Decentralized Authentication in OpenStack Nova

Integration of OpenID

RASIB HASSAN KHAN

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

AALTO UNIVERSITY
School of Science and Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Rasib Hassan Khan

Decentralized Authentication in OpenStack Nova

Integration of OpenID

Master's Thesis
Espoo, June 2011

Supervisors: Professor Tuomas Aura, Aalto University, Finland
Professor Gerald Maguire Jr., The Royal Institute of Technology, Sweden
Instructor: Jukka Ylitalo D.Sc. (Tech.), Nomadic Lab, Ericsson Research, Finland

AALTO UNIVERSITY	ABSTRACT OF
School of Science and Technology	MASTER'S THESIS
Faculty of Information and Natural Sciences	
Degree Programme of Security and Mobile Computing	

Author:	Rasib Hassan Khan	
Title of Thesis:		
Decentralized Authentication in OpenStack Nova Integration of OpenID		
Date:	June 2011	**Pages:** 11 + 88
Professorship:	Data Communications Software	**Code:** T-110
Supervisors:	Professor Tuomas Aura Professor Gerald Maguire Jr.	
Instructor:	Jukka Ylitalo D.Sc. (Tech.)	

The evolution of cloud computing is driving the next generation of internet services. OpenStack is one of the largest open-source cloud computing middleware development communities. Currently, OpenStack supports platform specific signatures and tokens for user authentication. In this thesis, we aim to introduce a platform independent, flexible, and decentralized authentication mechanism in OpenStack. We selected OpenID as an open-source authentication platform. It allows a decentralized framework for user authentication. OpenID has its own advantages for web services, which include improvements in usability and seamless SSO experience for the users.

This thesis presents the OpenID-Authentication-as-a-Service APIs in OpenStack for front-end GUI servers, and performs the authentication in the back-end at a single Policy Decision Point. The design was implemented in OpenStack, allowing users to use their OpenID Identifiers from standard OpenID providers and log into the Dashboard/Django-Nova graphical interface of OpenStack.

Keywords:	Authentication, EC2API, OpenID, OpenStack Nova, OSAPI, Security
Language:	English

AALTO-UNIVERSITETET
Tekniska Högskolan
Fakulteten för Informations och Naturvetenskaper
Utbildningsprogrammet för Datateknik

SAMMANFATTNING AV DIPLOMARBETE

Auktor: Rasib Hassan Khan		
Avhandlingens Titel: Decentralized Authentication in OpenStack Nova Integration of OpenID		
Datum: Juni 2011		**Sidantal:** 11 + 88
Professur: Datakommunikationsprogram		**Kod:** T-110

Utvecklingen av molndatabearbetning är drivande nästa generation av Internet-tjänster. OpenStack är en av de största öppen källkod mellan-programvara datormoln utveckling samhällen. För närvarande stöder IT-plattform specifika signaturer och pollett som för användarautentisering. I denna avhandling vill vi införa en plattformsoberoende, flexibel och decentraliserad autentiseringsmekanism i OpenStack. Vi valde OpenID som en öppen källkod autentisering plattform. Det möjliggör en decentraliserad ram för användarautentisering. OpenID har sina fördelar för webbtjänster, som omfattar förbättringar i användbarhet och sömlös SSO-upplevelse för användarna.

Denna avhandling presenterar de OpenID-Autentisering-as-a-Service APIer i OpenStack för front-end GUI servrar och utför autentisering i back-end i ett enda politiskt beslut punkt. Designen genomfördes i OpenStack, så att användarna kan använda sina OpenID kännetecken från standarden OpenID leverantörer och logga in på Dashboard / Django-Nova grafiskt gränssnitt av OpenStack.

Språk:	Engelska

Acknowledgements

I am grateful to everyone who supported me throughout my thesis work. I would like to specially thank my supervisors, Professor Tuomas Aura and Professor Professor Gerald Maguire Jr. for their excellent supervision and valuable feedback throughout the work.

I would like to show special gratitude to my instructor Jukka Ylitalo from Nomadic Lab, Ericsson Research, Finland, for his constant supervision and valuable feedback during the thesis period. I would like to thank all my co-workers at Nomadic Lab, and specially Abu Shohel Ahmed for his constant support during the work.

I would also like to thank my colleagues from the NordSecMob program and the program coordinators for their support during the whole Master's study period. I am grateful to Prof. Antti Ylä-Jääski from Aalto University, director of NordSecMob programme, for his continual support and motivation.

Finally, I thank my parents for their uninterrupted affection and moral support throughout the period of my study, and all through my life. I would like to thank my special one, friends, family members, and everyone else who supported and inspired me during my whole life.

Espoo, June 2011

Rasib Hassan Khan

Contents

Abbreviations and Acronyms

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
D-H	Diffie-Hellman Key Exchange
EC2API	Elastic Compute Cloud Application Programming Interface
GBA	Generic Bootstrapping Architecture
GUI	Graphical User Interface
HTTP	Hyper-Text Transfer Protocol
HTTPS	Hyper-Text Transfer Protocol Secured
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
MITM	Man-in-the-Middle
OP	OpenID Provider
OSAPI	OpenStack Application Programming Interface
PaaS	Platform as a Service
PAP	Policy Administration Point
PEP	Policy Enforcement Point
PDP	Policy Decision Point
RBAC	Role Based Access Control
REST	Representational State Transfer
RP	Relaying Party
SaaS	Software as a Service
SAML	Security Assertion Markup Language
SHA	Secure Hash Algorithm
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
SSO	Single-Sign-On
VM	Virtual Machine

List of Tables

List of Figures

Chapter 1

Introduction

Cloud Computing is a new paradigm for utilization of scalable resources over the internet. The Pay-Per-Use model for cloud infra-structures has introduced wide interest among users to utilize such services. Major cloud service providers such as Amazon AWS, Rackspace, Salesforce, etc. have driven development of multiple cloud platforms. The most prominent among the open source cloud projects are OpenStack, CloudStack, Eucalytus, and OpenNebula. The open-source cloud platforms provide the ability to deploy private Infrastructure-as-a-Service (IaaS) clouds. Many open-source cloud platforms have compatible application programming interfaces (API) with public clouds such as Amazon AWS, which improves the flexibility and usability of the private clouds. An explanation of the different types of cloud based services can be found in section 2.1 starting on page 5.

1.1 Motivation

As the open source cloud platforms become increasingly popular, usability is being improved with easy to use web interfaces and cloud service APIs. Users can use different client programs or APIs for accessing the different services on the cloud platform. The graphical user interface (GUI) for the platforms, accessible over the network or the Internet, also provide support to access the cloud services in an easy and user-friendly way. Logging into the cloud managerial console gives the users an easy-to-use dashboard, from which they can easily view and manipulate their resources. The management console is a major requirement for administrators, as it gives flexibility and improves usability for managing the users in the system, their privileges, the resources, and over all control of the cloud deployment.

However, these access mechanisms are still being implemented by simple web services. Thus the security threshold is rather low for cloud services. Even though a lot of research remains focussed on intra-cloud network security, data isolation, and policy enforcement; little research has been conducted in the area of security of access points to the cloud platforms. Present authentication techniques only involve processing requests for resources, with plaintext query/response methods to the local database.

1.2 Problem Area

Web GUI has become the most widely deployed front-end for delivering cloud services, both to administrators and users. However, there are certain limitations in what these fronts-ends make available. In the context of authentication of users, there exists the following limitations:

- The primary concern arises, when the architecture for these web GUI front-ends are designed using the same structure as traditional web servers. These front-ends were initially intended to simplify interaction of users and administrators with the cloud platform back-end. This meant, that unlike traditional web servers, these front ends should be 'dumb' servers, i.e. , they are only a window to view the available services. However, this is not what the current trend of development has done, where at present, they are being implemented as regular web servers providing internet services (see section 2.3.4 on page 23).
- The users are required to use username/password based login into the dashboard. This imposes all the disadvantages of such login systems, including limited usability by users, as compared to authentication frameworks such as OpenID [13, 54], Shibboleth [34], SAML [51, 57], etc.
- As the front-end GUI becomes a separate entity, it becomes a requirement for the front-ends to maintain separate user credentials. This requirement arises from the general concept of web services, where there is an absence of a federated login architecture. The absence of a centralized authentication architecture gives rise to the problem mentioned in the next point, of having multiple Policy Decision Points (PDPs).

- As the users log into the web GUI, the web front-end uses API calls to the cloud platform back-end to execute different operations on the cloud. This means, the users are first authenticated at the web GUI, and the back-end cloud platform does not play a role in the authentication process. This introduces multiple PDPs. This is contradictory to general security principles, where it is recommended that there should always be a single point of policy decision, while there can be multiple Policy Enforcement Points (PEPs).

- For the above mentioned reasons, we see that there should be complete trust between front-end GUI and back-end cloud platform. This also means that the front-end GUI can not simply be a "dumb" web server, but has to be more tightly coupled to the back-end.

1.3 Research Goals and Contributions

The initial part of this research included studying multiple cloud platforms. The selection of a suitable cloud platform for developing a prototype was an important step, from the perspective of its architecture, future research scopes, and for further service enhancements. In our research, we worked with OpenStack (see section 2.3 on page 8) for developing the prototype as a proof-of-concept.

The research focussed to introduce a decentralized authentication platform in OpenStack. Thus, we designed and implemented a mechanism to perform OpenID based authentication in OpenStack. However, the target was to keep the front-end GUI as a "dumb" server. Hence, we implemented the OpenID authentication in the back-end of the OpenStack server, with a mechanism to use OpenID as a service with the help of APIs from the front-end. The work included development of a pair of new APIs in the OpenStack API server.

Furthermore, an additional module, the Nova-OpenID-Controller, was incorporated in OpenStack. The implementation followed the current modular architecture of OpenStack, allowing a distributed OpenStack back-end just like all other existing modules.

Additionally, the prototype implementation included extension of the Django-Nova and Dashboard (see section 2.3.4 on page 23) GUI. The APIs were used from the web interface, to allow authentication of a standard OpenID user and to allow such a user to log into the OpenStack management console.

1.4 Organisation of Thesis

The rest of the thesis is organised as follows. Chapter 2 discusses the studies on the various cloud platforms, including the architectural and functional details of OpenStack, the selected platform. The working principles of OpenID authentication is explained in chapter 3. In chapter 4, we have presented the design for using OpenID-Authentication-as-a-Service in OpenStack. The implemented prototype, its architecture and working mechanism has been included in chapter 5. Chapter 6 discusses some analytical perspectives and evaluations on the developed prototype. Finally, chapter 7 provides the summary of the thesis and scopes for future work. Appendix A includes the traces from executing the OpenStack prototype implementation. An applicability analysis and initial design for OAuth in OpenStack is included in appendix B.

Chapter 2

Cloud Computing Platforms

2.1 Introduction to Cloud Computing

There are multiple ways to define Cloud Computing [21]. Ian Foster *et al.* in [38] have defined it as:

> *"A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet."*

Cloud Computing is a relatively new cyber-infrastructure, implying a service oriented architecture (SOA) for computing resources. Users access cloud services over a simple front-end interface to utilize the virtualized resources.

The SOA in clouds is usually defined in a hierarchical structure, as shown in figure 2.1. The layers of cloud computing services in SOA can be described as:

1. **Infrastructure as a Service** (IaaS) provides virtual CPUs, storage facilities, memory, etc. according to user requests. IaaS providers split, assign, and dynamically resize the resources flexibly to build ad hoc systems as requested by clients. Users access the virtualized resources over the network according to their requirements. *Example: Amazon AWS* [3].

2. **Platform as a Service** (PaaS) acts as an abstraction between the physical resources and the service. PaaS providers supply a software

platform and the application programming interfaces (APIs), where users execute their software components. Applications can then be deployed on the platform, together with other services such as cloud storage facilities, without the concern for resource availability, and easily scaling up with increasing utilization of the hosted applications. *Example: Google App Engine* [6].

3. **Software as a Service** (SaaS) provide end users with integrated services from the providers, comprising of hardware, development platforms, and applications. SaaS is a packaged solution and is deployed centrally by the SaaS provider who provides remote access to its users. The users simply utilize the services over a web interface and have minimum visibility of the implementation of the service as compared to IaaS and PaaS. *Example: Salesforce.com* [18].

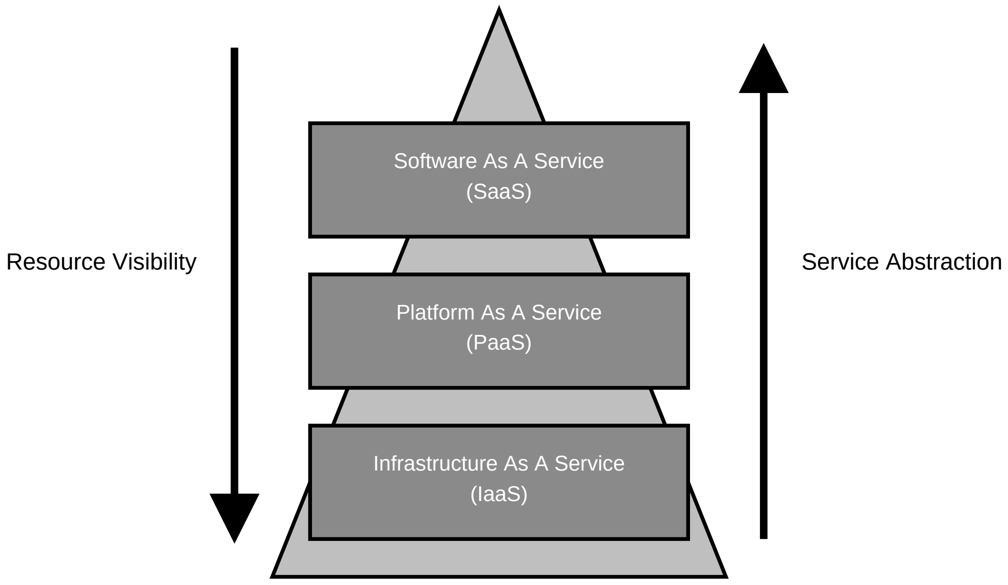


Figure 2.1: Service Oriented Architecture in Cloud Computing

2.2 Selection of A Cloud Platform

Currently, there are multiple cloud platform projects being developed in the open source community. All of them aim to fulfil the requirements of an IaaS provider. They enable the deployment and management, along with the configuration of a multi-subscriber tiered infrastructure of cloud services by enterprises and service providers. However, we will not focus on any of the commercially deployed cloud services, but rather on the open-source platforms available for private cloud deployments.

IaaS cloud platforms provide computing resources: CPU, memory, disk space, and network bandwidth. The middleware application uses an hypervisor running in the back-end to allow the creation of virtual machines (VMs). These VMs emulate physical computers, and each have a CPU, memory, disk, and network resources. The actual physical resources for the creation of the VMs are provided by virtual hosts. The most popular hypervisors in the community are KVM, Xen, VMware, VirtualBox, etc. In the implementation of these middleware platforms, *libvirt* [8] is the most common C/C++ library used to communicate with the hypervisor from the middleware layer.

All middleware platforms are comprised of multiple modules, which can be executed in an array of physical resources. When configured in a distributed environment, they provide a collective service for utilizing virtual resources through the cloud middleware. We explored the deployment and services for the following mainstream cloud middleware platforms: CloudStack, OpenStack, and Eucalyptus.

We studied the different middleware, from the perspective of their scope for the integration of an open source decentralized authentication mechanism. Our study primarily focussed on the following features of the specific cloud middleware platforms.

- **The architectural modularity of the middleware**: This was a significant issue given the objective of our research. Even though all middleware were designed to be deployed in a distributed environment, our concern was the modularity on the different operational units in the cloud and the means of communication between the modules.

- **The authentication framework**: The authentication module required to be a separate module in the architecture. Additionally, we studied the mechanism which was being used to authenticate a request from a user.

- **Detachment of the core resource controller module from the authentication process**: We focussed on a detached core module from the authentication module. This enabled us to handle the authentication process separately, without being concerned with how the actual resources are allocated and distributed.

- **Provision for API services**: We investigated the handling of the API servers on the middleware, and how requests were passed to the core controller. This was an important factor for us to consider, as is explained in chapter 4, when we describe the process of developing *OpenID-Authentication-as-a-Service* APIs for our selected platform.

- **Separation of front-end GUI services from the back-end cloud servers**: With the evolution of web services, it should be noted that the traditional architecture of web based services is not the same as that of cloud services. In cloud servers, as the actual physical resources lie on the back-end, the PDP needs to reside in the back-end, which is not always the case with standard web services. For this reason, we required a certain level of detachment and an absence of dependency of the front-end GUI server from the cloud server running on the back-end.

- **Applicability of OpenID**: OpenID is a widely used open platform for decentralized authentication. We considered the application of OpenID authentication where there was a separation of the front-end GUI from the back-end.

- **Public network exposure of the cloud middleware**: Decentralized authentication using OpenID [13] required an exposure to the public Internet, for interaction with the identity providers. Thus, for security, the specific module responsible for OpenID authentication, should be separated from the core and interacting with the public Internet.

2.3 OpenStack Nova: The Selected Platform

We considered the above mentioned criteria for the selection of a suitable cloud computing platform, and selected OpenStack for our research. This section presents the design and working principles of OpenStack.

2.3.1 Architecture

Nova, the cloud computing middleware fabric controller from OpenStack, is a widely utilized open source project with many contributors. It originated as a project at NASA Ames Research Laboratory and started as open-source software in early 2010.

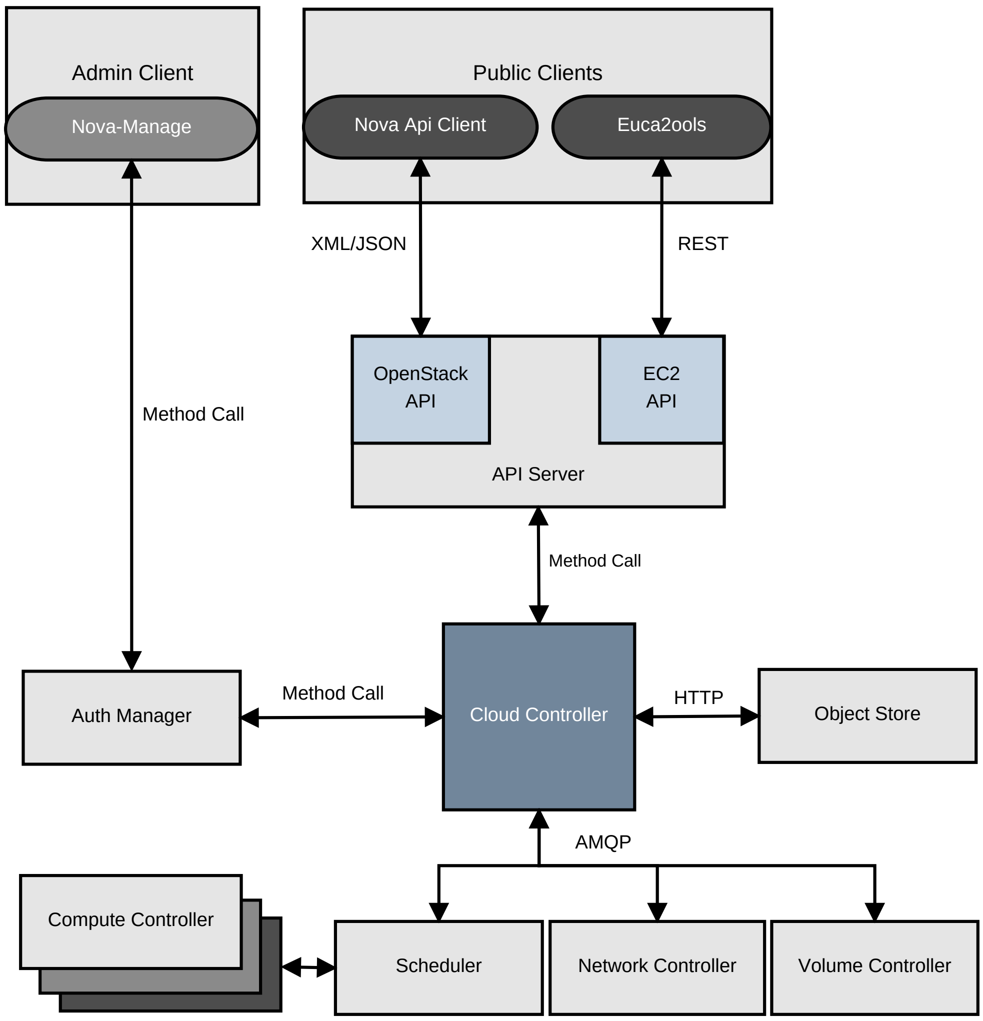


Figure 2.2: OpenStack Nova Architecture

OpenStack supports virtualization with KVM, UML, XEN, and HyperV, using the QEMU emulator. The central core of OpenStack Nova is the Cloud Controller, which interacts with the other modules in different ways. As shown in figure 2.2, the following are the components of OpenStack:

- **Cloud Controller** is the central component which represents the global state, and interacts with the other components.
- **API Server** is an HTTP server which provides two sets of APIs to interact with the Cloud Controller: the Amazon EC2APIs and the OpenStack OSAPIs.

- **Auth Manager** provides authentication and authorization services for OpenStack, which can interact with the Nova-Manage client using local method calls.
- **Compute Controller** provides the compute server resources.
- **Scheduler** is responsible for selecting the most suitable Compute Controller to host an instance.
- **Object Store** component is responsible for storage services.
- **Volume Controller** provides permanent block-level storage for the compute servers.
- **Network Controller** handles the virtual networks for the VMs to interact with the public network.

2.3.2 Authentication and Authorization Framework

The authentication of requests from a user, and the authorizing of resources for the request are handled by the *Auth Base* module. OpenStack uses a *Role Based Access Control* (RBAC) [58] mechanism to enforce policies. For administrative operations, the *Nova-Manage* module is used to interact with the *Auth Base* database (as shown in figure 2.2). The following sub-sections discuss the credentials maintained for each OpenStack user, followed by a description of OpenStack's RBAC model, and details of the *Nova-Manage* module.

2.3.2.1 User Credentials

When a user is created, an *Access Key* and a *Secret Key* are assigned to the user. They can be randomly generated or can be specified by the administrator during user creation. The credentials in the user database for each OpenStack user are shown in table 2.1.

These credentials are used in different ways to authenticate a user's incoming API requests. The use of these credentials will be discussed in section 2.3.3 on page 14. The use of the access key and the secret key differ from their use in EC2APIs (see section 2.3.3.2 on page 16) and OSAPIs (see section 2.3.3.3 on page 20). This difference is due to a design inconsistency in the core of OpenStack, and is currently under consideration for modification.

Table 2.1: User Credentials in OpenStack

Credential	Description
id	Unique identifier for each user
name	Usually, human readable *username* for a user
access_key	Unique, and can be randomly generated or specified during user creation
secret_key	Unique, and can be randomly generated or specified during user creation
is_admin	Set to "1" if an "admin" user is created, or "0" otherwise

2.3.2.2 Role Based Access Control (RBAC)

OpenStack uses a Role Based Access Control (RBAC) [58] mechanism to enforce authorization of resources within the cloud. In RBAC, instead of defining specific privileges for each user, the platform defines a set of permissions based on the user's authorization. Each set of permissions is referred to as a "role" or a "security group", and each user is associated with a specific role. Thus, the role defines the authorized actions when a user attempts to access a service, as shown in figure 2.3.

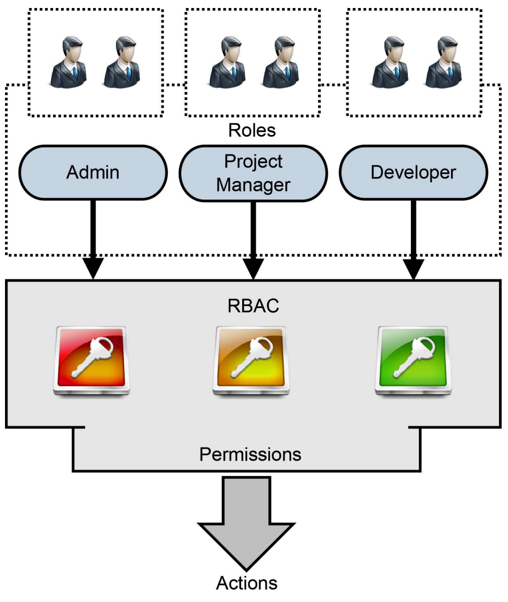


Figure 2.3: Role Based Access Control Architecture

In OpenStack Nova, the administrators are only allowed to define the set of rules for each role, and assign a role to the users. This greatly simplifies the task of management of users and organizational security.

OpenStack Nova defines five different roles for the users, which the administrator has to assign to each OpenStack user. Roles in OpenStack can be global or project specific. The following are the roles used in managing users in OpenStack:

1. **Cloud Administrator (admin)**: Users assigned this role have complete system access and modification privileges. Users can be created with admin roles for specific projects, and will have all the same rights as an administrator. The administrators are allowed to add, remove and modify users, projects, keys, images, and instances. They are also allowed to manage the access network, and modify roles for other users.

2. **IT Security (itsec)**: This is a global role, and is limited to IT security personnel. Users of this role are allowed to quarantine instances. However, this is not one of the commonly used roles in OpenStack deployments.

3. **Project Manager (projectmanager)**: Project owners are assigned this role by default. A project manager is allowed multiple operations, and could alternatively be termed as the administrator for the specific project. Thus, project managers are allowed to perform just the same operations as administrators, but only on the specific project.

4. **Network Administrator (netadmin)**: This role allows users to perform specific network related actions on OpenStack. Thus, a netadmin is allowed to allocate and assign the IP addresses for the VMs, and also manage the firwall policies for the access network.

5. **Developer (developer)**: This is the default role assigned to users created via OpenStack. This is a general purpose role, and the users are required to be added to specific projects. Once added, the users can create and download access keys, and use the access keys to access any running instances within the project. A developer is allowed to use the operations which are available for all user roles, and include viewing instances, images, volumes, keypairs, and additionally, create his own keypair to access the instances.

An overview of the roles and their privileges is summarized in table 2.2

Table 2.2: RBAC Model in OpenStack Nova

Roles	admin	itsec	projectmanager	netadmin	developer
`Global`	yes	yes	no	no	no
`Local`	no	no	yes	yes	yes
`Key Management`	yes	yes	yes	n/a	yes
`Instance Management`	yes	no	yes	n/a	no
`Image management`	yes	no	yes	n/a	no
`Network Management`	yes	no	yes	yes	no
`Project Management`	yes	no	yes	n/a	no
`Create/Modify Firewall`	yes	no	yes	yes	no

2.3.2.3 The *Nova-Manage* Module

As shown in figure 2.2, OpenStack provides the *Nova-Manage* module for administrative tasks such as user, roles, vpn, network, and other management functions. Nova-Manage is a self-executable python command-line tool, with specific commands to directly interact with the OpenStack database. After a user is created, the operations of including the user in a specific project, defining their roles, etc. are all done through the *Nova-Manage* module.

Each *nova-manage* command is in the form:

```
$ nova-manage category command [args]
```

For example, the `list` command in the `fixed` category displays the pool of all existing fixed IP addresses, and is written as:

```
$ nova-manage fixed list
```

The command can also be run without arguments, and it would show the list of available command categories, that is:

```
$ nova-manage
```

A summary of the different command categories available with *nova-manage* is shown in table 2.3. Additionally, if the command category is run without any arguments, it displays all the available operations for the specific

Table 2.3: Commands Categories in *Nova-Manage*

Category	**Commands**	**Description**
`user`	admin, create, delete, exports, list, modify, revoke	Operations related to OpenStack users
`project`	add, create, delete, environment, list, modify, quota, remove, scrub, zipfile	Operations related to OpenStack projects
`role`	add, has, remove	Operations on the OpenStack user roles
`shell`	bpython, ipython, python, run, script	Provides a command line prompt for the supported engines
`vpn`	change, list, run, spawn	Operations on virtual private networks (VPNs) between VMs on the cloud
`fixed`	list	Displays all the fixed IP addresses available for the VMs on the cloud
`floating`	create, delete, list	Network management options for floating IP addresses for the VMs on the cloud
`network`	create, list	Network management options for available networks for the IP pool for created VMs
`service`	disable, enable, list	Operations on the different service modules for the OpenStack deployment
`log`	request	Generate log files for specific service modules
`db`	sync, version	Operations for OpenStack database maintenance
`volume`	delete, reattach	Operations on the volumes available for the VM instances on the cloud
`flavor`	create, delete, list	Management of available hardware configuration for VM instances.

category. For example, the following command displays the operations available for `"user"`:

```
$ nova-manage user
```

2.3.3 Application Programming Interfaces

OpenStack Nova exposes two sets of APIs: the OpenStack API (OSAPI) and Amazon Elastic Compute Cloud API (EC2API). The OSAPI is the list of APIs being developed as OpenStack matures with time. On the other hand, the EC2APIs are a list of comprehensive APIs, designed and defined by Amazon Web Services (AWS) [3]. In all cases, OpenStack relies on Representational State Transfer (REST) to handle the responses from the APIs. The following sub-section discusses REST in detail. Later sub-sections discuss the two sets of APIs for OpenStack Nova.

2.3.3.1 Representational State Transfer (REST)

REST [37, 55, 42, 43] is an architecture for designing web applications. In typical implementations, REST relies on a stateless, client-server, cacheable communications protocol, usually over HTTPS. An example of a RESTful client-server interaction is shown in figure 2.4.

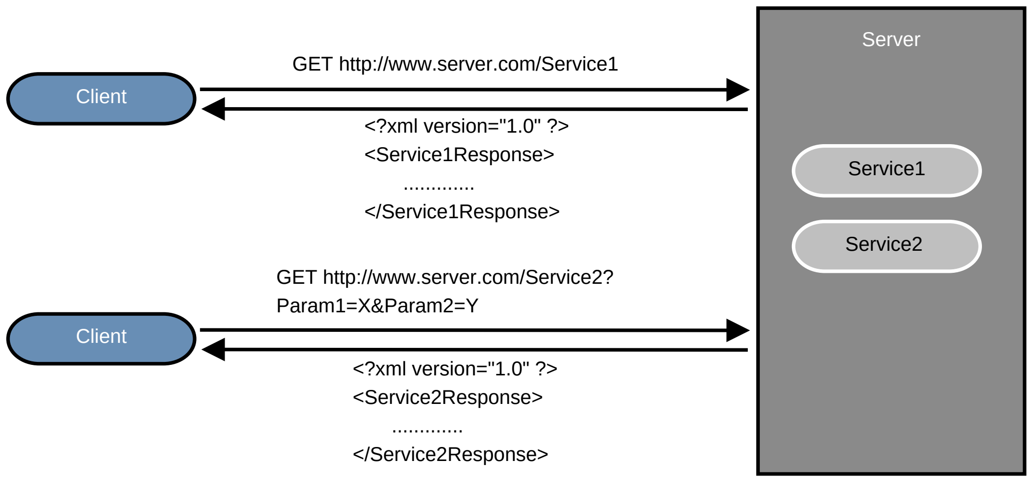


Figure 2.4: A RESTful Client-Server Interaction Example

REST introduces a method for addressing and defining resources and enables us to write web services using a Uniform Resource Identifier (URI) [26]. All clients and servers supporting HTTP/HTTPS can use REST resources with the specific URIs that are implemented by the server.

In REST, the terminology *"resources"* refers to any API residing on a RESTful server. The information being sent by the client to the REST server can be included in the HTTP headers or in the URI, both of which are acceptable by standard REST practices. The response to a REST API call depends on the service provider, and can range from a simple text string to a fully described XML object.

REST requires the API calls to be sent over HTTPS (HTTP running over SSL) for confidentiality, integrity and authenticity. Thus, for security reasons, both the client and the server must implement the SSL technology in RESTful services.

As RESTful services are stateless, each request is required to be handled in complete isolation. Hence each request should include all the required information to invoke the specific API at the RESTful server. The server never stores a *"state"* from previous requests, and the client should resend parameters if required.

The fundamentals of REST include:

- A resource is described as any information or services, which is accessible by a client from a RESTful server. As shown in figure 2.4, *Service1* and *Service2* can be referred as resources.
- Each resource on a RESTful server is uniquely referenced using a URI. (e.g. `http://www.server.com/Service1`)
- Any interaction with a resource residing on a RESTful server is stateless.
- Communication and manipulation of resources are done with the four generic HTTP methods: GET, POST, PUT and DELETE.

All API services on OpenStack are RESTful. However, even though REST requires HTTPS connections for security reasons, current implementation of the API Server on OpenStack does not use HTTPS. Implementation of the API server with HTTPS is a future development task for OpenStack, but not yet incorporated in the framework.

2.3.3.2 EC2 APIs

Amazon EC2 has introduced the EC2APIs, which allows users to interact with Amazon AWS services, along with the server instances in Amazon's data centers. The EC2APIs have also been implemented in OpenStack, similar to their implementation in Amazon AWS [1]. These APIs support Query requests, which are basic HTTP requests to the RESTful API Server. The EC2APIs on OpenStack use the HTTP *GET* method. The data sent as response to the request are serialized as XML objects. However, OpenStack currently do not incorporate any security mechanisms, and the messages are interchanged over HTTP as plaintext.

The APIs in OpenStack are accessible by sending Query requests to the Nova API server. For using these APIs, the *Access Key* and the *Secret Key* are used as the *AWSAccessKeyID* and *AWSSecretKeyID* respectively. By default, the Nova API Server uses TCP port 8773, and upon invoking a specific API, it uses method calls to interact with the Nova Cloud Controller.

All requests include mandatory fields [1], including an *"Action"* parameter, which is used to specify the requested operation with the API. Another important parameter to validate the integrity of the request is the *"Signature"* parameter. The whole HTTP query string is used to generate the signature for a request, with the AWSSecretKeyID (*secret_key*) as the key, and the SHA-256 hash algorithm.

```
Signature = HashFunction (
                    HTTP_Verb + "\n" +
                    Host_Header + "\n" +
                    HTTP_Request_URI + "\n" +
                    QueryString
                   )
```

Therefore, for making an EC2API call, we first generate the signature, and then append it to the arguments in the query string in the REST URI for the specific API. It should also be mentioned, that all text in the URL should be encoded as specified in RFC 3986 [26]. The common parameters along with examples of using them in RESTful EC2API requests are shown in table 2.4.

Table 2.4: Query Parameters for the EC2API in OpenStack

Parameter Name	**Example**	**Required**
`Action`	*Action=DescribeProjects*	yes
`Version`	*Version=nova* (DEFAULT)	yes
`AWSAccessKeyId`	*AWSAccessKeyId=admin* (manually specified) or *AWSAccessKeyId=617367c3-46f2-40ba-8fd7* (randomly generated)	yes
`Timestamp`	*Timestamp=2011-05-02T08:21:56*	yes
`Signature`	*Signature=zPtTNga4ftpjxoPzqRot/tFo62aqM=*	yes
`SignatureMethod`	*SignatureMethod=HmacSHA256*	yes
`SignatureVersion`	*SignatureVersion=2* (DEFAULT)	yes
`Name`	*Name=rasib*	no

The different parameters and their purpose are as follows:

1. `Action`: Used to request a specific action, which is being invoked by the specific API.
2. `Version`: The specific version of EC2API being invoked. This is important in the EC2API set in Amazon AWS [3]. In the case of OpenStack, we use the default value of *"nova"*.
3. `AWSAccessKeyId`: The Access Key ID (*Access Key*) for the user sending the request. They can be automatically generated, or manually specified when creating a user on OpenStack.
4. `Timestamp`: The date and time at the client, when the specific API query call was signed. The timestamp is in the format YYYY-MM-DDThh:mm:ss. Alternatively, Amazon AWS EC2APIs [1] allows the `"Expire"` parameter instead of `"Timestamp"`.
5. `Signature`: The signature of the query request using the required parameters. As mentioned above, this is generated from the query request, and then appended with the query parameters. For authentication, once a request is received, the signature is regenerated, and compared to the one included in the request.
6. `SignatureMethod`: The hash algorithm used to create the signature for the query request. For signing EC2API calls, either SHA-1 or SHA-256 hash algorithms can be used for the signature. OpenStack implements both SHA-1 and SHA-256 hash algorithm for the signature verification in the query requests (similar to AWS EC2APIs [1]). Both SHA-1 and SHA-256 are strong hash algorithms, with SHA-256 being more resistant to collision attacks because of its greater output size. A comparison of both the algorithms is shown in table 2.5.
7. `SignatureVersion`: The signature version which has been used to sign the query request. The default value, for both Amazon AWS, and in OpenStack is *"2"*.
8. `Name`: An optional parameter along with the query request, to specify the name of the user. It is usually used when the `AWSAccessKeyId` is a random long string, and this parameter specifically provides the username in human readable text.

Table 2.5: SHA-1 and SHA-256 Comparison

Algorithm	**Internal State Size(bits)**	**Output Size(bits)**	**Block Size(bits)**	**Max. Message Size(bits)**	**Rounds**
SHA-1	160	160	512	$2^{64} - 1$	80
SHA-256	256	256	512	$2^{64} - 1$	64

The whole query string is used to generate the signature in an EC2API call. For example, for the *DescribeProjects* API, the parameters for generating the signature would look like the following:

```
Signature = HashFunction (
                    GET\n
                    localhost:8773\n
                    /services/Admin/\n
                    AWSAccessKeyId=admin
                    &Action=DescribeProjects
                    &SignatureMethod=HmacSHA256
                    &SignatureVersion=2
                    &Timestamp=2011-03-02T08%3A20%3A38
                    &User=admin
                    &Version=nova
                   )
```

Thus, with all the standard parameters, the format for a query request to the EC2API server on OpenStack, with a *DescribeProjects* action request looks like the following:

```
GET http://localhost:8773/services/Admin/?AWSAccessKeyId=admin&
Action=DescribeProjects&SignatureMethod=HmacSHA256&SignatureVer
sion=2&Timestamp=2011-03-02T08%3A20%3A38&User=admin&Version=nova&
Signature=YKII5A1hCm2iztjT%2BfcDKee0nSxsjrs%3D HTTP/1.0
```

The EC2API server replies to API calls with XML data serialization, with a specific structure for each action. For example, for the *DescribeProjects* action API call above, OpenStack responds with the following XML:

```
<?xml version="1.0" ?>
<DescribeProjectsResponse xmlns="http://ec2.amazonaws.com/doc/nova/">
    <requestId>P7HBV0KXPVL0C628Y7N</requestId>
    <projectSet>
        <item>
            <projectname>AdminProject</projectname>
            <projectManagerId>admin</projectManagerId>
            <description>AdminProjectDesc</description>
        </item>
    </projectSet>
</DescribeProjectsResponse>
```

2.3.3.3 OpenStack APIs

The OpenStack Compute API (OSAPI) [53] was initially designed by Rackspace US [17], the initiators of OpenStack. OSAPI is a RESTful web service interface, and supports both JavaScript Object Notation (JSON) [28] and XML data serialization.

By default, the OSAPI server runs on TCP port 8774 of the OpenStack Server. Each HTTP query request to the OSAPI requires specific authentication credentials. It uses the `X-Auth-User` and the `X-Auth-Key` header values in the query requests to specify the *username* and the *API Access Key* respectively. However, similar to EC2APIs, the OSAPIs also communicates over HTTP with plaintext messages.

For using OSAPIs, the username is the "*Name*" parameter in the user database, and this value is placed in the `X-Auth-User` field. Along with that, the *API Access Key*, is placed in the `X-Auth-Key` field, and is matched against the *Access Key* from the user credentials table in the database. After an initial authentication request to an Authentication URL, if successful, an `X-Auth-Token` is returned by the server.

For authenticating a user to OpenStack with OSAPI, an authentication request would have the following format:

```
GET /v1.0 HTTP/1.1
Host: <OSAPI_Server_Authentication_URL>
X-Auth-User: <username>
X-Auth-Key: <access_key>
```

Once an authentication request is received, the request is validated by comparing the *Access Key* and the *Name* against the `X-Auth-Key` and the `X-Auth-User` respectively. If successful, the `X-Auth-Token` is generated, by a hashing the *username*, the *access key*, and a *timestamp*.

```
Token = HashFunction (
                username + access_key + timestamp
             )
```

All of the required information for the token are stored in the OpenStack database, and are compared against the hash value of the token hash. A successful authentication responds with an "`HTTP 204 No Content`" response with the `X-Auth-Token` in the header, as shown below:

```
HTTP/1.1 204 No Content
Date: <Day>, <Date> <Month> <Year> <Time:hh:mm:ss>
Server: Nova
X-Server-Management-Url: <OSAPI_Server_Authentication_URL>
X-Storage-Url: <NULL>
X-CDN-Management-Url: <NULL>
X-Auth-Token: <Token>
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
```

The `X-Server-Management-Url` is the URL where all operation requests should be made for OSAPIs. The `X-Storage-Url` and `X-CDN-Management-Url` are fields used by Rackspace [17], but do not currently provide any specific features within OpenStack.

At present, the token issued by OSAPI is considered valid for a duration of 2 days, after which, the client has to perform another authentication and receive a new token. An `HTTP 401 Unauthorized` response is sent by the API Server if a request is made with an expired token.

Once the OSAPI client receives the token, it can then send action requests to other APIs on the OSAPI server. In an OSAPI call, the request format (JSON or XML) is specified using the `Content-Type` header value. The desired response format from the server can also be specified in the API requests, and can be different from the request format. The `Accept` header value is used to specify the response format. Additionally, the response format can be referenced using `.xml` or `.json` extension in the query request, which essentially refers to different *"resources"* in the REST request line. A comparison of XML and JSON type request and response formats is shown in table 2.6.

Table 2.6: XML/JSON Serialization Specification in OSAPIs

Format	Content-Type/Accept Header	Query Entension	Default
JSON	application/json	.json	yes
XML	application/xml	.xml	no

An example API request with a JSON header, but specifying an XML return type, for retrieving the list of *"flavors"* would look like the following:

```
GET /flavors HTTP/1.1
Host: http://localhost:8774/
Content-Type: application/json
Accept: application/xml
X-Auth-Token: adflkroiabdl-adf-asdadfwr-agfger
```

The `Content-Type` parameter can be changed to `"application/xml"` to specify an XML request type. Alternatively, the `Accept` parameter could be changed to `"application/json"` to specify a JSON return type. To specify a return type serialization in the query extension, the first line of the request would look like the following:

```
GET /flavors .xml HTTP/1.1
Host: http://localhost:8774/
...
```

Upon successful authentication of the token, an example XML response for the *"flavors"* would look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<flavors xmlns="http://localhost:8774/servers/api/v1.0">
    <flavor id="1" name="Medium" ram="1024" disk="20"  />
    <flavor id="2" name="Large" ram="4012" disk="50" />
</flavors>
```

If the serialization type for the response was specified as JSON, the response would look like the following:

```
{
"flavors" : [
            { "id":1, "name":"Medium", "ram":1024, "disk":20 },
            { "id":2, "name":"Large", "ram":4012, "disk":50 }
      ]
}
```

2.3.4 Web Interface - Dashboard/Django-Nova

The ongoing work in OpenStack is also focussed towards developing an easy-to-use managerial Web GUI. The GUI is built on the Django framework, the "Django-Nova" project, with another wrapper framework for the interface called "Dashboard".

The Django-Nova framework is an implementation layer of the APIs for OpenStack. Currently, the EC2APIs (see section 2.3.3.2 on page 16) implemented in OpenStack provide a more comprehensive list of functions. Thus, Django-Nova uses the EC2APIs to implement the GUI for OpenStack.

Dashboard is a wrapper implementation of the Django framework for OpenStack. It provides the views for the graphical interface. As shown in figure 2.5, Dashboard simply returns the HTML views for the GUI. Internally, it uses python method calls to the Django-Nova framework. Subsequently, Django-Nova implements the EC2API calls, and thus interacts with the API server on OpenStack.

Django-Nova uses an "admin" user account on OpenStack back-end, and requires the access and secret keys for the "admin" user during configuration of the front-end GUI. It uses these keys to interact with the API server, on behalf of the users accessing the services from OpenStack through the Dashboard. Initially, the Administrator account is used to retrieve the specific user credentials from OpenStack. Once received, Dashboard/Django-Nova then uses that user's credentials to further access the resources.

Additionally, the GUI front-end incorporates a separate user management in the Django framework. When a user tries to access the managerial console, the user first authenticates with the Dashboard/Django-Nova authentication system. Once authenticated, the user is allowed access to the managerial console, and then, the HTML views for the dashboard are processed according to the responses from the OpenStack API server.

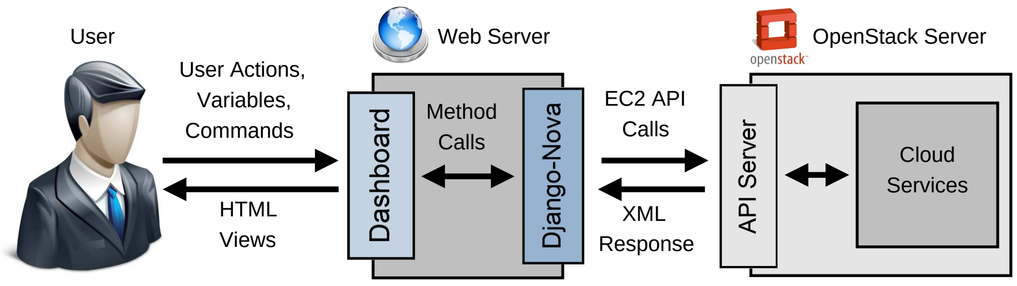


Figure 2.5: DashBoard/Django-Nova with OpenStack

As we will see later, the separate management of user authentication will be the focus of our research. This separate authentication framework is responsible for the incompatibility with standard security practices of having a single PDP in contrast to multiple PDPs (as used is the current design) [25, 40].

2.4 Other Open Source Cloud Middlewares

There are numerous cloud platforms being researched and under construction. While some remain proprietary, some are open-source. In our study, we evaluated the open-source cloud middleware solution provided by *Cloud.com*, CloudStack [27], and another called Eucalyptus [5, 48]. The following sections discuss the architectural specifications of these two platforms. However, there are other well known open-source IaaS platforms, such as OpenNebula [14], Hadoop [7], etc., which for lack of time have not been included in these discussions.

2.4.1 *Cloud.com* CloudStack

The CloudStack open-source cloud computing platform is powered by *Cloud.com*. The community edition is the open-source tool, and has been in development since 2008. CloudStack offers an innovative open-source software technology for deploying either public or private cloud environments.

CloudStack offers a well defined and easy-to-use AJAX enabled graphical web interface, with the core mainly coded in Java. Once the administrator/user logs in, the dashboard offers all the expected services, including the creation of domains, memory allocation, CPU type definition, creation, launching and termination of virtual machine instances, monitoring the performances of the virtual machines, etc. It is a ready-to-deploy package, with a complete set of functionalities.

The architecture of CloudStack introduces three separate components, and a four layer approach, as shown in figure 2.6. The compute controller is the module which manages the VM instances. The compute controller uses a hypervisor to realize the operations. The network controller manages the bridges to the hardware interface, along with the virtual local networks for the VM instances. The storage controller is the module which provides storage facilities, mainly for image management for VM creation, along with VM instance snapshots.

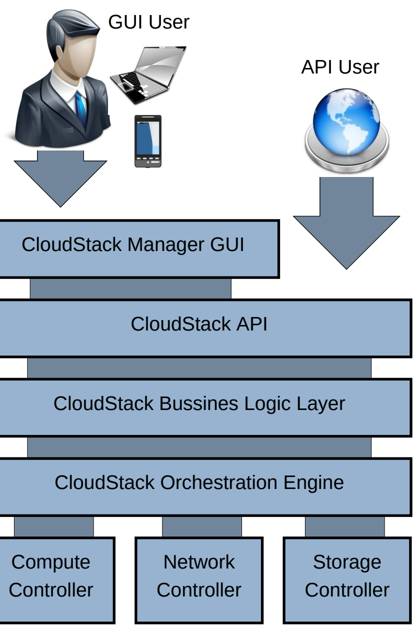


Figure 2.6: Cloudstack Architecture

The CloudStack API server exposes an interface, which developers can use to make API requests with HTTP calls, and interact with CloudStack according to the defined actions. The API requests are in the form:

```
http://<apiURL>?command=<apiCommand>&<param1=val>&<param2=val>
```

CloudStack incorporates two different API URLs. The public API URL allows access for developers and users, and access to this URL must be secured. The private API URL allows full, unsecured access to the entire API server, and is assumed to be secured behind a firewall. Apart from that, there is also the CloudStack Manager, which is a fully integrated web service point, providing a GUI for the users.

Below the interface layer, is the CloudStack business logic layer. This layer incorporates the functionality for authorization and accounting operations. The bottom layer, the Orchestration Engine, directly interacts and schedules the operations of the compute controller, the network controller, and the storage controller.

2.4.2 Eucalyptus

Another popular open-source cloud middleware platform is Eucalyptus. Eucalyptus has been developed at the University of California, Santa Barbara. It is believed to be a portable, modular, and simple IaaS platform for research organizations, and was built to resemble Amazon AWS. Additionally, it includes features such as WS-Security policies [57] with SOAP [63] for secure communication between its internal components.

As shown in figure 2.7, the Eucalyptus architecture is a simple hierarchical structure with node controllers, cluster controllers, and cloud controller. The user interface mainly relies on Java, while the underlying layer in built with Python and C. However, Eucalyptus seemed to have slowed down its development, and is already a matured solution. It is very easy to deploy, and is a relatively integrated product, that does not allow many modifications of its architecture.

The entry point to the Eucalyptus platform is through the cloud controller. This includes an API engine, which includes both REST and SOAP [63] protocols to handle the requests, make high level scheduling operations, and implement them on the cluster controllers. Handling of both REST and SOAP requests in the API engine is completely compatible with Amazon's EC2APIs [1]. Additionally, it includes a web GUI, through which users and administrators can perform different management operations in the cloud. Not shown in the diagram, the cloud controller also incorporates a storage controller, referred to as Walrus. The storage controller provides a mechanism to store and access VM objects and user data, and implements Amazon's S3 interface [2].

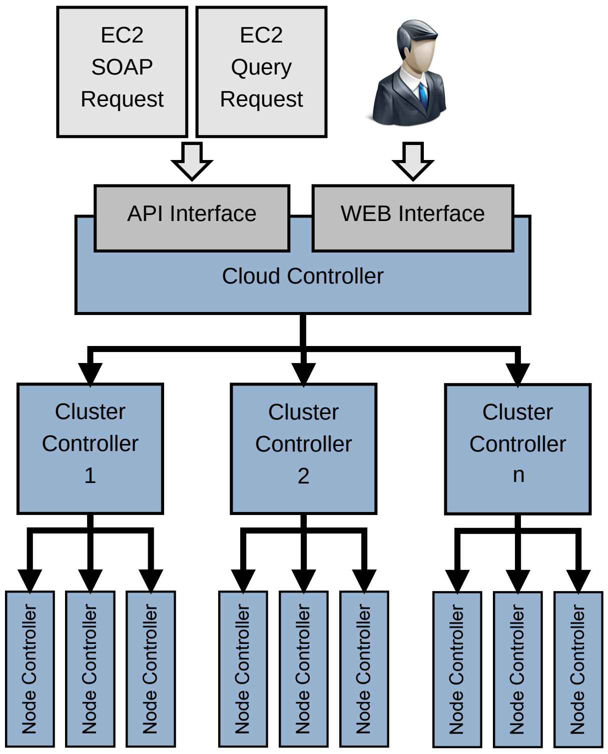


Figure 2.7: Eucalyptus Architecture

A cloud controller can control multiple cluster controllers. Specifically, a cluster controller is responsible for scheduling the specific node controllers in its domain for VM executions. Cluster controllers are also responsible for managing the virtual local networks for the instances. As shown hierarchically, there can be multiple node controllers in each cluster. The node controllers are the specific hosts, which provide the resources for the IaaS, and control the execution, inspection, and termination of the VM instances.

Chapter 3

OpenID Background

OpenID is one of the most popular open source decentralized identity management solutions for web services. In this chapter, we provide a detailed specification of the required information on OpenID [13, 54].

3.1 OpenID Terminology

There are various entities and terms used in the specification for OpenID authentication. Below, we provide the terminologies used in OpenID.

1. ***Identifier:*** An "http" or "https" URI used as the OpenID identity, for a specific user, or for an OpenID Provider.
2. ***User-Agent:*** An HTTP/1.1 [36] client at the user's end.
3. ***Relying Party (RP):*** A web service point which requires user authentication of an OpenID Identifier.
4. ***OpenID Provider (OP):*** An authentication service provider, allowing users to authenticate against a provided OpenID Identifier.
5. ***OP Identifier:*** An Identifier for an OP.
6. ***User-Supplied Identifier:*** An Identifier provided by user. A user may provide either a personal Identifier or an OP Identifier. If an OP Identifier is used, after authentication, the OpenID Provider returns the specific user's Claimed Identifier.

7. ***Claimed Identifier:*** An Identifier finally returned to the RP from the OP, which is linked to a local user identity at the RP.

8. ***Return To URL:*** The URL provided by the RP, where the OP would send the User-Agent back after authentication.

9. ***Trust Root:*** The URL against which the OP will authenticate the user.

10. ***OP Endpoint URL:*** The absolute HTTP/HTTPS URL which performs the user authentication. This is discovered by the RP using the Yadis Discovery Protocol [19] for the Identifier provided by the user .

3.2 Authentication

OpenID is a well known open source authentication mechanism. It provides decentralized user centric identity management for web services, and allows seamless Single-Sign-On (SSO) authentication. The current version of OpenID is 2.0 [13, 54]. Previously, OpenID 1.0 only supported stateful authentication. However, OpenID 2.0 supports both stateful and stateless OpenID authentication. The sequence of a stateless OpenID authentication is shown in figure 3.1. The stateless OpenID authentication mechanism works as follows:

1. The User-Agent first requests a page over HTTP from a web service point, which supports OpenID authentication.

2. The web server returns a page over HTTP to the User-Agent.

3. The user supplies the OpenID Identifier, and the User-Agent submits the value to the web server.

4. The web server acts as an RP. It normalizes the Identifier, and performs the discovery process, using the Yadis protocol [19] (XRI Resolution protocol [61] was used in OpenID 1.0, but is avoided in OpenID 2.0). At this point, the RP receives the meta-information from the OP, required for the redirection of the User-Agent to the OP endpoint URL.

5. The RP then sends an `HTTP 302 Redirection` response to the User-Agent. The redirection includes several different parameters, including

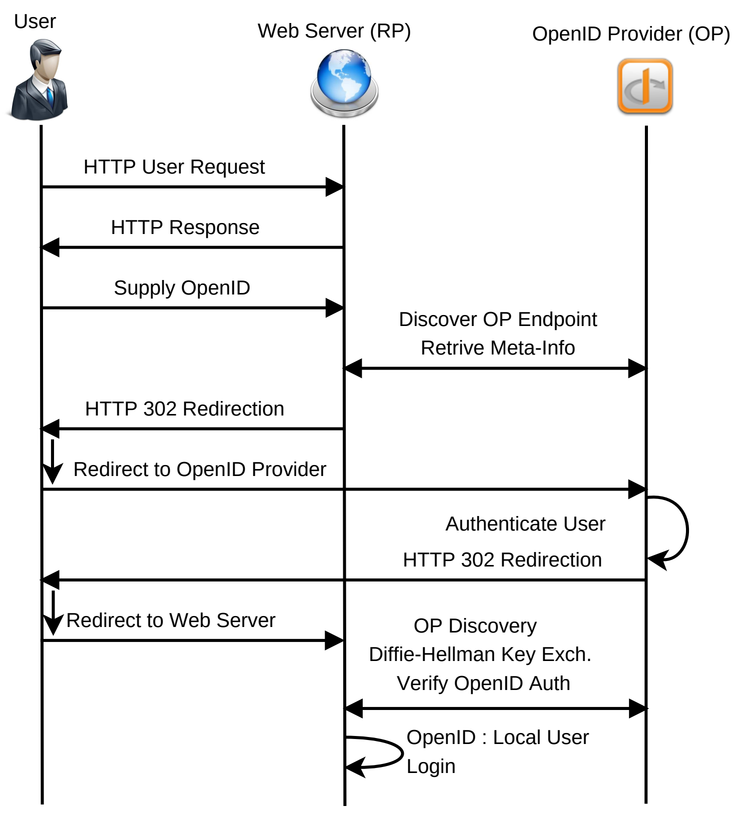


Figure 3.1: Signalling Sequence for Stateless OpenID Authentication

the "`openid.return_to`" URL. The URL contains a nonce value generated by the RP, for example: `http://10.0.0.25:8080/openid/verify /?janrain_nonce=2011-03-23T07:20:56ZLnMsYN`.

6. The User-Agent receives the `HTTP 302 Redirection`, and is redirected to the OP server, at the OP endpoint URL.

7. The OP can use any method to authenticate the user (such as username/password, certificates, smart-cards, generic bootstrapping architecture based device authentication [23], etc). Once authenticated, the OP returns the User-Agent back to the RP, at the `return_to` URL specified by the RP in the previous phase. The OP passes a long string in the `GET` request line, in order for the RP to validate the authentication, also referred to as the assertion URL. An assertion URL usually holds the following parameters:

```
/openid/verify/?janrain_nonce=<nonce sent from RP>
&openid.ns=<OpenID version specification at OP>
&openid.mode=<id_res:OpenID response>
&openid.op_endpoint=<OP endpoint URL>
&openid.response_nonce=<nonce from OP>
&openid.return_to=<RP return URL>
&openid.assoc_handle=<D-H key association handle at OP>
&openid.signed=op_endpoint,
               claimed_id,
               identity,
               return_to,
               response_nonce,
               assoc_handle
&openid.sig=<signature of openid.signed parameter values>
&openid.identity=<authentication user Identifier>
&openid.claimed_id=<claimed user Identifier>
```

8. At this point, the RP verifies the signature. It re-calculates the `openid.sig` from the values of all the parameter names included in `openid.signed`. If verification is successful, the RP parses the `openid.op_endpoint`, and after discovery of the OP, it sets up a key association using Diffie-Hellman (D-H) key exchange [32] and the `openid.assoc_handle`.

9. After the D-H key establishment, the RP verifies the response for the specific `openid.identity` with the `check_authentication` operation for the following fields which the RP has received from the OP: `op_endpoint`, `claimed_id`, `identity`, `return_to`, `response_nonce`.

10. Once the parameters are all successfully verified, referred to as the assertion, the RP links the `openid.claimed_id` with the identity of a local user in its own server and allows the user to login to the service point.

In a "stateful" OpenID authentication, the D-H key exchange [32] occurs in the initial discovery phase at the RP. The "`openid.assoc_handle`" is stored together with the shared key at the RP when the shared key is established. The User-Agent is then redirected to the OP. After authentication at the OP and when the user is redirected back to the RP, there is no D-H key exchange. Instead, the "`openid.assoc_handle`" is used to retrieve the stored key from the previous step, and the received parameter values are then verified. Thus, in this process, it is required that the RP is able to maintain a key database.

3.3 Security in OpenID

Even with widespread use, OpenID is not fully secure from attacks. OpenID 2.0 had significant improvements over OpenID 1.0 [54]. However, any RP or OP, implementing OpenID functionality, should consider some issues. Here, we summarize the different threats that OpenID is vulnerable to, including the results from the OpenID's working group specifications [13], A. Lindholm [46], P. Sovis *et al.* [59], and M. Oostdijk *et al.* [60].

1. **Eavesdropping and Reusing Assertions**: OpenID is vulnerable to eavesdropping attacks if the nonce value in the assertion URL is not checked. This implementation flaw at the RP or the OP would allow an attacker to reuse authentication assertions. However, using an SSL/TLS [31, 44] tunnel to prevent eavesdropping, or the implementation executing a check for nonce reuse in the verification phase, either would prevent assertion reuse.

2. **Wrong Use of SSL/TLS**: Even with SSL/TLS tunnels in the authentication process, there might still be potential for intruder attacks. In some cases, the user might even have a false impression of security and be a victim to Cross-Site-Scripting attacks [30], if non-HTTPS Identifiers are used. To avoid such loopholes, the session of the User-Agent with the RP's service point should be an HTTPS session. Additionally, the OP's `op_endpoint` URL and the RP's `return_to` URL are required to be secured connections for fully secure transaction with the User-Agent.

3. **Man-in-the-Middle Attacks (MITM)**

   (a) *Attribute Exchange Vulnerability*: OpenID Attribute Exchange [12] is an extension to OpenID 2.0, which allows the inclusion of additional parameters to interchange information between the RP and OP. The OpenID assertion URL from the OP to the RP after authentication contains the `openid.signed` parameter, the value of which implies all the different parameters which have been included in the signature, `openid.sig`. However, if the additional parameters in attribute exchange are not included by the OP in the signature, then they are vulnerable to modifications by a MITM.

(b) *Attribute Exchange Parameter Injection*: A MITM can append additional parameters in the assertion URL from the OP to the RP after authentication using Attribute Exchange parameters. Therefore, to detect MITM information injection, the OP should include all the attribute exchange parameters in the signature, and the RP should verify each parameter when checking the `openid.sig`, and verify that all the parameter names are included in `openid.signed`.

(c) *Discovery Tampering*: The discovery process includes DNS address resolution and retrieving the meta-information from the OP. Both mechanisms are susceptible to attacks. If the DNS resolution is somehow manipulated, the MITM can impersonate a false OP. Furthermore, if the MITM can breach the session between the RP and the OP, and tamper with the meta-information, the User-Agent can be redirected as the adversary wants. Security against these attacks could be achieved using mutually authenticated certificates for the information being exchanged between the RP and the OP.

(d) *Adversary Relay Proxy*: The MITM could act as a false RP, and instead of redirecting the user to the OP, could act as a proxy for the OP. Thus, the adversary RP would acquire the user's credentials.

4. **Denial-of-Service Attacks (DoS)**: DoS attacks are a significant concern for any web services [39]. DoS attacks can be launched by a misbehaving RP, by sending repeated OpenID authentication requests to an OP. Additionally, a misbehaving User-Agent can also send repeated requests to an RP, to start the OpenID authentication process. However, using "stateless" OpenID reduces a lot of the pre-processing and storage tasks for the RP. This has been considered in our design for integration of OpenID in OpenStack, which is discussed later in chapters 4 and 5.

3.4 Other Authentication Frameworks

There are other authentication platforms similar to OpenID, which provide similar decentralized authentication mechanisms. Microsoft.NET Passport

[50], and its successor Microsoft CardSpace [47] are two identity management platforms from Microsoft. Shibboleth [56, 34] is an open source federated identity management specification, which uses the Security Assertion Markup Language (SAML) as an internal specification. SAML [57, 51] is an open standard for exchanging authentication and authorization data between security domains using XML. With Shibboleth and SAML, users authenticate themselves to an identity provider, and then the service provider and the identity provider exchange identity credentials in the back-end. A summary of these mechanisms is shown in table 3.1.

Table 3.1: Comparison of Authentication Frameworks

Authentication Framework	**Technology Dependency**	**Application Domain**
OpenID	HTTP, open source	Decentralized, user-centric, between provider and service point with user interaction.
Microsoft.NET Passport & CardSpace	Operating system, client end certificates, licensed technology, WS-Security [57]	Centralized, user-centric, Windows workstation account, between Microsoft Live and service point with user interation.
SAML	XML, open source [57, 51]	Organization centric, between provider and service point without user interaction.
Shibboleth	HTTP, SAML framework, open source [57, 34]	Organization centric, between provider and service point without user interaction.

Chapter 4

OpenID with OpenStack Nova

OpenID allows a decentralized and flexible authentication mechanism for web services. In this chapter, we present the design to integrate OpenID authentication in OpenStack Nova.

4.1 OpenID in OpenStack

OpenStack (see section 2.3 on page 8) performs authentication, based on access and secret keys. Using OpenStack with API tools (euca-tools EC2API client [4], python-nova OSAPI client [16]), is not much of a concern. However, using APIs to interact with cloud services is not a very convenient form of interaction. Thus, the GUI plays a more significant role in the services provided to users. The Dashboard/Django-Nova framework (see section 2.3.4 on page 23) provides a suitable GUI for users.

OpenStack encourages the use of its APIs (EC2API or OSAPI) for implementing front-end GUI services. The weakness in this implementation is that, the users are required to be authenticated in a separate authentication framework in the front-end, with a username/password pair. Once authenticated, the front-end GUI server then uses the Admin credentials to retrieve the user's credentials. This way, the backend OpenStack server never plays a role when the GUI server initially authenticates the user.

Therefore, the current approach for developing a front-end GUI for OpenStack does not follow the standard practices for policy enforcement and management [25, 40]. In standard architecture for such federated login mechanisms, the front-end GUI server is a "dumb" server, providing only views to the User-Agent. The Policy Administration Point (PAP) and the

Policy Decision Point (PDP) are always required to be a single point in any architecture. However, there could be multiple Policy Enforcement Points (PEPs), where the decision by the PDP are enforced.

In contrast to the standards, in the present formation, if OpenID is implemented on OpenStack, the relaying point will reside in the front-end server. As shown in figure 4.1, the GUI server interacts with the OP, authenticates the user, and provides access. This makes the front-end act as a secondary PDP, in parallel to the user administration in the back-end.

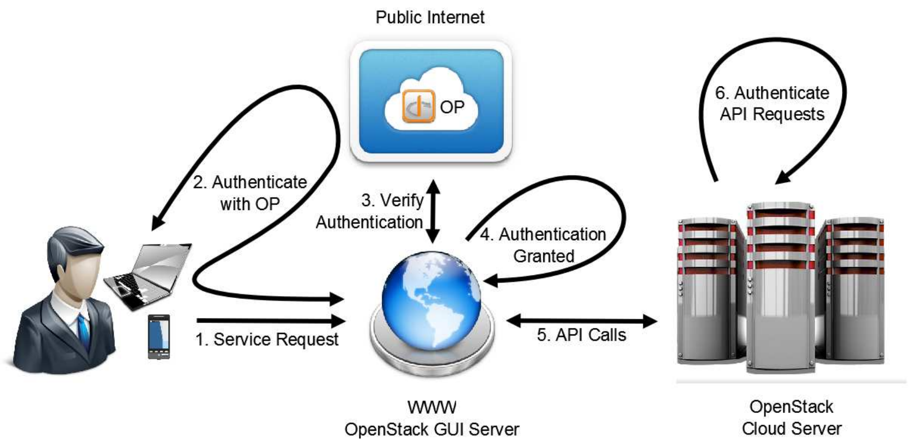


Figure 4.1: Improper Authentication Enforcement Point

4.2 Applicability of OpenID in OpenStack

To apply OpenID authentication mechanism in OpenStack, we first needed to consider the architectural perspective. Implementing OpenID at the front-end GUI server, as shown in figure 4.1, and making it act as a simple RP in the process, was a simple task. However, that was not our target, as we needed to combine a dual-PDP scenario into a single-silo formation.

Mechanisms such as OpenID were targeted for basic web service architectures. In contrast, cloud services such as OpenStack do not necessarily follow such an organisation. As shown in figure 4.2, our design aim was to shift the point of authentication, the PDP, to the back-end OpenStack server.

Shifting the PDP and performing the authentication in the back-end requires the front-end to initiate an authentication request. This way, the authentication decision is made in the back-end within the PAP, the **Auth**

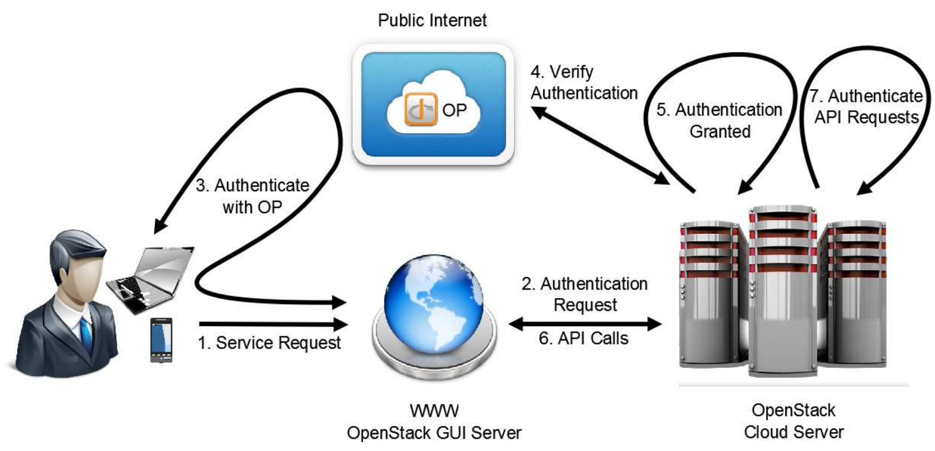


Figure 4.2: OpenID-OpenStack Authentication Overview

Manager module in OpenStack (see section 2.3.1 on page 8).

Enabling OpenID authentication in OpenStack would allow users to log into OpenStack with OpenID URLs. However, the administrator should link an OpenID URL to an existing OpenStack user to enable OpenID authentication for the user. This is a standard practice in all web services providing OpenID authentication, where a local user is linked to his specific OpenID URL (see section 3.2 on page 29).

4.3 OpenID as a Service

Performing OpenID authentication on the back-end would combine the decision points into a single PDP. Thus, we propose a design to implement OpenID-Authentication-as-a-Service on the back-end. The following sub-sections present the factors considered in the design, followed by the detail messaging sequence to utilize the OpenID authentication service.

4.3.1 Service Design Concept

While designing an architecture to integrate OpenID with OpenStack, we converged on a solution based upon OpenID-Authentication-as-a-Service for OpenStack. We considered the following issues when designing the solution:

- The front-end GUI should be a "dumb" server, only processing views.

- There should not be any requirement for the GUI server to maintain any user credentials for authentication.
- Even though the views on the GUI are based on responses from the API server, the front-end should interact with the **Auth Manager** in the back-end to provide the initial authentication grant.
- The HTTP User-Agent only interacts with the front-end server. The back-end OpenStack server should not have any direct communication with the User-Agent.
- The process of authentication of a user in the front-end should be realized as a service from the back-end server.
- All interaction between the front-end and the back-end should be stateless, as required by the RESTful API server [55, 37, 1, 53].
- Ensure that we meet all of the specifications of OpenID [13, 54].
- Simplify implementation requirements at the front-end server.
- Ensure all security requirements for OpenID in all phases of interaction, as discussed in section 3.3 on page section 32.
- Maintain a modular and distributed structure for the integration points, in order to comply with the current architecture of OpenStack.

4.3.2 OpenID Authentication Service API

As mentioned in section 3.2 on page 29, OpenID authentication at an RP involves two phases: (a) OP endpoint URL discovery and retrieving meta-information, and (b) Verifying an authentication assertion URL received from an OP. Therefore, we divided the OpenID-Authentication-as-a-Service operation in OpenStack into two phases, each invoked with a separate API.

The two APIs which have been defined are:

- **OpenIDAuthReq:** This API is invoked in the initial phase by the front-end server. It executes an OpenID authentication request, and performs the first phase in the process.
- **OpenIDAuthVerify:** This API is invoked in the second phase by the front-end server. It executes the authentication verification for the OpenID authentication assertion URL received from the OP.

The message sequence for OpenID authentication is shown in figure 4.3. The entities in the mechanism are the User-Agent, the front-end GUI Server, the back-end OpenStack Server, and the OpenID provider. The functionality of the entities are described in table 4.1.

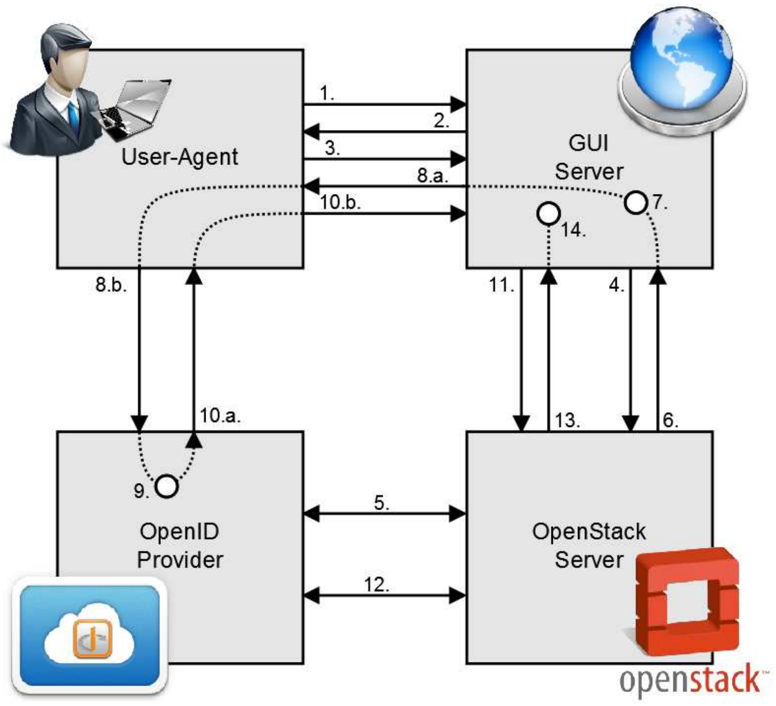


Figure 4.3: Message Sequence for OpenID Authentication API

Prior to authentication with OpenID, the administrator is required to add a specific OpenID User Identifier to an existing OpenStack user. The information is addded into the user database in OpenStack, which is used later in the process of authentication. The sequence of operations for the user to authenticate against OpenStack, using an OpenID URL are:

1 The user requests the index page from the GUI server with the HTTP User-Agent.

2 The GUI server returns the login page.

3 The user enters an OpenID Identifier (either user Identifier, or OP Identifier).

4 The GUI server invokes the *OpenIDAuthReq* API on the OpenStack API server to request an OpenID user authentication.

Table 4.1: Entities in OpenID Authentication for OpenStack

Entities	Description
User-Agent	An HTTP client application running at the user's end.
GUI Server	A front-end server, running GUI services for managerial operations on OpenStack.
OpenStack Server	The back-end OpenStack server, which is running the API services for OpenStack.
OpenID Provider	The OP, with which the user has already created an OpenID account.

5 The API functionality is executed at the OpenStack server. The process runs the Yadis discovery protocol, and performs the OP endpoint discovery, and retrieves all the other required meta-information from the OP to redirect the User-Agent to the OP.

6 The API server uses all the parameters for the redirection and generates the response XML for the *OpenIDAuthReq* API call.

7 The GUI server receives the XML response and parses the contents. It uses the values in the response to generate an `HTTP 302 Redirection` for the User-Agent.

8 (a) The User-Agent receives the `HTTP 302 Redirection` response from the GUI server. (b) The User-Agent is redirected to the OP endpoint URL.

9 The OpenID provider authenticates the user. Authentication mechanisms can vary from simple username/password, smart cards, client side certificates, device authentication (GBA), etc. Once the user is authenticated, the OP generates an authentication assertion response, with the required parameters and a signature (see section 3.2 on page 29).

10 (a) The OP sends back the User-Agent to the GUI server. (b) The GUI server receives the User-Agent and triggers the next phase of action from the front-end server.

11 The GUI server invokes *OpenIDAuthVerify* API, the second API in the authentication process, to request verification of the OpenID assertion URL.

12 There are multiple operations in this phase of the authentication process.

  a The API functionality is executed at the back-end OpenStack server.

  b The process verifies the signature in the assertion URL, performs discovery of the OP, and establishes a D-H shared key. It uses the key to securely verify all the parameters in the OpenID authentication assertion URL.

  c Once verified, **Auth Manager** retrieves the local OpenStack user, which has the specific OpenID user Identifier added to its credentials.

13 The API server uses the parameters for the successfully authenticated user and generates the response XML for the *OpenIDAuthVerify* API. If either the user authentication was not successful, or an OpenStack user with the asserted OpenID Identifier was not found, the API server returns an error code in the response XML.

14 The GUI server receives the XML response, and uses the user information to provide the login, and the HTML dashboard view to the user.

4.4 Usability of OpenID in OpenStack

A lot of discussions on the usability of OpenID has occurred [24, 29, 62, 56, 45]. In summary, the following advantages in authentication of users could be attained with OpenID in OpenStack:

- OpenID is the most widely used open standard for authentication. There are many OP providers for the user to choose from. These include Google [41], Yahoo [20], MyOpenID [10], and LiveJournal [9].

- It will provide a decentralized user centric authentication delegation for using OpenStack services. Users will have control over his or her own identity management and authentication.

- Usability will improve, as OpenID aims for a single user versus multiple service points applicability. Users using OpenID for other web services, can use the same OpenID to use OpenStack services.

- Users will have a seamless Single-Sign-On (SSO) experience. OPs allow users to remain signed in with the providers, and store cookies in the User-Agent to save login information. Thus, when a user is already signed in at the OP, trying to access the OpenStack Dashboard is a smooth and simple operation for the user, without requiring any extra interaction for re-authentication.

- Users will have flexibility in the authentication mechanism, as OPs allow different authentication mechanisms for their users. Most OpenID providers support username/password, and client certificate based authentication for users. Apart from that, Leicher *et al.* in [45] describe a trusted computing environment using OpenID, A. S. Ahmed in [24] presents a 3GPP standard authentication mechanism for smart phones, and Watanabe *et al.* in [62] illustrate a cellular subscriber ID and OpenID federated authentication architecture. Ericsson Labs also provides an Identity Management service, which uses GBA [23, 22, 35] based device authentication services with OpenID.

Chapter 5

Goals and Prototype Implementation

In this chapter, we present the description and the details of our prototype implementation. As mentioned in the research goals (see section 1.3 on page 3), the prototype is a proof-of-concept of the OpenID-Authentication-as-a-Service, as described in section 4.3 on page 37.

5.1 Research Methodology

This section includes the research work carried out to solve the initial problems. Apart from the details provided in the previous chapters, this section presents some of the specific implementation oriented investigations performed.

5.1.1 Dual PDP to Single PDP

As shown previously in figure 4.2 on page 37, the initial challenge was to devise a method to logically shift the front-end GUI server's role for conducting the OpenID authentication relaying to the back-end OpenStack server.

Therefore, the design for a dual-API service for providing OpenID authentication was made. With that, the two phases of OpenID authentication can be performed via API calls on the back-end.

5.1.2 OpenID Relay Point Complication

In traditional OpenID implementations, the RP performs all the operations for OpenID authentication from the same URL domain. This means, the URL domain the User-Agent requests, is the same domain which performs the redirection in the initial OpenID authentication request with the OP. Following authentication at the OP, the User-Agent is redirected to the same domain, and is the same point from where verification of the authentication assertion is performed. However, as we split the OpenID authentication process with the APIs, we have two domains: the front-end which interacts with the User-Agent, and the back-end which performs the rest of the operations.

Thus, we exploited the regular use of the OpenID parameters, by using the "`return_to`" and the "`trust_root`" parameters in a different way, but maintained the OpenID specifications [13, 54]. We used them to specify the front-end URL in the OpenID authentication setup by the back-end with the OP in the initial phase. As the User-Agent is only in contact with the front-end server, the OP recognises the service point using the `trust_root` information, and effectively redirects the User-Agent to the `return_to` URL at the front-end after authentication is performed at the OP.

5.1.3 Implementating a RESTful Service

The OpenStack **API Server** implements RESTful services. This means, all services are stateless, and all required information is sent with each request to the server. However, in the regular "stateful" OpenID authentication, the D-H shared key is stored after the first phase, and is retrieved during the verification session.

Therefore, we used the "stateless" OpenID authentication mechanism. This provided us with two benefits. First, it made the OpenID authentication possible to implement in a RESTful manner, without requiring store of any information for a session. Second, it provided a subtle protection against DoS attacks on the framework. As the shared key is set up only in the final phase, any extra processing in the initial phase after invoking the first API is avoided.

5.1.4 Architectural Consideration

Integrating OpenID functionality on the back-end server required some modifications in the architecture. Previously, the only module interacting with the public network was the **API Server**. However, with OpenID, it is necessary for OpenStack to have a point of interaction with the OP on the public network. Hence, we wanted this to be a separate module, detached from the core components, to ensure privacy of the internal network.

Therefore, we introduced a new module in the architecture of OpenStack, to implement the functionality of OpenID authentication. This module interacts with the OP in the public Internet on one interface, and with the **Cloud Controller** on another interface. In our prototype, the internal communication is done with XML over HTTP, thus allowing a modular deployment of OpenStack.

5.2 OpenID Authentication As A Service

The following sub-sections describes the architectural details of our prototype for OpenID authentication in OpenStack. All modules, added features, and extensions are discussed in the following sub-sections. It also includes the signalling sequence and the architectural action flow in the prototype.

5.2.1 Prototype Architecture

The basic architecture of OpenStack has been discussed previously in section 2.3 on page 8, and was shown in figure 2.2 on page 9.

In our prototype, we introduced some new modules to meet our requirements, but followed the structure of OpenStack. The modified OpenStack architecture is shown in figure 5.1.

The **Nova OpenID Controller** is the new module added to the architecture. Furthermore, we introduced an extension in the **API Server**, the **Cloud Controller**, **Auth Manager** and its **Nova-Manage** admin client. We also implemented invocation of the APIs from the **Django-Nova/Dashboard** framework to use the OpenID authentication functionality.

The OpenStack server runs the following services: *Nova-API*, *Nova-Objectstore*, *Nova-Compute*, *Nova-Network*, *Nova-Scheduler*, *Nova-Volume*, and *Nova-*

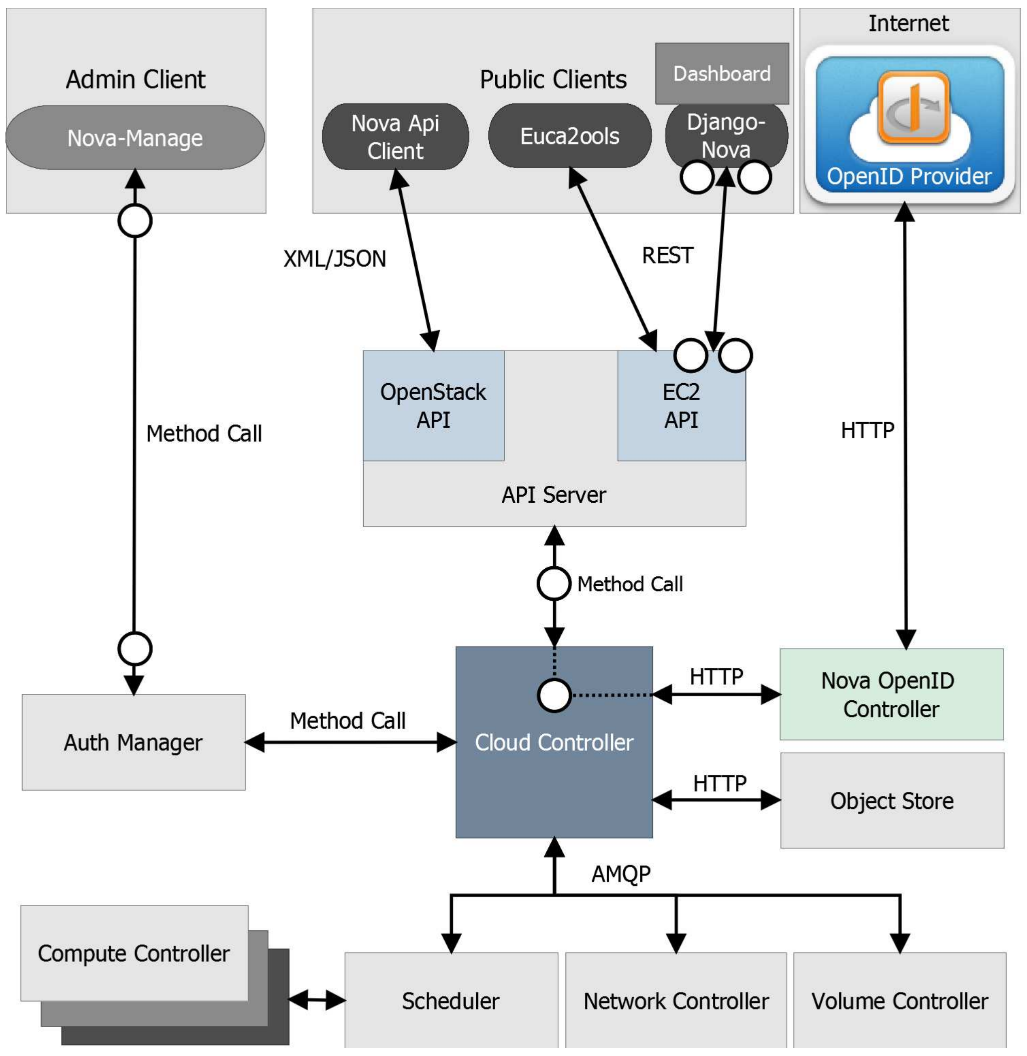


Figure 5.1: Additions to OpenStack Architecture

Ajax-Console-Proxy. In addition to these services, our prototype runs the *Nova-Openid-Controller* service on TCP port 9988 by default.

The circular points in figure 5.1 shows the points of the implemented extensions, along with the position of the **Nova OpenID Controller**. The following sections present further details of this implementation.

5.2.2 Modifications in *API Server*

Our design included two APIs to perform authentication with OpenID in OpenStack. OpenStack has two sets of APIs, the EC2APIs, and the OSAPIs (see section 2.3.3 on page 14).

However, we chose to implement our prototype as an addition in the set of EC2APIs. This was because the specifications of EC2APIs are more well documented and supported by Amazon AWS. Furthermore, the functions supported by EC2APIs are more comprehensive compared to OSAPIs. The Django-Nova/Dashboard framework also uses EC2APIs to provide the graphical interface to the users. Hence, we implemented the OpenID authentication mechanism with EC2APIs, and used the Django-Nova/Dashboard framework to realize the new API services.

We designed two APIs to perform OpenID-Authentication-as-a-Service from the front-end server. The following present the design of the two APIs.

- **OpenidAuthReq**
  - *Action Description:* The front-end server initially invokes this API to request an OpenID authentication from the back-end.
  - *Parameters:* All mandatory parameters for an EC2API call (see section 2.3.3.2 on page 16). Additional parameters include the user supplied OpenID Identifier, and the "`openid.return_to`" URL for the front-end server. The additional parameters are set as values in the "`Name`" parameter in the EC2API request, separated by an "`&`".
    E.g. `Name=profile.google.com/rasib&http://localhost:8000/openid-auth-return`
  - *Internal Execution:* The **API Server** receives the request, parses the parameters, and executes the API functionality with a method call to the **Cloud Controller**, and waits for the response variables.
  - *Response Handler:* When the response variables are received from the **Cloud Controller**, it generates the response XML for the API call. The `<OpenidAuthReqResponse>` tag holds the parameters in the response XML object.
- **OpenidAuthVerify**
  - *Action Description:* The front-end server invokes this API to request an OpenID authentication verification from the back-end,

after the User-Agent has been redirected to the front-end with the assertion URL from the OP.

- *Parameters:* All mandatory parameters for an EC2API call (see section 2.3.3.2 on page 16). Additional parameter includes the whole assertion URL sent from the OP to the `return_to` URL at the front-end server. The additional parameter is set as value in the `"Name"` parameter in the EC2API request. Format of the assertion URL in OpenID authentication is shown in section 3.2 on page 29.
- *Internal Execution:* Similar to the previous API, the **API Server** receives the request, parses the parameters, and executes the API functionality with a method call to the **Cloud Controller**, and waits for the response variables.
- *Response Handler:* When the response variables are received from the **Cloud Controller**, it generates the response XML for the API call. The `<OpenidAuthVerifyResponse>` tag holds the parameters in the response XML object.

5.2.3 Modifications in *Cloud Controller*

The **Cloud Controller** incorporates function handlers for each API in the **API Server**. Once the **API Server** receives a request, it invokes the specific handler in **Cloud Controller**. Thus, we extended the function of **Cloud Controller** by including two additional handlers for the two above mentioned APIs in the **API Server**.

The corresponding handlers for the two APIs are `openid_auth_req()`, and `openid_auth_verify()` respectively. Both of these handlers in **Cloud Controller** internally communicates with the `OpenIDHandler` sub-module, which specifically interacts with the **Nova-OpenID Controller**.

The `OpenIDHandler` communicates with **Nova-OpenID Controller** with GET requests over HTTP. In the present implementation, it is assumed that this interface is within a secured domain. Thus, no additional security is imposed in the messages. However, to ensure maximum security, it is suggested that HTTPS is used if the deployment does not have a secured internal network.

5.2.4 Addition of *Nova-OpenID Controller*

As shown in figure 5.1 on page 46, the **Nova-OpenID Controller** module is added to the OpenStack architecture. It accepts GET requests over HTTP from the **Cloud Controller**.

The **Nova-OpenID Controller** uses two sub-domain URLs for the **Nova-Openid-Controller** to receive the *OpenidAuthReq* and the *OpenidAuthVerify* action requests, implemented in a RESTful way. The description of the processes are as follows:

- *OpenidAuthReq* ⇒ `openid_auth_req()`
  - *Service URL:* http://localhost:9988/request/
  - *Required parameters:* `openid_identifier` and `return_to` URL.
  - *Action:* Runs Yadis discovery protocol [19], and discovers the OP endpoint URL and its specifications in the meta-information for the `openid_identifier`. It then uses the `return_to` URL and the received information to generate the redirectional parameters.
  - *Response Parameters:* According to the specifications of OpenID authentication [54, 13], the following variables are returned to Cloud Controller to be used for the redirection: `op_endpoint_url`, `openid.return_to`, `openid.realm`, `openid.ns`, `openid.mode`, `openid.claimed_id`, and `openid.identity`.
- *OpenidAuthVerify* ⇒ `openid_auth_verify()`
  - *Service URL:* http://localhost:9988/verify/
  - *Required parameters:* The assertion URL received from the OP at the `return_to` URL on the front-end server.
  - *Action:* Receives the assertion URL, and verifies the signature. Upon successful verification, it parses the `return_to` URL from the assertion URL, and performs a stateless `check_authentication` operation for the `return_to` URL with the `op_endpoint_url`. It sets up a D-H shared key, and uses the key to securely verify the following parameters in the assertion URL: `op_endpoint`, `claimed_id`, `identity`, `return_to`, and `response_nonce`.
  - *Response Parameters:* Once successfully verified, it returns a success message to Cloud Controller, and the `openid.claimed_id`.

Failure in any state of the verification process sends back an error status message.

A trace from performing OpenID authentication with the APIs on OpenStack is included in appendix A.2 on page 79.

5.2.5 Modifications in *Auth Manager*

Any web server providing OpenID authentication has an internal mechanism to link a local user to an OpenID Identifier. Thus, we included an additional credential for OpenStack users for OpenID authentication in OpenStack. In addition to the previously mentioned parameters (see section 2.3.2 on page 10), the user database in OpenStack includes the "*openid*" credential. The OpenStack administrator is expected to add an OpenID Identifier for a user to enable OpenID authentication for the user.

The **Nova-OpenID Controller** sends the `openid.claimed_id` to **Cloud Controller** upon successful verification in the second phase of the authentication process. The **Auth Manager** then uses this OpenID URL to retrieve the OpenStack user from the database, for that specific authenticated `openid.claimed_id`. This is similar to existing use cases for other APIs, where the **Auth Manager** uses access keys, secret keys, or usernames to retrieve the user information from the database.

5.2.6 Modifications in *Nova-Manage*

As explained in section 2.3.2.3 on page 13, the **Nova-Manage** module is an admin tool for **Auth Manager**. Thus, we included an additional function in **Nova-Manage**, using which the administrator can add and modify the OpenID information for the user.

The command for adding an OpenID URL for an OpenStack user is:

```
$ nova-manage user openid [user_openid_url]
```

Thus, the administrator adds the OpenID Identifier to enable OpenID authentication for the user during user creation. However, the current implementation allows one-to-one mapping of OpenStack users to OpenIDs, and a specific user can have only one OpenID Identifier.

5.2.7 Configuration Presets

The prototype requires the following presets for executing OpenID authentication in OpenStack.

1. The Dashboard/Django-Nova GUI server is running.
2. Django-Nova has an "admin" account with OpenStack. The OpenID authentication is requested as a service on behalf of this account from the front-end.
3. Dashboard/Django-Nova has a pre-specified "`return_to`" URL set by the administrator.
4. OpenStack has the *Nova-API* and *Nova-OpenID-Controller* services running.
5. By default, the `API Server` runs on TCP port 8773, and the `Nova-OpenID Controller` runs on TCP port 9988.
6. The administrator has created a user on OpenStack, and included the user's OpenID Identifier in the credentials.

Specific instructions and commands for configuring the presets is included in appendix A.1 on page 78.

5.2.8 Message Sequence

The signalling sequence in our prototype is shown in figure 5.2. When the User-Agent requests the index page from the front-end server, the login page contains the "*openid-identifier*" field for entering the OpenID URL for the user. Thus, the user enters his OpenID URL (User Identifier or OP Identifier) and submits the value to the server.

GUI server then calls the *OpenIDAuthReq* EC2API on the OpenStack API server. Additionally, it sets the value of the parameter "`Name`" as "`openid_url&return_to_url`".

The API Server receives the API request, and Nova-OpenID Controller runs the Yadis discovery protocol for the `openid_url`, to retrieve the OP endpoint URL, and meta-information for the OP. The API server then generates the response XML for the *OpenIDAuthReq* API call and sends it to the front-end server to redirect the user.

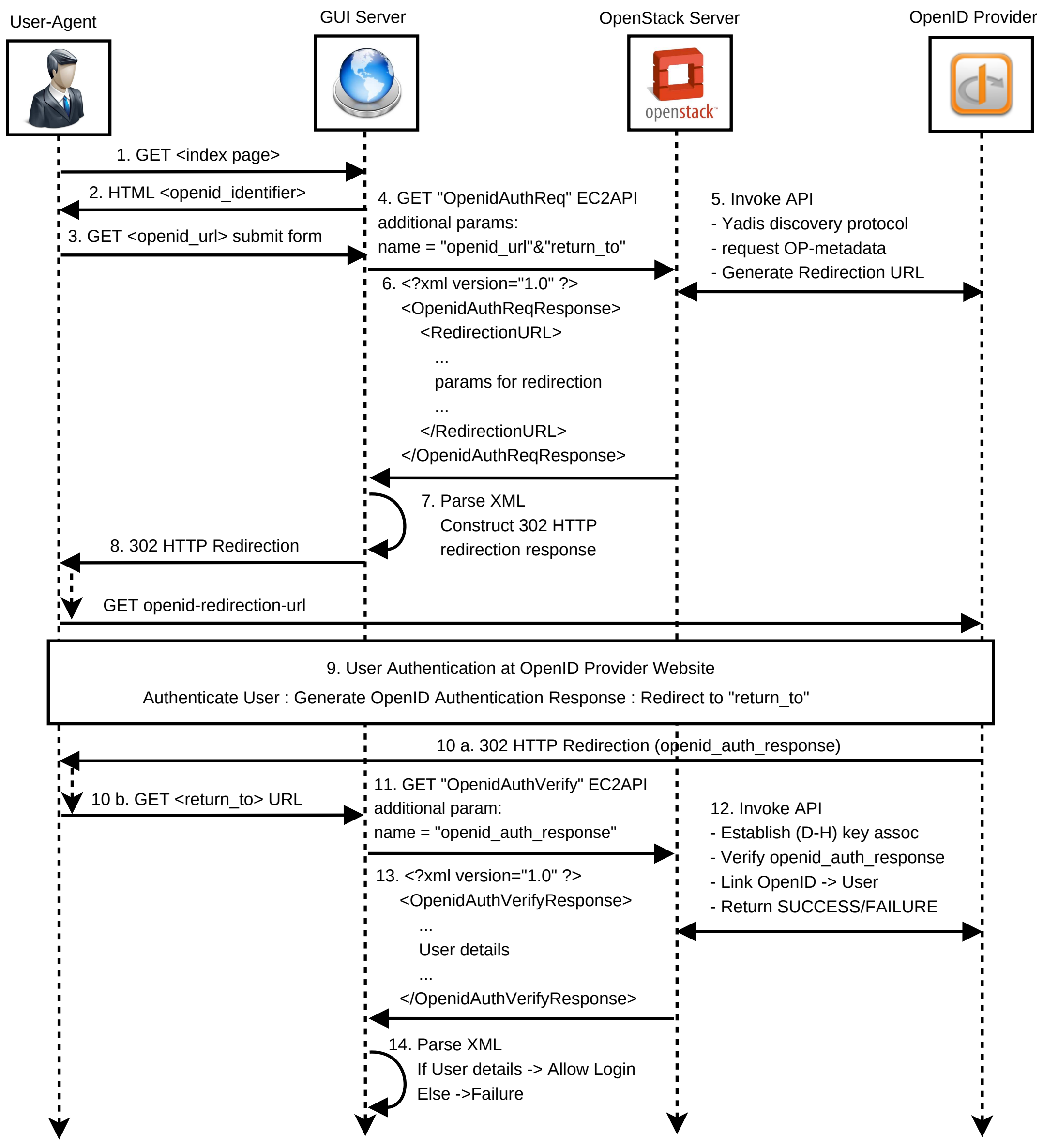

User-Agent
GUI Server
OpenStack Server
OpenID Provider
openstack
1. GET <index page>
2. HTML <openid_identifier>
3. GET <openid_url> submit form
4. GET "OpenidAuthReq" EC2API
additional params:
name = "openid_url"&"return_to"
5. Invoke API
- Yadis discovery protocol
- request OP-metadata
- Generate Redirection URL
6. <?xml version="1.0" ?>
<OpenidAuthReqResponse>
<RedirectionURL>
...
params for redirection
...
</RedirectionURL>
</OpenidAuthReqResponse>
7. Parse XML
Construct 302 HTTP
redirection response
8. 302 HTTP Redirection
GET openid-redirection-url
9. User Authentication at OpenID Provider Website
Authenticate User : Generate OpenID Authentication Response : Redirect to "return_to"
10 a. 302 HTTP Redirection (openid_auth_response)
10 b. GET <return_to> URL
11. GET "OpenidAuthVerify" EC2API
additional param:
name = "openid_auth_response"
12. Invoke API
- Establish (D-H) key assoc
- Verify openid_auth_response
- Link OpenID -> User
- Return SUCCESS/FAILURE
13. <?xml version="1.0" ?>
<OpenidAuthVerifyResponse>
...
User details
...
</OpenidAuthVerifyResponse>
14. Parse XML
If User details -> Allow Login
Else ->Failure

Figure 5.2: Signalling Sequence for OpenID Authentication in OpenStack

The GUI server uses the values, and sends an HTTP 302 Redirection to the User-Agent. The User-Agent is then redirected to the OP. The OP authenticates the user, and sends back the User-Agent to the `return_to` URL at the GUI server with the assertion URL.

At the `return_to` URL, the GUI server invokes *OpenIDAuthVerify* EC2API.

Additionally, it sets the value of the parameter `"Name"` as the whole OpenID assertion URL received from the OP.

The API Server receives the request, and Nova-OpenID Controller runs the verification process with the OP. It runs the Yadis discovery protocol to discover the OP services, and establishes a D-H shared key to verify the assertion URL. Upon verification, OpenStack then links a local user with the `openid.claimed_id`.

The API server then creates the response XML for the *OpenIDAuthVerify* API call with the user information, which the front-end server uses to allow the user to log in. If the authentication was unsuccessful, it returns an error message in the response XML.

5.2.9 Prototype Action Flow

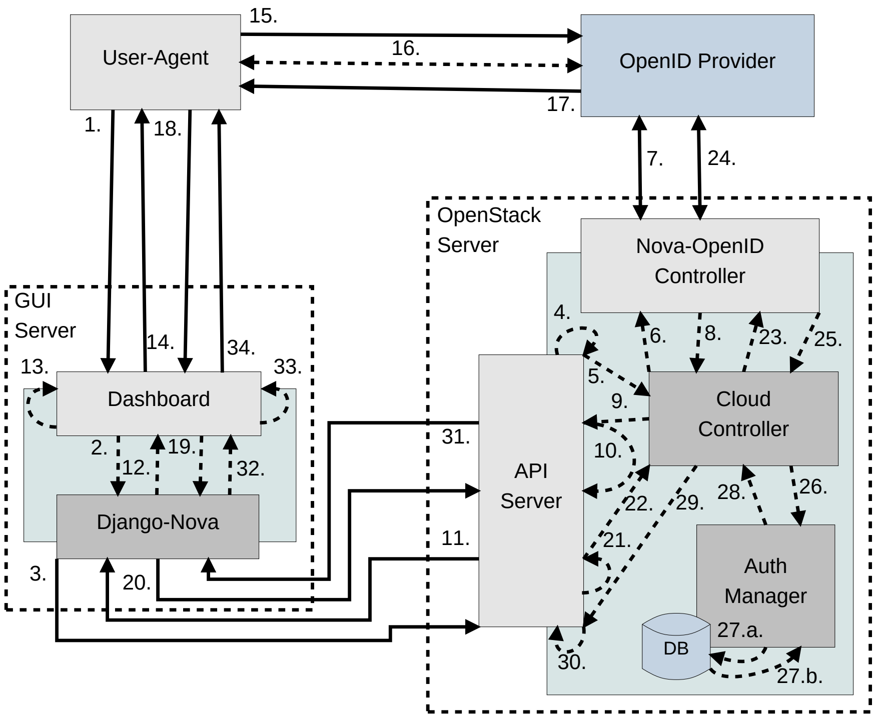


Figure 5.3: Action Flow for OpenID Authentication in OpenStack

A detail description of the internal communication of the modules in the architecture and the action flow for the prototype is shown in figure 5.3. The sequence of actions are as follows:

1: From the index page, the User-Agent submits the OpenID Identifier URL for authentication.

2: Dashboard receives the OpenID URL and invokes an internal method call to the Django-Nova framework.

3: Django-Nova invokes the "OpenidAuthReq" API to the back-end OpenStack API server.

4, 5: API Server parses the parameters and invokes the `openid_auth_req()` handler in Cloud Controller.

6: Cloud Controller sends a function request to the `http://localhost:9988/request` URL in Nova-OpenID Controller.

7: Nova-OpenID Controller runs the Yadis discovery protocol to retrieve the OP end point URL, and OP's other meta-information and specifications.

8: Cloud Controller receives the parameters required for redirecting the User-Agent to the OP.

9, 10, 11: API Server receives the parameters, forms the response XML object, and sends it as response to the API call from Django-Nova.

12: Django-Nova parses the XML object, and sends the values to Dashboard.

13: Dasboard uses the values to populate a form template, and uses the HTTP POST method to submit the values for redirection of the User-Agent.

14, 15: User-Agent receives the redirection form, and is redirected to the OP with the redirectional parameters.

16: The user authenticates himself to the OP using whichever authentication mechanism available at the OP.

17, 18: User-Agent is redirected back to the `openid.return_to` URL at the front-end GUI server.

19: The `return_to` URL on Dashboard receives the assertion URL from the OP, and invokes an internal method call to the Django-Nova framework for verification of the assertion URL.

20: Django-Nova invokes the "OpenidAuthVerify" API to the back-end OpenStack API server.

21, 22: API Server parses the parameters and invokes the `openid_auth_verify()` handler in Cloud Controller.

23: Cloud Controller sends a function request to the `http://localhost:9988/verify` URL in Nova-OpenID Controller.

24: Nova-OpenID Controller verifies the signature, sets up a D-H shared key, and securely verifies values of all the parameter names which are included in `openid.signed`.

25: If successfully verified, Nova-OpenID Controller returns a success message, and the `openid.claimed_id` to the Cloud Controller. If verification was unsuccessful, it returns an error message.

26: If successfully verified, Cloud Controller requests Auth Manager to retrieve the user information from the user database. If verification was unsuccessful, Cloud Controller forms a "user not found!" error message and sends it back to the API server.

27, 28: Auth Manager interacts with the database to retrieve the user credentials, and sends it to Cloud Controller.

29, 30, 31: API Server receives the parameters (from step 26 in case of unsuccessful verification), forms the response XML object, and sends it as response to the API call from Django-Nova.

32: Django-Nova parses the XML object, and sends the values to Dashboard.

33: Dasboard receives the values. If authentication was successful, the response values include the user information. If authentication was unsuccessful, the response is a "user not found!" error message.

34: Thus, based on the received response values, Dashboard either allows the user to log in to the interface, or displays a login error message.

The action flow completes the sequence of using the OpenID-Authentication-as-a-Service APIs on OpenStack from the front-end GUI server. Thus, the PDP and PAP is shifted to the back-end, and the front-end only acts as a "dumb" PEP.

5.3 OpenID Authentication APIs

This section gives the description of the API invocation and the possible responses for the call. In addition to the OpenID parameters, both the APIs require the mandatory parameters for all EC2APIs (see section 2.3.3.2 on page 16).

5.3.1 OpenidAuthReq API

This section describes the format of *OpenidAuthReq* API call, and the expected response XML objects.

5.3.1.1 API Invoke

As mentioned earlier in the configuration presets (section 5.2.7 on page 51), the authentication service API is a service required to be invoked by an admin user on OpenStack owned Django-Nova. Thus, the format for calling the *OpenidAuthReq* EC2API is shown in table 5.1.

Table 5.1: Format for OpenidAuthReq EC2API

```
GET http://localhost:8773/services/Admin/
?AWSAccessKeyId=<admin_access_key>
&Action=OpenidAuthReq
&Name=<user_supplied_openid_url>&<gui_server_return_to_url>
&SignatureMethod=HmacSHA256
&SignatureVersion=2
&Timestamp=<timestamp>
&Version=nova
&Signature=<signature_for_the_request>
```

5.3.1.2 Response Format

The response for *OpenidAuthReq* EC2API maintains the standard response format for EC2APIs. If a valid OpenID Identifier is supplied by the user, **API Server** responds with the variables for the redirection. Table 5.2 shows a successful response format.

Table 5.2: Success Response Format for OpenidAuthReq EC2API

```
<?xml version="1.0" ?>
<OpenidAuthReqResponse xmlns="http://ec2.amazonaws.com/doc/nova/">
  <requestId>ec2api_request_id</requestId>
  <input>
    <openidClaimedId>openid_claimed_id</openidClaimedId>
    <openidReturnTo>return_to_url?janrain_nonce=XXXX</openidReturnTo>
    <openidNs>openid_specification_version</openidNs>
    <openidIdentity>user_supplied_openid</openidIdentity>
    <openidMode>checkid_setup</openidMode>
    <openidRealm>return_to_url</openidRealm>
  </input>
  <form>
    <action>op_endpoint_url</action>
    <acceptCharset>UTF-8</acceptCharset>
    <id>openid_message</id>
    <enctype>application/x-www-form-urlencoded</enctype>
    <method>post</method>
  </form>
</OpenidAuthReqResponse>
```

If the user supplied an invalid OpenID URL, it responds with an error message, as shown in table 5.3.

Table 5.3: Failure Response Format for OpenidAuthReq EC2API

```
<?xml version="1.0"?>
<Response>
  <Errors>
    <Error>
      <Code>NotFound</Code>
      <Message>Invalid OpenID Provider</Message>
    </Error>
  </Errors>
<RequestID>ec2api_request_id</RequestID>
</Response>
```

5.3.2 OpenidAuthVerify API

This section describes the format of *OpenidAuthVerify* API call, and the expected response XML objects.

5.3.2.1 API Invoke

Once the assertion URL is received at the front-end GUI server, it calls the *OpenidAuthVerify* EC2API. The format for invoking the API is shown in table 5.4.

Table 5.4: Format for OpenidAuthVerify EC2API

```
GET http://localhost:8773/services/Admin/
?AWSAccessKeyId=<admin_access_key>
&Action=OpenidAuthVerify
&Name=<assertion_url_from_op>
&SignatureMethod=HmacSHA256
&SignatureVersion=2
&Timestamp=<timestamp>
&Version=nova
&Signature=<signature for the request>
```

5.3.2.2 Response Format

The response for he *OpenidAuthVerify* EC2API depends on the result of the verification. If successful, the response XML object contains the user information, which is similar to the response for the existing *DescribeUser* EC2API [1]. The format for a successful response is shown in table 5.5.

If the verification was unsuccessful, the **API Server** responds with an error message, as shown in table 5.6.

5.4 Dashboard/Django-Nova with OpenID

The Dashboard/Django-Nova framework uses the EC2APIs to provide the GUI service. For generating API calls, Django-Nova uses the *python-boto* library [52]. Thus we used *python-boto* to implement the *OpenidAuthReq* and

Table 5.5: Success Response Format for OpenidAuthVerify EC2API

```
<?xml version="1.0" ?>
<OpenidAuthVerifyResponse xmlns="http://ec2.amazonaws.com/doc/nova/">
  <requestId>ec2api_request_id</requestId>
  <username>authenticated_user</username>
  <secretkey>user_secret_key</secretkey>
  <accesskey>user_access_key</accesskey>
  <file>user_credential_file_url</file>
  <openid>user_openid</openid>
</OpenidAuthVerifyResponse>
```

Table 5.6: Failure Response Format for OpenidAuthVerify EC2API

```
<?xml version="1.0"?>
<Response>
  <Errors>
    <Error>
      <Code>NotFound</Code>
      <Message>
         No user for OpenID:claimed_openid
      </Message>
    </Error>
  </Errors>
<RequestID>ec2api_request_id</RequestID>
</Response>
```

OpenidAuthVerify API calls to OpenStack API Server, to perform OpenID-Authentication-as-a-Service from the front-end.

In current implementation, Django-Nova owns an "admin" user account in OpenStack server. All initial requests from the server are made with the "admin" user's credentials. Once a user logs in successfully, and Django-Nova receives the user information, it then makes all further requests with the user's credentials. Thus, the OpenID authentication service is only usable by an Admin user.

At first, Dashboard/Django-Nova requests the OpenID authentication service. Section 5.3.1 on page 56 describes how to invoke the *OpenidAuthReq* API, and the expected response XML.

Once Django-Nova receives the response, it parses the values in the XML object, and passes them on to Dashboard as *name:value* pairs. Dashboard then uses the values to populate a form template for the `HTTP 302 Redirection`. The structure of the redirection form, along with a small auto-submission script we used, is included in the appendix (see table A.1 in section A.2 on page 79).

After the user has successfully authenticated himself at the OP, the User-Agent is redirected to the `return_to` URL on the Dashboard server. The URL executes a method call to Django-Nova, to invoke the *OpenidAuthReq* API. Section 5.3.2 on page 58 describes how to invoke the *OpenidAuthVerify* API, and the expected response XML. A successful authentication returns the user information, which Dashboard uses to populate the managerial interface for the authenticated user.

Chapter 6

Analysis and Discussion

This chapter presents a post-implementation analysis and discussion of the prototype and the security concerns. The goal of the thesis was a proof-of-concept for the integration of OpenID in OpenStack. Thus, this chapter presents an evaluation and analysis on the architectural ideas rather than thorough performance measurements.

6.1 Critical Security Points

Any web service is prone to security attacks. In theory, the level of security of a system is equal to the security of the minimum secured point in the architecture.

In our implementation, the exposed interfaces are the **GUI Server** on the front-end with the **User-Agent**, the **API Server** on the back-end with the **GUI Server**, the **OP** end point with the **User-Agent** on the public network, and the **Nova-OpenID Controller** back-end with the **OP**.

In all cases where the User-Agent is involved, using HTTPS is the safest solution. Standard web services provide server-side certificates to authenticate the server to the User-Agent. However, in this case, it is crucial to authenticate the client to the web server. Thus, using both client-side and server-side certificates for mutual authentication is recommended. Additionally, other mechanisms to protect the integrity of the messages relayed through the User-Agent include nonce checking, and verifying `openid.sig` for all the parameters included in `openid.signed`. It should be noted that using an HTTPS connection increases the latency of the authentication procedure.

For the API server, all RESTful requests include signatures with a pre-shared secret between the GUI server and OpenStack. Thus, unless the front-end server is vulnerable to a compromise by an attacker, the connection to the back-end can be considered as an integrity protected channel. Currently, the communication between Django-Nova and the API Server is over HTTP. Hence, using SSL between the front-end Django-Nova and the API Server is the most reasonable solution for providing confidentiality. This is a standard practice for all RESTful services. The API Server is an HTTP-supported public interface, usable with other API clients (such as euca-tools [4], and python-nova client [15]). Security technologies such as IPSec [33] are intended for network level host-to-host security, rather than application-to-application security. Therefore, IPSec is not a recommended security solution for the RESTful API Server. However, HTTPS support in the OpenStack API Server has not been implemented yet, and remains as a future task.

The verification of the assertion URL by Nova-OpenID Controller and the OP occurs in the back-end. This is significantly different from regular OpenID implementations, where the front-end is always the RP. However, the back-end communication of Nova-OpenID Controller with the OP cannot be considered hidden from an attacker. An attacker can sniff packets from the network to intercept the communication between Nova-OpenID Controller and the OP. Hence, following OpenID specifications, this communication is implemented over an encrypted channel with the D-H shared key between Nova-OpenID Controller and the OP, as the OP is assumed to be on the public Internet.

An important part of the implementation on Nova-OpenID Controller was the generation of redirection parameters. As the "return_to" URL is specified by the front-end, we ensured its integrity in the verification process by making sure it is included in the "`openid.signed`" parameter list, and the parameter values are all included in the "`openid.sig`" signature.

However, there is still scope for an attacker to manipulate the information. As explained in section 3.3 on page 32, the solution can be vulnerable to Discovery Tampering, Adversary Relay Proxy, and DoS attacks.

Session management between the User-Agent and Dashboard/Django-Nova front-end is another area where the security should be improved. Authentication of the user is done in the back-end. However, OpenStack still has to rely on the front-end to maintain the user session. Services on OpenStack are RESTful services, and no session information is stored, while the front-end is a session-based service point for the User-Agent. It is contradictory with the design principles of RESTful services to maintain such session based security. Therefore, OpenStack trusts the front-end Dashboard/Django-Nova to manage the user session.

6.2 Use Case Study

The implementation of OpenID on OpenStack was tested against two specific use cases. The following sections present the details for the use cases.

6.2.1 Standard OpenID Providers

Our implementation follows all the specifications of OpenID 2.0 [54, 13]. Thus, authentication in OpenStack using OpenID is supported for all standard OpenID providers supporting OpenID 2.0.

We have verified use cases for authentication using our own implemented OP, and also with OpenIDs from *Google*, *Yahoo*, and *MyOpenID*. Table 6.1 summarizes the features of the OpenID providers. In each case, we were concerned with the returned value for `openid.claimed_id`, which is the user Identifier the OpenStack administrator is required to include in the user credentials.

Table 6.1: Standard OpenID Providers

OP	Comments
Google	OP Identifier `https://www.google.com/accounts/o8/id` returns random string as Identifier. User Identifier `https://profiles.google.com/username` returns user specified Identifier. Supports OpenID 2.0.
Yahoo	`https://me.yahoo.com/username` returns user specified Identifier. Supports OpenID 2.0.
MyOpenID	`http://username.myopenid.com/` returns user specified Identifier. Insecure HTTP OP endpoint. Supports OpenID 2.0.

6.2.2 GBA Provider

3rd Generation Partnership Project (3GPP) specifies the Generic Bootstrapping Architecture (GBA) [23] as a mechanism to authenticate devices. It uses cellular technology with the Internet to provide authentication services to mobile devices. As an extension, the 3GPP also specifies a standardization for the integration of OpenID with GBA [22, 24].

Ericsson Labs provide a GBA based authentication service in their Identity Management (IDM) portal [35]. Currently, it can be used over the Internet, making any device behave as a next generation mobile device, by installing the *SoftSim* application on the device, available on the portal.

In our use case, we used an Android enabled mobile device for accessing the OpenStack GUI. We used the GBA with OpenID mechanism to authenticate the Android device with the *SoftSim* application. Ericsson IDM uses the OP Identifier `https://idm.labs.ericsson.net` to initiate the OpenID authentication. A successful authentication returns a user specified `https://idm.labs.ericsson.net/portal/id/username` Identifier from the portal.

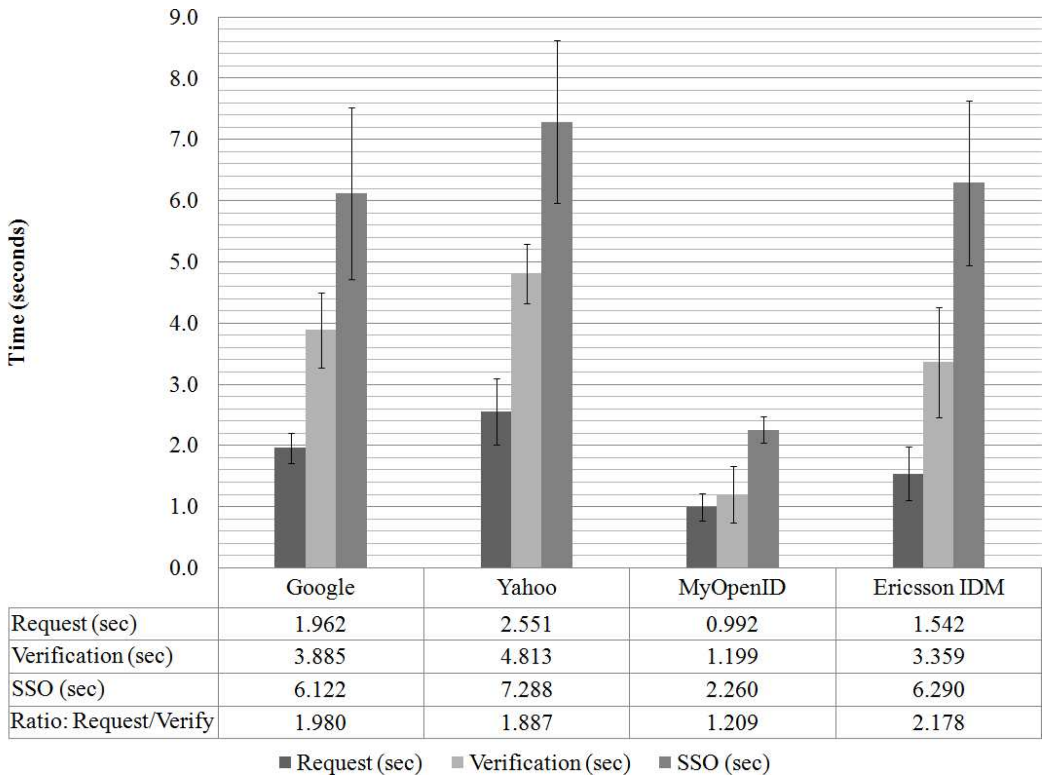
6.2.3 Performance Evaluation

In our use cases, we found that the execution time varied with the different OpenID providers. We were running the OpenStack server back-end and the Dashboard/Django-Nova front-end GUI server on the same machine. The server was running the *Ubuntu 10.04.2 LTS Lucid* 64-bit operating system, on a Tower MacPro4.1, with 8GB RAM, and a 2.27GHz Intel(R) Xeon(R) 16 Core 64-bit processor.

We captured network packets and calculated the time differences to evaluate the performance of our prototype. We recorded 30 observations for each OpenID provider. The times were calculated based on two phases. The "Request" phase includes the time from the User-Agent submitting the OpenID Identifier until the User-Agent reaches the OP endpoint URL. The "Verification" phase includes the time from the authenticated User-Agent being redirected from the OP, until the time when the user is logged into the Dashboard interface.

Furthermore, we recorded another 30 measurements for each provider, when the user is already signed in at the OP. The user had a seamless SSO experience, without the need to re-authenticate at the OP. The timing includes the time for the user to submit their OpenID Identifier on the Dashboard interface, and directly log in without any additional user interaction.

The measurements had an approximately Gaussian distribution. Thus, we calculated the mean of the readings. The recorded measurements are shown in figure 6.1. The graph shows the mean of each set of readings, along with the population standard deviation for each OP.



	Google	Yahoo	MyOpenID	Ericsson IDM
Request (sec)	1.962	2.551	0.992	1.542
Verification (sec)	3.885	4.813	1.199	3.359
SSO (sec)	6.122	7.288	2.260	6.290
Ratio: Request/Verify	1.980	1.887	1.209	2.178

Figure 6.1: Time Measurements for OpenID Authentication

As shown in the graph, the request phase for all the OPs has small standard deviation compared to the verification phase and the SSO timings. The request phase only requires the Nova-OpenID Controller to discover the OP meta-information, and thus exhibits a relatively consistent behaviour.

The table shows the ratio of the verification phase to the request phase for each provider. It can be seen that, except for *MyOpenID*, all providers require approximately double the time in the verification phase compared to the request phase. The verification phase includes setting up a D-H shared key, and encryption and decryption of all information while verifying the assertion URL. Thus, Nova-OpenID Controller requires more time in the second phase. A higher standard deviation in the readings is understandable because of the varying processing times both at the Nova-OpenID Controller and at the OP end.

The time measurements for the providers show that *MyOpenID* takes the least time in all cases. This is because *MyOpenID* uses only a basic HTTP connection and is thus faster, **but unsecured**.

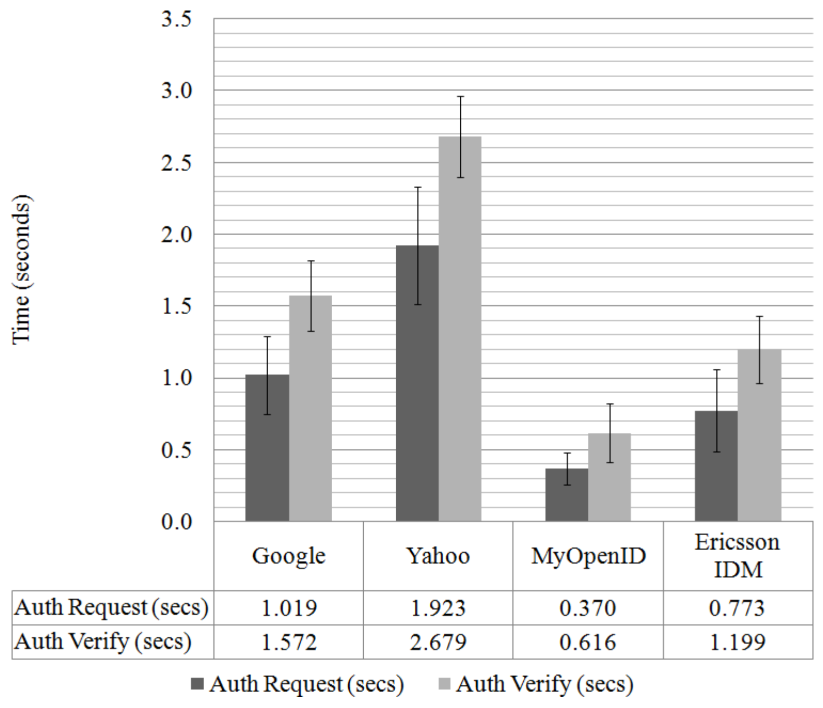
For all OPs, the SSO timing is greater than the summation of the authentication and the verification phase timings. The first two measurements did not include the user interaction while performing authentication at the OP. However, in the measurements for SSO, the User-Agent requires time for the extra processing needed to authenticate itself to the OP with the cookies stored in the device. The SSO timings for all OPs display the highest standard deviation. This is because the timing includes processing delays at the User-Agent (to retrieve the cookies), at the OP, and at Nova-OpenID Controller. As because these three entities have varying performance, the recorded timing intervals had a comparatively high variance.

Furthermore, we measured the internal timing between the Nova-OpenID Controller and Cloud Controller to evaluate the performance of the internal module. We recorded the duration of time for the Cloud Controller to send a request to Nova-OpenID Controller and receive a response, for both the *openid_auth_req* and *openid_auth_verify* API handlers. We recorded 30 measurements for each of the OPs for each operation. The recorded measurements are shown in figure 6.2. The graph shows the average duration of time between the request and the response for the authentication request and the authentication verification along with the standard deviation for each of the OPs.

The graph in figure 6.2 for the internal timing measurements shows a similar pattern to the external measurements in figure 6.1. The standard deviations for all the OPs were also consistent.

However, figure 6.2 does not show the timing for the verification phase to be twice the time required for the request phase as in figure 6.1. Additionally, the sum of the internal timings on figure 6.2 is much lower compared to the external timing in figure 6.1 for all the OPs. This is because the external measurements include the time required for the API Server to process the variables and generate the response XML for the API call, the time required for Dashboard/Django-Nova front-end to process the data and generate the HTML view and, primarily, the time required for the authentication process at the OP end point.

However, the performance of the prototype and the timing measurements depend largely on the hardware configuration of the server. Additionally, OpenStack does not incorporate any efficiency improvement mechanisms at present, and the design and architecture of the whole system is still evolving.



	Google	Yahoo	MyOpenID	Ericsson IDM
Auth Request (secs)	1.019	1.923	0.370	0.773
Auth Verify (secs)	1.572	2.679	0.616	1.199

Figure 6.2: Time Measurements for Nova-OpenID Controller

6.3 Version Information

OpenStack released its initial version, Austin, in October 2010. The second release version, Bexar, came out in February 2011.

Our implementation was integrated with the Bexar release. After a successful implementation with Bexar, we then integrated our solution with Cactus, the third release, which came out in April 2011.

Diablo, the fourth release of OpenStack is scheduled to be released in September 2011. However, beginning with the Diablo release, the authentication framework design is supposed to utilize a new architecture. It will integrate the *KeyStone* project [49] as the authentication module, the design of which is still evolving.

6.4 Discussion

The goal of this thesis project was to introduce a flexible decentralized authentication mechanism for cloud computing platforms. Studying the usability and availability of services, we chose OpenID as the authentication mechanism. After extensive research on open-source cloud middleware solutions, we chose OpenStack to continue our work. However, OpenID has its own flaws and vulnerabilities, and OpenStack also has its limitation and complexities.

After achieving our initial goal, we conducted further research, and converged on the idea of implementing OpenID-Authentication-as-a-Service APIs in OpenStack. In addition to fulfilling the requirement to perform authentication with OpenID, we introduced the concept of performing the authentication in a rather unusual manner. We divided the authentication process into two phases, and implemented the processes with two separate APIs on the API Server.

During the thesis work, we faced certain technical challenges. Section 5.1 on page 43 discusses the way each of the issues were handled. The prototype incorporated all of the standard security practices and meets the specifications of the technologies being used.

One of the most challenging parts of the work was to logically adapt the dual PDP scenario into a single PDP. Even with a working design, the implementation required extensive exploration of the OpenID parameters to divide the functionality of a standard RP in to the necessary components. Verification of the parameters and inclusion of the parameters in the signature was one of the crucial features of the implementation.

Another difficult design task was choosing the point of integration of the features in the OpenStack architecture. We believe that we have selected an interesting approach, by adding the Nova-OpenID Controller as a separate module and implementing an internal interface with the Cloud Controller. This was especially important from the perspective of internal network security. As this module interacted with the OP on the public Internet, the separation provides the required abstraction of the internal network from the public Internet.

One of the features of the implementation is that it only supports OpenID version 2.0. This was a requirement as we needed to implement a RESTful service, and a similar "stateless" mode is only supported in OpenID 2.0. This cannot be viewed as a limitation, as OpenID 2.0 is considered to be more secure than its previous versions.

Chapter 7

Conclusions and Future Work

The evolution of cloud computing is driving the next generation of internet services. In addition to proprietary platforms, multiple open-source middleware solutions are available on the Internet.

Currently, all middleware support platform specific technologies for authentication and access control. In our research we chose OpenStack as our open-source cloud platform. OpenStack allows access via its two set of RESTful APIs: the EC2APIs and the OSAPIs. These APIs use access keys and secret keys to authenticate users in the framework when accessing services.

Nonetheless, the adoption of these new technologies must be easy for the users. To improve usability, OpenStack offers a graphical interface with the Dashboard/Django-Nova framework. However, the current architecture of OpenStack lacks specific GUI based services. Thus, the Dashboard/Django-Nova framework uses the set of EC2APIs to provide HTML views of the GUI. For initial authentication, the front-end incorporates a separate username and password based validation mechanism. This introduces a dual PDP scenario, which is not a recommended practice for web services.

In this thesis, we introduced a flexible decentralized authentication service for the front-end. We selected OpenID as an open-source authentication platform. OpenID has its own advantages for web services, which include improvements in usability and seamless SSO experience for the users. OpenID allows users to choose identities from multiple providers on the Internet, use a range of authentication mechanisms depending upon the mechanisms supported by the provider, and to access multiple service points on the Internet with the same OpenID Identifier.

In our design, OpenID authentication in the front-end is used as a service from the back-end OpenStack server. As a result, we were able to shift the dual points of decision making and perform the authentication at a single PDP in the back-end. For our implementation, we explored OpenID authentication. We were able to divide the mechanism into two phases: the authentication request and the authentication verification phase. The two phases in the authentication process were then implemented with two separate APIs.

The design was implemented on OpenStack. We used Dashboard/Django-Nova as the front-end GUI, and implemented the APIs as EC2APIs. The two implemented APIs on the OpenStack API server are **OpenidAuthReq** and **OpenidAuthVerify**. Additionally, we added the **Nova-OpenID Controller** a new module in the OpenStack architecture. This module communicates with the **Cloud Controller** over an internal HTTP interface, and with the OP on the public Internet over another interface. Furthermore, we extended the functionality of the **Nova-Manage** administrator tool, to add and modify the OpenID credentials for existing OpenStack users.

The OpenID support for the prototype is designed only for use with OpenID version 2.0. The prototype implementation was successfully tested against standard OpenID providers on the Internet supporting OpenID 2.0. The use cases were evaluated against OpenID Identifiers from *Google*, *Yahoo*, and *MyOpenID*, along with a GBA based authentication mechanism from *Ericsson IDM Services*.

The **Nova-OpenID Controller** on the back-end incorporates secure authentication and verification of the OpenID assertion URL with the OP. Therefore, secure interaction with the OP over HTTPS is supported. However, Nova-OpenID Controller is able perform the authentication process over an insecure HTTP connection, depending upon the OP's capabilities. In our timing measurements, we found that using HTTPS requires more time compared to using HTTP. This is a basic trade-off between security and performance of any system. Execution time traces of performing OpenID authentication in OpenStack is included in appendix A on page 78.

In our thesis, we developed a prototype as a the proof-of-concept for using OpenID-Authentication-as-a-Service from front-end GUI servers. The security of the solution follows the relevant standards. This concept can be applied to similar architectures, where there is a separation of the PDP on the service back-end, and a PEP on the "dumb" front-end.

Future Work

The research performed during this thesis project revealed further possibilities. The first objective would be to introduce greater flexibility in the choice of authentication mechanisms for the user. Second, one should introduce open platforms for authorization delegation in OpenStack.

To provide flexibility in the choice of authentication on OpenStack, we suggest that other authentication platforms, such as Shibboleth/SAML, be considered. Shibboleth is an organization centric authentication mechanism. The authentication tokens and other information are exchanged between the authentication provider and the service provider on the back-end.

A use case for Shibboleth authentication is where an organization has a service agreement with an OpenStack cloud provider. Thus, any user from the organization should be able to use the OpenStack services using his or her organization centric identity using Shibboleth authentication.

To provide a generic solution for authentication, we aim to design a common Authentication-as-a-Service API in OpenStack. This API will allow the front-end GUI to use any method of authentication based on the user's choice.

The second objective would be to introduce open authorization platforms in OpenStack. We will consider OAuth [11] as the most appropriate open-source authorization mechanism. OAuth is a user centric authorization delegation specification, and allows two parties to securely interchange specific information about authorized resources.

We have already performed an applicability analysis of OAuth on OpenStack. A possible use case for OAuth is when a user has accounts on two OpenStack providers. With OAuth, the user will be able to define authorized resources on provider A. Then, the user can utilize resources from provider B, with provisions for dynamic scaling of resources to provider A. A high level design of delegated authorization with OAuth in OpenStack is included in appendix B on page 84.

Bibliography

[1] Amazon AWS EC2 API Reference, http://docs.amazonwebservices.com/awsec2/latest/apireference/, last accessed 30th April 2011.

[2] Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/s3/, last accessed 30th April 2011.

[3] Amazon Web Services (AWS), http://aws.amazon.com, last accessed 30th April 2011.

[4] Euca-Tools, Eucalyptus Community, http://open.eucalyptus.com/wiki-/toolsecosystem, last accessed 10th May 2011.

[5] Eucalyptus Open Source Cloud Platform, http://www.eucalyptus.com, last accessed 5th March 2011.

[6] Google App Engine, https://code.google.com/appengine, last accessed 15th March 2011.

[7] Hadoop, http://hadoop.apache.org/, last accessed 10th March 2011.

[8] Libvirt Online Working Group, The Virtualization API, http://libvirt.org, last accessed 25th April 2011.

[9] LiveJournal OpenID Services, http://www.livejournal.com, last accessed 1st February 2011.

[10] MyOpenID OpenID Services, https://www.myopenid.com, last accessed 14th June 2011.

[11] OAuth Community Site, http://oauth.net/, last accessed 15th june 2011.

[12] OpenID Attribute Exchange 1.0, http://openid.net/specs/openid-attribute-exchange-1_0.html, last accessed 30th May 2011.

[13] OpenID Foundation, http://openid.net, last accessed 14th June 2011.

[14] OpenNebula, http://opennebula.org, last accessed 15th March 2011.

[15] Python-Nova Client, http://pypi.python.org/pypi/python-novaclient/2.3, last accessed 10th may 2011.

[16] Python-Openid 2.2.5, http://pypi.python.org/pypi/python-openid, last accessed 5th May 2011.

[17] Rackspace US, http://www.rackspace.com, last accessed 3rd March 2011.

[18] SalesForce CRM & Cloud Computing, www.salesforce.com, last accessed 3rd March 2011.

[19] Yadis 1.0, The Identity and Accountability Foundation for Web 2.0, http://yadis.org, last accessed 5th June 2011.

[20] Yahoo OpenID Services, http://openid.yahoo.com, last accessed 14th June 2011.

[21] Twenty-one experts define cloud computing. In *SYS-CON Media, Inc., http://virtualization.sys-con.com/node/612375.* January 2009.

[22] 3rd Generation Partnership Project. 3GPP TR 33.924. Identity management and 3gpp security interworking; identity management and generic authentication architecture (gaa) interworking, (release 9). http://www.3gpp.org/ftp/specs/html-info/33924.htm, 2009.

[23] 3rd Generation Partnership Project. 3GPP TS 33.220. Technical specification group services and system aspects; generic authentication architecture (gaa); generic bootstrapping architecture (release 8), http://www.3gpp.org/ftp/specs/html-info/33220.htm, 2009.

[24] Ahmed, A. S. A user friendly and secure openid solution for smart phone platforms. Master's thesis, Faculty of Information and Natural Sciences, School of Science and Technology, Aalto University, June 2010.

[25] Almulla, S., and Yeun, C. Y. Cloud computing security management. In *Engineering Systems Management and Its Applications (ICESMA), 2010 Second International Conference on* (30 2010-april 1 2010), pp. 1 –7.

[26] Berners-Lee, T., Fielding, R., and Masinter, L. Uniform Resource Identifier (URI): Generic Syntax. *Internet Request for Comments RFC 3986 (Standard)* (Jan. 2005).

[27] Cloud.com. Cloudstack, http://cloud.com, last accessed 2nd March 2011.

[28] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON). *Internet Request for Comments RFC 4627 (Informational)* (July 2006).

[29] Dhamija, R., and Dusseault, L. The seven flaws of identity management: Usability and security challenges. *Security Privacy, IEEE 6*, 2 (march-april 2008), 24 –29.

[30] Di-Lucca, G., Fasolino, A., Mastoianni, M., and Tramontana, P. Identifying cross site scripting vulnerabilities in web applications. In *Web Site Evolution, 2004. WSE 2004. Proceedings. Sixth IEEE International Workshop on* (sept. 2004), pp. 71 – 80.

[31] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. *Internet Request for Comments RFC 2246 (Proposed Standard)* (Jan. 1999). Obsoleted by RFC 4346, updated by RFCs 3546, 5746.

[32] Diffie, W., and Hellman, M. New directions in cryptography. In *IEEE Transactions on Information Theory* (nov 1976), vol. 22, pp. 644 – 654.

[33] Doraswamy, N., and Harkins, D. *IPSEC: The New Security Standard for the Internet, Intranets, and Virtual Private Networks.* Prentice Hall, 1999.

[34] Erdos, M., and Cantor, S. Shibboleth architecture protocols and profiles, http://shibboleth.internet2.edu/shibboleth-documents.html.

[35] Ericsson Labs Identity Management Framework. https://labs.ericsson.com/developer-community/blog/identity-management-framework-now-available-download, last accessed 15th May 2011.

[36] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. *Internet Request for Comments RFC 2068 (Proposed Standard)* (Jan. 1997). Obsoleted by RFC 2616.

[37] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[38] Foster, I. and Yong Zhao and Raicu, I. and Lu, S. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08* (November 2008), pp. 1 – 10.

[39] Garber, L. Denial-of-Service Attacks Rip the Internet. *Computer Volume 33*, Number 4 (April 2000), 12–17, doi: 10.1109/MC.2000.839316.

[40] Gollmann, D. Computer security. *Wiley Interdisciplinary Reviews: Computational Statistics 2*, 5 (2010), 544–554.

[41] Google. Google OpenID Services, http://code.google.com/apis/accounts/docs/openid.html, last accessed 15th March 2011.

[42] Han, H., Kim, S., Jung, H., Yeom, H., Yoon, C., Park, J., and Lee, Y. A restful approach to the management of cloud infrastructure. In *IEEE International Conference on Cloud Computing* (2009), pp. 139–142.

[43] Kumaran, S., Liu, R., Dhoolia, P., Heath, T., Nandi, P., and Pinel, F. A restful architecture for service-oriented business process execution. In *Proc. IEEE Int. Conf. e-Business Engineering ICEBE '08* (2008), pp. 197–204.

[44] Lee, H. K., Malkin, T., and Nahum, E. Cryptographic strength of ssl/tls servers: current and recent practices. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2007), IMC '07, ACM, pp. 83–92.

[45] Leicher, A., Schmidt, A., Shah, Y., and Inhyok, C. Trusted computing enhanced openid. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for* (nov. 2010), pp. 1 –8.

[46] Lindholm, A. Security Evaluation of the OpenID Protocol. Master's thesis, School of Computer Science and Communication, Royal Institute of Technology (KTH), 2009.

[47] MICROSOFT. MSDN CardSpace Token Acquisition Protocol Specification (MS-CTAP), http://msdn.microsoft.com/en-us/library/cc717408(v=prot.10).aspx, last accessed 25th may 2011.

[48] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2009), CCGRID '09, IEEE Computer Society, pp. 124–131.

[49] OPENSTACK KEYSTONE PROJECT. http://wiki.openstack.org/openstack-authn, last accessed 25th May, 2011.

[50] OPPLIGER, R. Microsoft .net passport and identity management. *Information Security Technical Report 9*, 1 (2004), 26 – 34.

[51] PHILPOTT, R., MALER, E., RAGOUZIS, N., HUGHES, J., MADSEN, P., AND SCAVO, T. OASIS Open 2008, Security Assertion Markup Language (SAML) V2.0 Technical Overview, Committee Draft 02, http://docs.oasis-open.org/security/saml/post2.0/sstc-saml-tech-overview-2.0.html.

[52] PYTHON BOTO. http://code.google.com/p/boto/, last accessed 15th may, 2011.

[53] RACKSPACE US, INC. Openstack compute developer guide api 1.0, api v1.0, january 2011.

[54] RECORDON, D., AND REED, D. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management* (New York, NY, USA, 2006), DIM '06, ACM, pp. 11–16.

[55] RICHARDSON, L., AND RUBY, S. *RESTful Web Services.* O'Reilly Media, May 2007.

[56] RIEGER, S. User-centric identity management in heterogeneous federations. In *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on* (may 2009), pp. 527 –532.

[57] ROSENBERG, J., AND REMY, D. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption.* Pearson Higher Education, 2004.

[58] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. Role-based access control models. *Computer 29*, 2 (feb 1996), 38 –47.

[59] Sovis, P., Kohlar, F., and Schwenk, J. Security Analysis of OpenID. In *Sicherheit'10* (2010), vol. 170, F. C. Freiling, pp. 329–340.

[60] van Delft, B., and Oostdijk, M. A security analysis of openid. In *Policies and Research in Identity Management*, E. de Leeuw, S. Fischer-Hübner, and L. Fritsch, Eds., vol. 343 of *IFIP Advances in Information and Communication Technology*. Springer Boston, 2010, pp. 73–84.

[61] Wachob, G., Reed, D., Chasen, L., Tan, W., and Churchill, S. Extensible resource identifier (xri) resolution v2.0, http://docs.oasis-open.org/xri/2.0/specs/xri-resolution-v2.0.pdf.

[62] Watanabe, R., and Tanaka, T. Federated authentication mechanism using cellular phone - collaboration with openid. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on* (april 2009), pp. 435 –442.

[63] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP), http://www.w3.org/tr/soap, last accessed 14th April 2011.

Appendix A

Executing OpenID in OpenStack

This section includes some execution instructions and traces from the execution of the OpenID authentication mechanism on OpenStack.

A.1 Setup Information

This section provides the setup instructions for the OpenStack prototype to function.

The first requirement is to set up the database.

```
$ InstallationDir/bin/nova-manage db sync
```

An admin user is created on OpenStack for the Dashboard/Django-Nova framework, with the access key "admin", and the secret key "admin". Another regular user, "rasib" is then created, who will be using the service. The access and secret key is automatically generated by OpenStack in this case.

```
$ InstallationDir/bin/nova-manage user admin dashboardAdmin admin admin
$ InstallationDir/bin/nova-manage user admin rasib
```

Having user "rasib" created, we would then add the user's OpenID Identifier with his OpenStack credentials. In this case, we would be using his Google OpenID information, which would be used to log into OpenStack GUI.

```
$ InstallationDir/bin/nova-manage user openid rasib https://profiles.
```

```
google.com/rasib
```

A project is created on OpenStack, with the Dashboard administrator as the project manager.

```
$ InstallationDir/bin/nova-manage project create project1 dashboardAdmin
```

Finally, a small network for the pool of IP addresses for the VMs is created.

```
$ InstallationDir/bin/nova-manage network create 172.16.0.0/16 1 16
```

Having done all of it, it is then required to run the OpenStack services with the following commands.

```
$ InstallationDir/bin/nova-api
$ InstallationDir/bin/nova-objectstore
$ InstallationDir/bin/nova-compute
$ InstallationDir/bin/nova-network
$ InstallationDir/bin/nova-scheduler
$ InstallationDir/bin/nova-volume
$ InstallationDir/bin/nova-ajax-console-proxy
$ InstallationDir/bin/nova-openid
```

Finally, after setting up OpenStack services, we would then run the Dashboard/Django-Nova GUI service.

```
$ DashboardDir/tools/with_venv.sh DashboardDir/dashboard/manage.py runserver
0.0.0.0:8080
```

A.2 Execution Trace

We had run the services on the Nomadic Lab network, on host *n25*. Thus, initially, the User-Agent requests the index page from Dashboard from:

```
http://n25.nomadiclab.com:8080
```

The user then types his OpenID URL in the openid-identifier box, and submits the value. Thus, the *OpenidAuthReq* API is called, with the appropriate parameters. The API request looks like the following:

```
GET http://n25.nomadiclab.com:8773/services/Admin/
?AWSAccessKeyId=admin
&Action=OpenidAuthReq
&Name=profiles.google.com/rasib\
      &http://n25.nomadiclab.com:8080/openid/verify/
&SignatureMethod=HmacSHA256
&SignatureVersion=2
&Timestamp=2011-03-23T07:20:55
&Version=nova
&Signature=FnNUPULRvynIjYG6ylVWK9PrjWj3NCmWrf0dgzNY8s=
```

As this API is called, the following messages on the Nova-OpenID Controller service describes its functions:

```
OpenID identifier:  profiles.google.com/rasib
Service redirection URL: http://n25.nomadiclab.com:8080/openid/verify/
Trust root included:  http://n25.nomadiclab.com:8080/openid/verify/
Generated checkid_setup request to https://www.google.com/accounts/
o8/ud?source=profiles using stateless mode
```

With the discovered information by Nova-OpenID Controller, it returns the redirectional parameters in the response XML for the front-end to redirect the User-Agent to *Google* for user authentication.

```
<?xml version="1.0" ?>
<OpenidAuthReqResponse xmlns="http://ec2.amazonaws.com/doc/nova/">
  <requestId>M84ZV2V9T9ZLUZRG5VEP</requestId>
  <input>
    <openidClaimedId>https://profiles.google.com/rasib85</openidClaimedId>
    <openidReturnTo>http://10.0.0.25:8080/openid/verify/
        ?janrain_nonce=2011-03-23T07%3A20%3A56ZLnMsYN
    </openidReturnTo>
    <openidNs>http://specs.openid.net/auth/2.0</openidNs>
    <openidIdentity>https://profiles.google.com/rasib85</openidIdentity>
    <openidMode>checkid_setup</openidMode>
    <openidRealm>http://10.0.0.25:8080/openid/verify/</openidRealm>
  </input>
  <form>
    <action>https://www.google.com/accounts/o8/ud?source=profiles</action>
    <acceptCharset>UTF-8</acceptCharset>
    <id>openid_message</id>
    <enctype>application/x-www-form-urlencoded</enctype>
    <method>post</method>
  </form>
</OpenidAuthReqResponse>
```

Using the response XML, Dashboard then uses a form to populate the fields and an auto-submission script to POST to redirect the user. The format of the form is shown in table A.1.

Table A.1: Dashboard HTTP 302 Redirection Form

```
<body onload="document.forms[0].submit();">
  <h1>OpenID transaction in progress</h1>
  <form id="{{formId}}" action="{{formAction}}"
       method="{{formMethod}}"
       accept-charset="{{formAcceptCharset}}"
       enctype="{{formEnctype}}">
    <input type="hidden" name="openid.return_to"
           value="{{inputOpenidReturnTo}}"/>
    <input type="hidden" name="openid.realm"
           value="{{inputOpenidRealm}}"/>
    <input type="hidden" name="openid.ns"
           value="{{inputOpenidNs}}"/>
    <input type="hidden" name="openid.claimed_id"
           value="{{inputOpenidClaimedId}}"/>
    <input type="hidden" name="openid.mode"
           value="{{inputOpenidMode}}"/>
    <input type="hidden" name="openid.identity"
           value="{{inputOpenidIdentity}}"/>
    <input type="submit" value="Continue"/>
  </form>
  <script>
    var elements = document.forms[0].elements;
    for (var i = 0; i < elements.length; i++) {
      elements[i].style.display = "none";
    }
  </script>
</body>
```

The User-Agent is redirected to the OP, and once the user authenticates himself, he is sent back to the `return_to` URL on Dashboard/Django-Nova. Dashboard/Django-Nova then calls the *OpenidAuthVerify* API.

```
GET http://n25.nomadiclab.com:8773/services/Admin/
?AWSAccessKeyId=admin
&Action=OpenidAuthVerify
&Name=/openid/verify/
    ?janrain_nonce=2011-03-23T07:20:56ZLnMsYN
    &openid.ns=http://specs.openid.net/auth/2.0
    &openid.mode=id_res
    &openid.op_endpoint=https://www.google.com/
                        accounts/o8/ud?source=profiles
    &openid.response_nonce=2011-03-23T07:20:58ZQ02wNXOYIlqBcQ
    &openid.return_to=http://10.0.0.25:8080/openid/verify/
                      ?janrain_nonce=2011-03-23T07:20:56ZLnMsYN
    &openid.assoc_handle=AOQobUd67rDCBApRSNOX2lBiEA_
                         jmOuOmONcNzI4im8sDgz6KGo1iZ1E
    &openid.signed=op_endpoint,
                   claimed_id,
                   identity,
                   return_to,
                   response_nonce,
                   assoc_handle
    &openid.sig=9m2AfSwsGQ/40frxzNJlB4KqRbk=
    &openid.identity=https://profiles.google.com/rasib
    &openid.claimed_id=https://profiles.google.com/rasib
&SignatureMethod=HmacSHA256
&SignatureVersion=2
&Timestamp=2011-03-23T07:20:58
&Version=nova
&Signature=8jnIm7Ede/KbNYuAcKu73Yx9aT MxHyLNijjwhAoSBE=
```

As this API is called, the following messages on the Nova-OpenID Controller service describes its functions:

```
No pre-discovered information supplied.
Performing discovery on https://profiles.google.com/rasib
Received id_res response from https://www.google.com/
accounts/o8/ud?source=profiles using association
AOQobUd67rDCBApRSNOX2lBiEA_jmOuOmONcNzI4im8sDgz6KGo1iZ1E
Using OpenID check_authentication
op_endpoint
claimed_id
identity
return_to
response_nonce
assoc_handle
Status:  success
```

```
Identifier: https://profiles.google.com/rasib
```

After the verification is completed successfully, Auth Manager retrives the user information for OpenID "*https://profiles.google.com/rasib*". The API server then responds with the user information for Dashboard/Django-Nova to allow the user to log in into the interface.

```
<?xml version="1.0" ?>
<OpenidAuthVerifyResponse xmlns="http://ec2.amazonaws.com/doc/nova/">
  <requestId>C5J4N394A9-TI8Y5BD4O</requestId>
  <username>rasib</username>
  <secretkey>ad68e65d-5599-47ee-b191-8803c57f24b3</secretkey>
  <accesskey>a7caceb8-3b25-435a-a836-d4ce632f450f</accesskey>
  <file/>
  <openid>https://profiles.google.com/rasib</openid>
</OpenidAuthVerifyResponse>
```

Appendix B

Applicability of OAuth in OpenStack

OAuth [11] is an open-source platform for authorization delegation between web services. OAuth enables a user-centric sharing and access of resources between two service points.

We performed an initial analysis on the applicability of OAuth on OpenStack. The following sections present a high level implementation design for a possible use case of OAuth in OpenStack.

B.1 OAuth Overview

OAuth performs allows authorization delegation on behalf on the user, between to service points. OAuth specifications [11] provides the following sequence of operations to implement authorization with OAuth:

1. User-X is at ServerA, and wants to access a remote resource from ServerB.
2. ServerA sends sends a request to ServerB for a "`request_token`" to the *request_token_URL* in ServerB.
3. ServerB responds with a "`request_token`" to ServerA.
4. ServerA redirects User-X with the "`request_token`" to the *authorization_URL* in ServerB to authorize the remote resource.

5. User-X authenticates and authorizes the resource for ServerA at ServerB.

6. ServerB redirects User-X back to ServerA with the authorized `"request_token"`.

7. ServerA uses the authorized `"request_token"` to send a request to the *access_token_URL* in ServerB for an `"access_token"`.

8. ServerB responds with an `"access_token"`.

9. ServerA then uses the `"access_token"` to send requests to the *resource_URL* in ServerB to access the remote resource which the user has authorized.

B.2 Use Case

An overview of a use case for OAuth in OpenStack is shown in figure B.1.

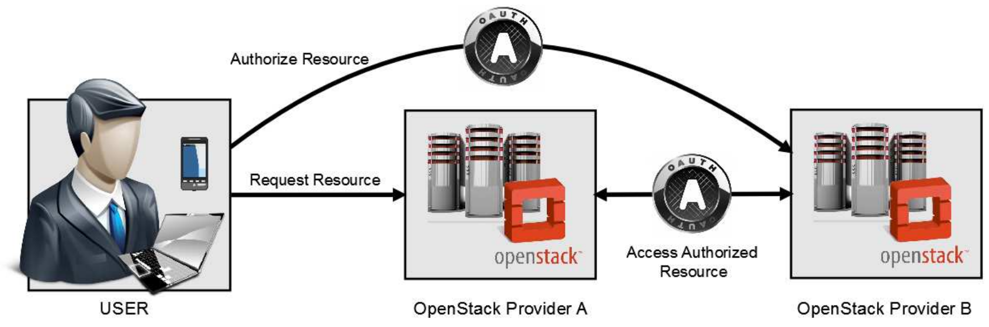


Figure B.1: OAuth Use Case Overview

User-X is assumed to own two accounts with two separate OpenStack providers. OpenStack providers A and B are assumed to pre-share a service level agreement. There can be two use cases in this scenario.

B.2.1 Use Case 1

User-X Requests Resource from OpenStack Provider B through OpenStack-Provider A

User-X is at OpenStack provider A. However, the user wishes to request a resource from OpenStack provider B, through provider A.

Thus, the remote resource request is implemented using OAuth between OpenStack providers A and B. Provider A redirects User-X to provider B. At provider B, the user authorizes some resources for providers A. User-X is then again redirected to provider A, and delegates the OAuth authorization token.

Provider A then accesses the authorized resource on provider B with the OAuth token.

B.2.2 Use Case 2

User-X Pre-Authorizes Resources on OpenStack Provider B for OpenStack-Provider A

User-X is at OpenStack provider A. The user then configures a cloud scaling feature on provider A, such that provider A can dynamically scale the resources from User-X's account on OpenStack provider A to User-X's account on OpenStack provider B.

During configuration, provider A redirects User-X to provider B. The user then presets authorized resources for provider A for a certain duration. User-X is then redirected back to provider A with an OAuth authorization token for the specified duration.

Provider A saves the OAuth token for User-X, and can use the token to dynamically scale up the resources to provider B, till when the token expires.

B.3 Signalling Sequence Overview

In our future work, we aim to implement OAuth in OpenStack for delegated authentication. We have designed the initial signalling sequence of OAuth in OpenStack, shown in figure B.2, to implement the use cases described in section B.2 on page 85.

The user initially submits the remote OpenStack Provider's URL to access a remote resource. The front-end server on Provider-A thus calls the "*RemoteResource*" API in the back-end OpenStack server.

Provider-A then calls the "*DescribeURL*" API in Provider-B, to discover the `authorization_URL` in Provider-B. Provider-B thus responds with the

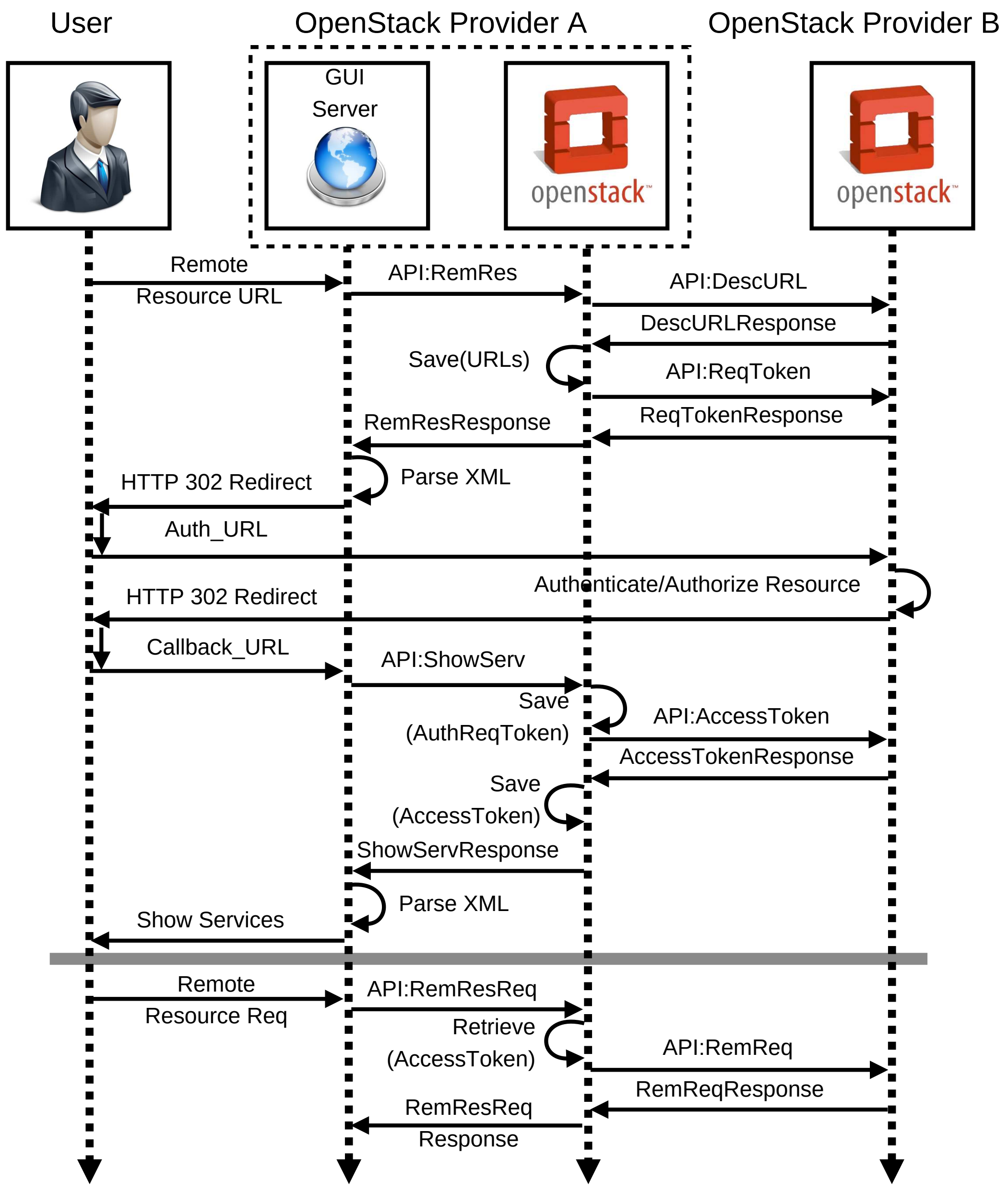


Figure B.2: OAuth in OpenStack Signalling Sequence

`authorization_URL` in the XML object, which Provider-A saves, and starts the second phase of operation.

Provider-A then calls the "`RequestToken`" API in Provider-B. Provider-B thus responds with a "`request_token`" to Provider-A. Provider-A's back-end then sends the "*RemoteResourceResponse*" XML, with the token, and the required redirectional parameters to the front-end GUI server. The front-end thus parses the XML object, and redirects the User-Agent to the `authorization_URL` in Provider-B.

The user then authenticates and authorizes the resource on Provider-B. After that, Provider-B redirects the User-Agent back to the front-end GUI server of Provider-A at the pre-configured `callback_URL`, which was included in the "`request_token`".

The `callback_URL` at the front-end GUI calls the "*ShowServices*" API in Provider-A, along with the authorized "`request_token`", to display the list of services authorized from Provider-B. Upon receiving the request, the Provider-A back-end saves the token, and calls the "*AccessToken*" API in Provider-B with the authorized "`request_token`". Provider-B receives the authorized "`request_token`", and responds with an "`access_token`". Provider-A then receives the "`access_token`", saves it, and sends an XML response to the front-end GUI to display the authorized services on Provider-B.

After this, the user can request a remote resource on the front-end GUI. The front-end in turn calls the "*RemoteResourceRequest*" API in Provider-A's back-end. The back-end receives the request, and retrieves the "`access_token`" to call the "*RemoteRequest*" API in Provider-B. Provider-B thus receives the request with the valid "`access_token`", allocates the resource, and sends back a success message to Provider-A.

B.4 Summary

Implementation of OAuth in OpenStack will introduce flexibility in the authorization of resources on the platform. It could be used and extended in various ways. This section provides an overview of OAuth, along with two applicable use cases for delegated authorization. Furthermore, we also presented the initial design for OAuth on OpenStack.

TRITA-ICT-EX-2011:158