed, particular VMs are marked for scaling up or down respectively. Listing 10 shows the part of the code used for this task. VMs that are marked for scaling are added to one of two specific lists. One list contains the VMs, which need to scale up, the other list contains all VMs marked for scaling down.

In our setup one single metric is defined as the scaledown_metric. If the defined thresholds are exceeded for this metric, a VM will be marked for scaling down. Every other metric causes autoscaled to mark the particular VM for scaling up (the else branch in the listing). Because the administrator can define minimum and maximum numbers of VMs for a certain type, it is important to check if these numbers have been reached in order to prevent that these values are exceeded. Finally, the VM (the VM's ID) is added to the particular list.

```
def _scale_up_vms(self):
  # iterate the upscaling ids
  if len(InformationManager.vms_to_scale_up) > 0:
    for vm_id, (cloud_id, vm_type) in InformationManager.vms_to_scale_up.iteritems():
      # check if the threshold would be exceeded after cloning the machine
      vm_type = VirtualMachine.objects.get(vm_id=vm_id).vm_type

      # create cloud connector (use the config manager)
      cc = CloudConnector(self.manager, cloud_id)

      # determine the name of the new VM
      vm_name = vm_type.type_name

      # clone the VM!
      new_vm = cc.clone_virtual_machine(
        vm_id,
        target_cloud = None,
        virtual_machine_name_prefix=vm_name,
        boot_vm = True)

      # update DB
      tenant_id = self.manager.get('cloud/%s/tenant_id' % cloud_id)
      InformationManager.update_vm_list(self._user, cloud_id, tenant_id)
      ip = InformationManager.get_ip_of_vm_in_cloud(new_vm, cloud_id)

      logger.debug('Add IP %s to loadbalancer for type %s' % (ip, vm_type))
      lb = self._lb_manager.get(vm_type)
      lb.add_server(ip)

      # we need to configure OpenSIPS once a OpenSIPS VM is started
      if 'opensips' in vm_name:
        logger.debug('This is an OpenSIPS VM! Configure OpenSIPS...')
        self._configure_opensips(new_vm, cloud_id)
```

Listing 11: Method to scale up VMs

In Listing 11 the method for scaling up the VMs is displayed. The scale down method is similar to the one showed above except for the fact that a VM is shut down and its IP will be removed from the load balancer. The displayed method iterates over all VMs marked for scaling up. Afterwards, the type of the VM is determined and the CloudConnector is initialized. As stated earlier, the CloudConnector object provides methods to shut down or create new VMs in the particular cloud represented by the object. It also provides the clone_virtual_machine() method which is called to live-clone the VM. After cloning, the VM will be booted immediately. The InformationManager queries the information about the VM, including its IP, which is added to the load balancer responsible for the type of the VM. In case that an OpenSIPS VM has been cloned, an additional step is performed which reconfigures the OpenSIPS installation in the newly created VM.

```
def clone_virtual_machine(self, virtual_machine_id, target_cloud = None,
virtual_machine_name_prefix = None, snapshot_dir = tempfile.gettempdir(), boot_vm =
True):
  '''
  There are two cloning modes. The first clones a VM within the cloud managed by the
  CloudController instance of this method, the second will live-clone a VM to
  another cloud specified by the target_cloud attribute. If target_cloud is None,
  the first method will be used.

  :param virtual_machine_id: ID of the VM which should be cloned
  :param target_Cloud:  None or CloudController of the target cloud
  :param virtual_machine_name_prefix: None or prefix used in the VM name
         to distinguish the different VM types
  :param snapshot_dir: directory where snapshots are saved when performing
         intercloud live-cloning (snapshots will be downloaded from the source
         cloud, saved there and uploaded to the target cloud -- directory is only
         used in intercloud cloning and will not be used when a vm gets cloned in
         the same cloud). Default snapshot_dir is /tmp
  :param boot_vm: defines if the cloned VM should be booted automatically
  '''

  snapshot_prefix = "autoscaled-snapshot"
  timestamp = "%.0f" % time.time()

  # create the names
  if virtual_machine_name_prefix is not None:
    snapshot_name = "%s%s%s" % (snapshot_prefix, CloudConnector.type_delimiter,
virtual_machine_name_prefix)
    new_vm_name = "%s%s%s" % (virtual_machine_name_prefix,
CloudConnector.type_delimiter, timestamp)
  else:
    snapshot_name = "autoscaled%s%s" % (CloudConnector.type_delimiter,
virtual_machine_id)
    new_vm_name = "%s%s" % (CloudConnector.type_delimiter, timestamp)

  # set name of the remote image
  remote_image_name = snapshot_name

  # first we need to create a snapshot
  snapshot_id = self.create_snapshot(virtual_machine_id, snapshot_name)

  # Set the CloudController which should be used to perform the operations
  cc = self

  # when cloning from one cloud into another we need to copy the image
  if target_cloud:
    # copy the image to the remote cloud and overwrite the snapshot_id
    snapshot_id = self.copy_image_to_remote_cloud(snapshot_id, target_cloud,
remote_image_name, working_dir=snapshot_dir)

    # set the CloudController to the target cloud
    cc = target_cloud

  # Now boot the VM
  if boot_vm:
    new_vm = cc.start_virtual_machine(new_vm_name,
      snapshot_id,
      self._get_flavor_id(self._get_flavor_list()[0]))

    # if available vm gets a floating ip assigned
    self._assign_public_ip(new_vm)
    return new_vm
  return snapshot_id
```

Listing 12: Method to clone a VM

One of the most important methods is shown in Listing 12. This method is responsible for cloning a VM both inside one cloud in an Intercloud setting. The first step is to create a snapshot of the running VM to be cloned. Therefore a unique name of the snapshot file is created including a timestamp for easy identification later on. The snapshot is created in the cloud running the VM. If the target_cloud is not set, the new VM will be booted in the same cloud. If target_cloud is set, the snapshot image needs to be copied to the target cloud before the VM can be started there. The method returns either the ID of the newly created VM if it has been started or the snapshot ID if no new VM has been started but only a snapshot has been created.

2.5 Specifics of the SIP setting

The general use case that applies for the demonstrator is the mitigation of overload situations in the mobile network provider's core network. For the demonstrator we have chosen a SIP server as core network component. This section provides a short description of the SIP load generators, responsible to activate the thresholds of the NAGIOS monitoring system that is used by autoscaled. The intention is not to perform a real load test, but to activate thresholds in the monitoring system in order to start an Intercloud Cloning of the OpenSIPS SIP server in Collaborative Cloud 1. Figure 7 shows the SIP load generator scenario:

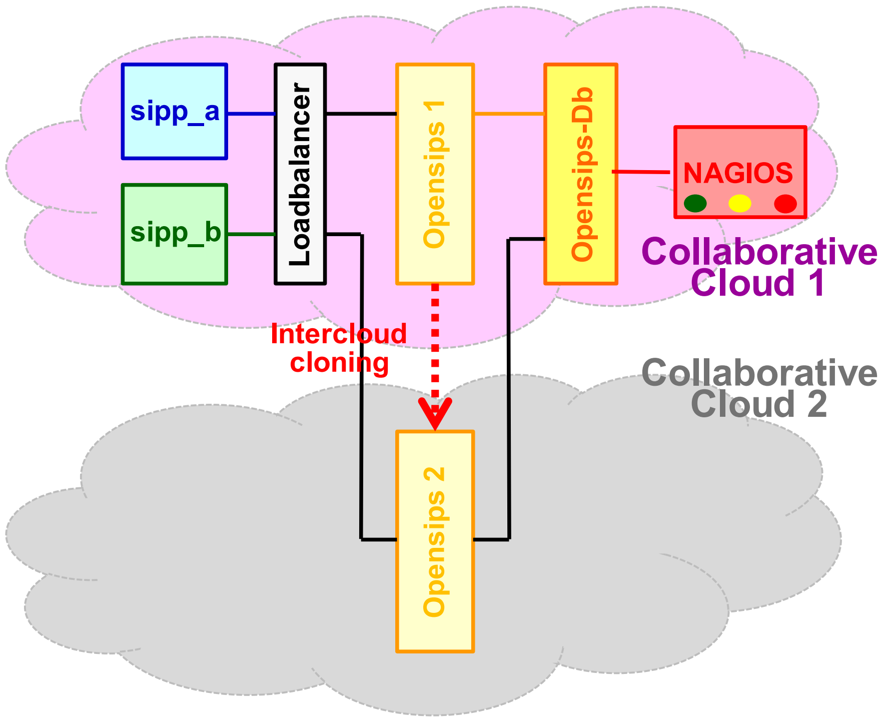


Figure 7: SIP load generator scenario

The two SIP load generators sipp_a and sipp_b, each residing in a separate VM in Collaborative Cloud 1, initiate calls over the SIP server OpenSIPS 1 until it is overloaded. After the VM was in an overload state for a specific time, as configured in the Collaboration Cloud Dashboard and detected by NAGIOS monitoring, an Intercloud cloning process of the OpenSIPS 1 VM to Cloud 2 is initiated. Once the new VM instance OpenSIPS 2 is available in Cloud 2, the load balancer is configured such that the SIP traffic is distributed to both OpenSIPS servers.

For both, sipp_a and sipp_b, separate respective XML-based scenarios have been written that control a call scenario supporting the ASMONIA Intercloud demonstrator. The calling scenarios are designed such that sipp_a always plays the active part with initiating and ending the call while sipp_b only answers the call. The calling scenario applies to the following connection diagram:

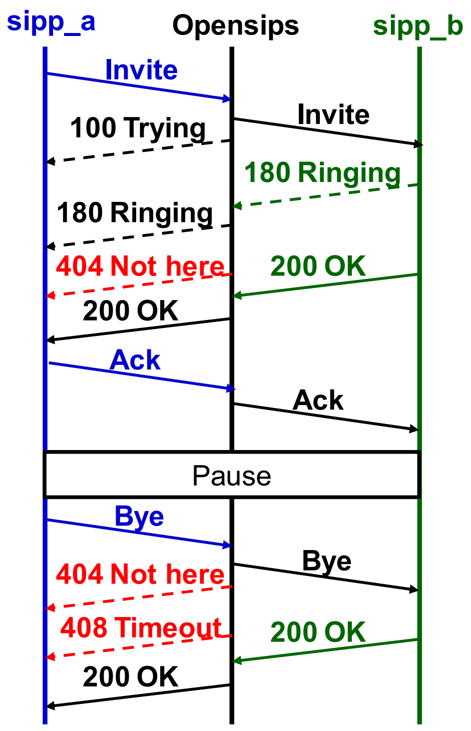


Figure 8: SIPP call diagram

The SIPP call diagram shows the mandatory requests and responses (drawn through lines), the optional responses (dotted lines) and the potential error responses from the OpenSIPS server (red dotted lines). After the call is established, a pause phase is inserted that corresponds to the active call phase of a SIP call. However no real audio path is established, only the SIP signaling is applied.

Figure 9 shows the file structure of sipp_a in more detail.

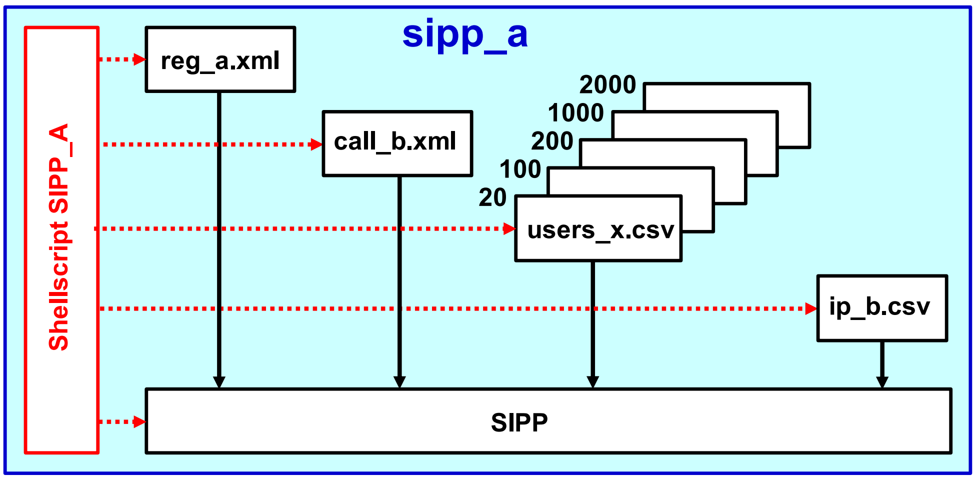


Figure 9: File structure of sipp_a

The load generators sipp_a and sipp_b use the open source SW 'SIPP'. Based on the XML files reg_a.xml and call_b.xml, the behavior of the open source SW SIPP is controlled such that the call scenario as described above is achieved. The file reg_a.xml controls the registering of the users at the OpenSIPS server while the file call_b.xml starts and ends continuously calls with users of sipp_b. While running sipp_a can be controlled via user inputs:

- pressing of '+* increases the number of initiated calls per interval (2 seconds) by 1
- pressing of '*' increases the number of initiated calls per interval (2 seconds) by 10
- pressing of '-' decreases the number of initiated calls per interval (2 seconds) by 1
- pressing of '/' decreases the number of initiated calls per interval (2 seconds) by 10
- pressing of 'p' halts the load generator, next pressing of 'p' starts the load generator again
- pressing of 'q' quits sipp_a smoothly (all started calls are finished)
- pressing of Ctrl C stops sipp_a directly (irrespective of call state)

The maximum number of users is controlled by the csv (comma separated values) files users_x.csv with x providing the values 20, 100, 200, 1000 and 2000 enabling a maximum of 10, 50, 100, 500 or 1000 calls. For sipp_a an additional csv file (ip_b.csv) contains the IP-address and port of sipp_b. The interval of 2 seconds as call setup period is chosen to enable manual control of sipp_a and sipp_b for the ASMONIA demonstrator.

The file structure of sipp_b is equivalent with the exception that the corresponding csv file ip_a.xml is present but not needed.

The configuration of the sipp_a parameters and the execution of the open source software SIPP according to the inserted parameters is controlled by the shell script SIPP_A. Each load generator enables to configure the following parameters:

- to decide whether the old log files (errors.log and messages.log) are deleted or not (default is 'delete' in order to save memory)
- to select the maximum number of calls (10, 50, 100, 500, 1000)

- to start the load generator with ‘Y’

The maximum number of calls must be identical for sipp_a and sipp_b because both VMs are not synchronized (synchronization is achieved by consistent user inputs). Additionally sipp_b must be started before sipp_a. Otherwise errors would occur because sipp_b would not be able to answer the calling attempts of sipp_a. Therefore the start command with the specific parameter ‘Y’ was introduced to enable halting of each load generator if not correctly or in the wrong order configured. This information is also provided by the shell script on the screen during configuration. Neither sipp_a nor sipp_b provide any precautions against misconfiguration. More detailed information is provided in the comments of the corresponding files.

3 Results

This section briefly presents the results of the ASMONIA Intercloud demonstrator with respect to the desired goal ‘provisioning of additional telco resources in other clouds in case of a DOS attack’. Therefore the main security objective is to maintain or increase the availability of cloud-based telecommunication infrastructures

The selected showcase for the ASMONIA Intercloud demonstrator as described in this document is the Intercloud Cloning of an OpenSIPS server VM from one cloud to another in case of overload. Relevant for assessment whether this solution may be beneficial for co-operating telco operators is the period of time that is necessary until the additional resources are available to mitigate the attack.

The key parameters to evaluate the results of the ASMONIA demonstrator are the size of the Intercloud cloned OpenSIPS VM and the bandwidth provided between the two Collaborative Clouds. For the ASMONIA Intercloud demonstrator the following values apply:

- **Size of OpenSIPS VM: ~ 2 Gbyte**
- **Bandwidth between the Collaborative Clouds: ~ 1 Gbps**

Depending on these parameters the behavior and the benefit for larger VM sizes or a larger amount of VMs or a differing bandwidth between the Collaborative Clouds can be roughly estimated.

If the time period to provide additional resources in other telco clouds seems too long, it is always possible to operate with pre-provisioned VM images that can be made available significantly faster compared to VM Intercloud cloning with live snapshots. The disadvantage of this alternative solution is a higher amount of synchronization, resource consumption and version management. The demonstrator focuses on the basic scenario without pre-provisioned images, because it was important to get a basic understanding of the required functionality and to gather real measured numbers as a basis for further discussions. The demonstrator can be extended in order to deal with pre-provisioned images, which basically represents an optimization.

The Intercloud cloning with live snapshots can be separated into multiple phases as shown beneath:

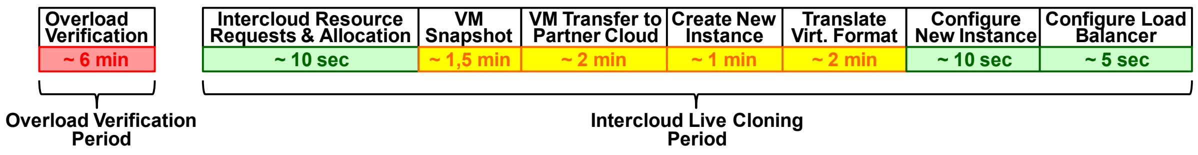


Figure 10: Time period for Intercloud Live Cloning

Below each phase name roughly measured time intervals can be found. The numbers have been gathered during implementation and testing of the ASMONIA Intercloud demonstrator. Please be aware that the width of the table cell does not correspond to the length of the Intercloud Cloning phase in the figure. Instead the length of the time interval is indicated by traffic light colors: from green like ‘negligible’ up to red like ‘overwhelming’.

The Overload Verification Period is not considered for the length of the Intercloud Live Cloning Period. As not in the focus of the ASMONIA Intercloud demonstrator, a rather simple verification method with a large hysteresis is used. And furthermore the size of the hysteresis is configurable in the Collaboration Cloud dashboard and can therefore be reduced by the user if desired. By using more sophisticated methods it is expected to reduce the Overload Verification Period to around 1 to 2 minutes thus turning the color from red to yellow.

To sum up the Intercloud Live Cloning of a 2Gbyte VM over real 1 Gbps links as provided by the existing ASMONIA demonstrator takes around 5 minutes without and 7 minutes with an assumed translation between two different virtualization formats. A more detailed discussion of the evaluation results will be published in the next deliverable D3.4 [2].

References

[1] ASMONIA consortium, “Architecture Concept for the Use of Cloud Systems ,” March 2012. [Online]. Available: http://asmonia.de/deliverables/D3.2_Architecture_Concept_for_the_Use_of_Cloud_Systems.pdf. [Accessed 24 April 2013].

[2] AMONIA consortium, “Evaluation of the Intercloud Demonstrator”, June 2013.

[3] ASMONIA OpenSIPS Nagios script on github.com: https://github.com/fallenbeck/opensips-nagios

Glossary and Abbreviations

Glossary

Term	Explanation	Source
Operator cloud	The operator is the ASMONIA term for a specific implementation of a private cloud. An operator cloud is a cloud system that is part of the infrastructure of a mobile network operator and is only available to the mobile network operator. None of the other collaborating members participating in the ASMONIA system have any access to it.	ASMONIA project
Intercloud	This is a new term currently not reflected in any NIST paper and is getting used to denote cloud architectures with some kind of horizontal federation between single domain clouds.	
Collaborative cloud	The collaborative cloud is a ASMONIA term for a specific implementation of an Intercloud. The collaborative cloud is a cloud that is shared with all collaborating members taking part in the ASMONIA system. All of them may access and use it.	ASMONIA project
Member cloud	The member cloud is an ASMONIA term for a cloud that is owned by an ASMONIA member and contributes to the construction of the ASMONIA Intercloud and as such is part of the collaborative cloud system.	ASMONIA project

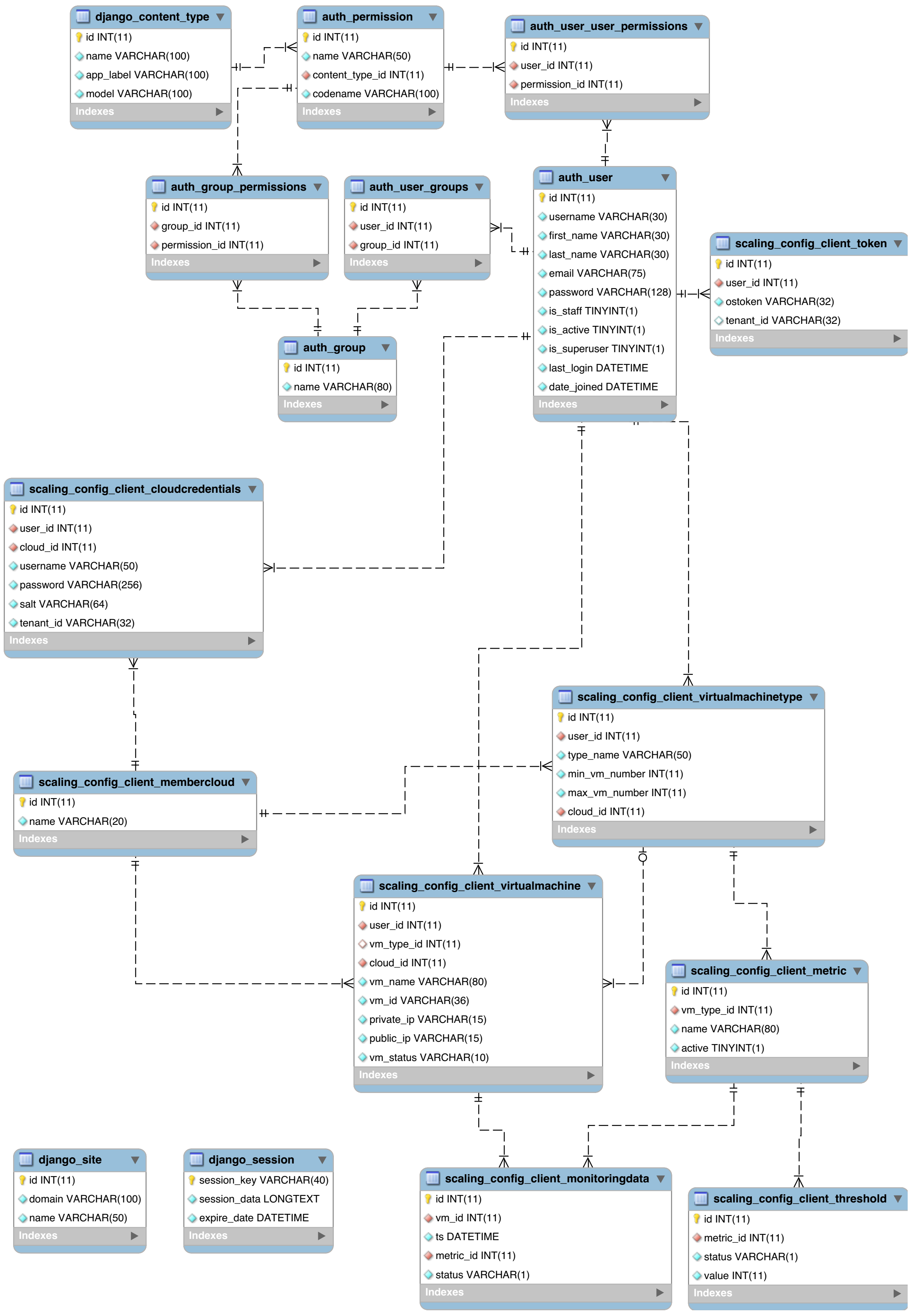
Abbreviations

Abbreviation	Explanation
NIST	National Institute of Standards and Technology

Revision History

Version	Date	Changes
0.1	2013-01-24	Initial version.
0.2	2013-04-22	Changes: • Wrote Chapter 2 • Wrote Chapter 3

Annex A Database Scheme


django_content_type
id INT(11)
name VARCHAR(100)
app_label VARCHAR(100)
model VARCHAR(100)
Indexes
auth_permission
id INT(11)
name VARCHAR(50)
content_type_id INT(11)
codename VARCHAR(100)
Indexes
auth_user_user_permissions
id INT(11)
user_id INT(11)
permission_id INT(11)
Indexes
auth_group_permissions
id INT(11)
group_id INT(11)
permission_id INT(11)
Indexes
auth_user_groups
id INT(11)
user_id INT(11)
group_id INT(11)
Indexes
auth_user
id INT(11)
username VARCHAR(30)
first_name VARCHAR(30)
last_name VARCHAR(30)
email VARCHAR(75)
password VARCHAR(128)
is_staff TINYINT(1)
is_active TINYINT(1)
is_superuser TINYINT(1)
last_login DATETIME
date_joined DATETIME
Indexes
scaling_config_client_token
id INT(11)
user_id INT(11)
ostoken VARCHAR(32)
tenant_id VARCHAR(32)
Indexes
auth_group
id INT(11)
name VARCHAR(80)
Indexes
scaling_config_client_cloudcredentials
id INT(11)
user_id INT(11)
cloud_id INT(11)
username VARCHAR(50)
password VARCHAR(256)
salt VARCHAR(64)
tenant_id VARCHAR(32)
Indexes
scaling_config_client_virtualmachinetype
id INT(11)
user_id INT(11)
type_name VARCHAR(50)
min_vm_number INT(11)
max_vm_number INT(11)
cloud_id INT(11)
Indexes
scaling_config_client_membercloud
id INT(11)
name VARCHAR(20)
Indexes
scaling_config_client_virtualmachine
id INT(11)
user_id INT(11)
vm_type_id INT(11)
cloud_id INT(11)
vm_name VARCHAR(80)
vm_id VARCHAR(36)
private_ip VARCHAR(15)
public_ip VARCHAR(15)
vm_status VARCHAR(10)
Indexes
scaling_config_client_metric
id INT(11)
vm_type_id INT(11)
name VARCHAR(80)
active TINYINT(1)
Indexes
django_site
id INT(11)
domain VARCHAR(100)
name VARCHAR(50)
Indexes
django_session
session_key VARCHAR(40)
session_data LONGTEXT
expire_date DATETIME
Indexes
scaling_config_client_monitoringdata
id INT(11)
vm_id INT(11)
ts DATETIME
metric_id INT(11)
status VARCHAR(1)
Indexes
scaling_config_client_threshold
id INT(11)
metric_id INT(11)
status VARCHAR(1)
value INT(11)
Indexes