Journal of Information Assurance and Security.
ISSN 1554-1010 Volume 7 (2012) pp. 111-118
© MIR Labs, www.mirlabs.net/jias/index.html

Service Oriented Integration of OpenID Authentication in OpenStack

Rasib H. Khan[1], **Abu Shohel Ahmed**[2], **and Jukka Ylitalo**[2]

[1]Helsinki Institute for Information Technology (HIIT)
Department of Computer Science and Engineering, Aalto University
Espoo, Finland
rasib.khan@aalto.fi

[2]Ericsson Research, Ericsson
Jorvas, Finland
ahmed.shohel@ericsson.com, jukka.ylitalo@ericsson.com

Abstract: **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 paper, we aim to introduce a cloud platform independent, user-centric, and decentralized authentication mechanism, using OpenID as an open-source authentication mechanism in OpenStack. OpenID allows a decentralized framework for user authentication. It has its own advantages for web services, which include improvements in usability and seamless Single-Sign-On experience for the users. This paper 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 (PDP). The work includes the architecture and implementation of the APIs in two generations of the OpenStack architecture. Our implementation allows users to use their OpenID Identifiers from standard OpenID providers, and log into the Dashboard/Django-Nova graphical interface of OpenStack. Further, we analysed the our design and implementation using OpenID providers on the Internet and measuring the performance against each.**

Keywords: Authentication, EC2API, OpenID, OpenStack, OS-API, Security

I. Introduction

Ian Foster *et al.* in [**?**] have defined Cloud Computing 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 new paradigm for utilization of scalable resources over the internet, 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. The layers of cloud computing services in SOA can be described as: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS).

IaaS providers, such as Amazon AWS[1], provide virtual CPUs, storage facilities, memory, etc. according to user requests. PaaS acts as an abstraction between the physical resources and the service. PaaS providers, such as Google App Engine[2], supply a software platform and the application programming interfaces (APIs), where users execute their software components. SaaS provider, such as Salesforce.com[3], provide end users with integrated services from the providers, comprising of hardware, development platforms, and applications.

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[4], Salesforce, etc. have driven development of multiple open-source cloud platforms. The most prominent among the open source cloud projects are OpenStack[5], CloudStack[6] , Eucalytus[7], and OpenNebula[8]. The open-source cloud platforms provide the ability to deploy private IaaS clouds. Many open-source cloud platforms have compatible application programming interfaces (APIs) with public clouds such as Amazon AWS EC2APIs [**?**], which improves the flexibility and usability of the private clouds.

However, the cloud solutions available today have little flexibility in their authentication system. All of the above mentioned platforms allow user authentication, based on

[1]Amazon Web Services (AWS), http://aws.amazon.com

[2]Google App Engine, https://code.google.com/appengine

[3]SalesForce CRM & Cloud Computing, www.salesforce.com

[4]Rackspace US, http://www.rackspace.com

[5]OpenStack, http://www.openstack.org/

[6]CloudStack, http://cloud.com

[7]Eucalyptus, http://www.eucalyptus.com

[8]OpenNebula, http://opennebula.org

proprietary mechanisms, which include tokens, signatures, etc. With the recent shift in identity solutions, from being organization centric to user centric, these platforms have no provision for open authentication mechanisms, such as OpenID [?, ?]. Overcoming the existing limitations and lack of provisions, this paper presents the design for OpenID-Authentication-as-a-Service APIs in OpenStack, including the implementation of a prototype for the proposed architecture.
We chose OpenStack for our research on cloud platforms, and its architecture is discussed in section II. In section III, we provide a detailed discussion on the shortcomings of cloud platforms, specifically OpenStack. Section IV includes a brief description of the OpenID authentication mechanism, followed by section V, where we present our innovative design for implementing OpenID authentication with APIs in OpenStack. The design is applicable to the Cactus release, as well as to the Diablo release, which was the stable version at the time of the ongoing research. Finally, the implemented prototype for the proposed design is discussed in section VI of the paper.

II. Cloud Computing with OpenStack Nova

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. At the beginning of this research, OpenStack had released the following versions: Austin (October 2010), Bexar (February 2011), and Cactus (April 2011). The stable release of OpenStack, Diablo, was then released in late September 2011.
OpenStack manages 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. OpenStack supports virtualization with KVM, UML, XEN, and HyperV, using the QEMU emulator. In the implementation, the *libvirt* [?] C/C++ library is used to communicate with the hypervisor from the middleware layer.

A. Architecture Overview

The components in the OpenStack architecture are: *Cloud Controller*, *API Server*, *Auth Manager*, *Nova-Manage*, *Scheduler*, *Object Store*, *Volume Controller*, *Network Controller*, and *Compute Controller*.
The Cloud Controller is the central component which represents the global state, and interacts with the other components. The Cloud Controller interacts with the API Server and the Auth Manager with internal method calls, with the Object Store over HTTP, and with the Scheduler, Network Controller and the Volume Controller, together, over the RabbitMQ [?] server using Advanced Message Queuing Protocol [?].
The 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.
The Auth Manager provides authentication and authorization services for OpenStack, which can interact with the Nova-Manage client using local method calls. Nova-Manage is an admin tool to communicate with the Auth Manager to directly interact with the OpenStack database.
The Scheduler is responsible for selecting the most suitable Compute Controller to host an instance, and the Compute Controller subsequently provides the compute server resources, according to the commands from the Scheduler. The Object Store component is responsible for storage services. The Volume Controller provides permanent block-level storage for the compute servers, while the Network Controller handles the virtual networks for the VMs to interact with the public network respectively.

B. Authentication and Authorization Framework

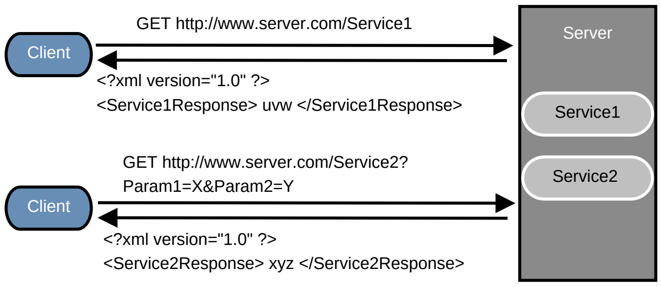
The authentication of requests from a user, and the authorizing of resources for the request are handled by the *Auth Manager* module. OpenStack uses a *Role Based Access Control* (RBAC) [?] mechanism to enforce policies.
When a user is created, an *Access Key* and a *Secret Key* are assigned to the 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 1. These credentials are used in different ways to authenticate a user's incoming API requests.

Table 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

C. 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). In all cases, OpenStack relies on Representational State Transfer (REST) to handle the responses from the APIs.
REST [?] 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 1.
In OpenStack, each RESTful service API is invoked with a corresponding URL on the API server over HTTP, which includes all the necessary parameters. The response from the API server is sent in a predefined format, in XML or JSON, according to the response format specified in the API call.



Client
GET http://www.server.com/Service1
<?xml version="1.0" ?>
<Service1Response> uvw </Service1Response>
Client
GET http://www.server.com/Service2?
Param1=X&Param2=Y
<?xml version="1.0" ?>
<Service2Response> xyz </Service2Response>
Server
Service1
Service2

Figure. 1: RESTful Client-Server Interaction

III. Problem Area

Identity management in web services is experiencing a paradigm shift, from organization centric, to user centric authentication mechanisms. User centric identity allows both scalability, and flexibility in application to multiple service points over the Internet [?]. Additionally, these user centric frameworks aim to provide Single-Sign-On (SSO) mechanism for its users, and thus, provide a certain leverage to introduce login federation, which greatly improves the usability for any service architecture.

OpenStack services can be utilized via API tools, such as *euca-tools* EC2API client [?] and the *python-nova* OSAPI client [?]. However, a graphical interface provides better usability, especially for users without much knowledge of API tools and commands. Thus, the web GUI has become a widely deployed front-end for delivering cloud services, both to administrators and users. The Dashboard/Django-Nova framework provides a suitable GUI for users. However, there are certain limitations in what the fronts-end GUI for OpenStack make available in the context of authentication of users.

OpenStack encourages the use of its APIs (EC2API [?] or OSAPI [?]) for implementing front-end GUI services. OpenStack performs authentication, based on access and secret keys. 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 (valid up to Cactus release). 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 participates to the front-end user authentication. Basically, OpenStack does not support federated identity management properties that are available today in OpenID [?, ?], Shibboleth [?], SAML [?], etc [?].

In standard federated login architectures, the Policy Administration Point (PAP) and the Policy Decision Point (PDP) are logically located at a single point in the architecture. There can be multiple Policy Enforcement Points (PEPs), which communicate with the PDP [?, ?]. However, as the front-end GUI server becomes a separate security domain in OpenStack, the front-end needs to maintain separate user credentials, due to an absence of a federated login architecture. The absence of a centralized authentication architecture causes the problem of multiple PAPs and PDPs. For the above mentioned reasons, we see that there should be complete trust (single security domain) 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 in terms of security, and also, with its separate user management system.

IV. Authentication with OpenID

OpenID is a well known open source authentication mechanism. It provides decentralized user centric identity management for web services, and allows seamless SSO authentication. The current version of OpenID is 2.0 [?, ?]. 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 2.

The User-Agent (UA) first requests a page over HTTP from a web service point, and the web server returns the page to the UA. The user then submits his OpenID Identifier. The web server acts as a Relay Point (RP). It normalizes the Identifier, and performs the discovery process, using Yadis discovery protocol [?] (XRI Resolution protocol [?] was used in OpenID 1.0, but is avoided in OpenID 2.0). The RP receives the meta-information from the OpenID Provider (OP) to redirect the UA to the OP endpoint URL. The RP then sends an `HTTP 302 Redirection` response to the UA, with multiple parameters.

After the UA is redirected to the OP endpoint URL, the OP can use any method to authenticate the user (such as username/password, certificates, smart-cards, generic bootstrapping architecture based device authentication, etc). After authentication, the OP returns the UA back to the RP, and passes a long string in the `HTTP GET` request line, also called the assertion URL. The RP verifies the signature in the assertion URL, and sets up a key association using Diffie-Hellman (D-H) key exchange [?]. Then, the RP verifies the response for the specific `openid.identity`.

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 [?] occurs in the initial discovery phase at the RP. The shared key is stored in a key database at the RP. The UA is then redirected to the OP. After authentication at the OP and when

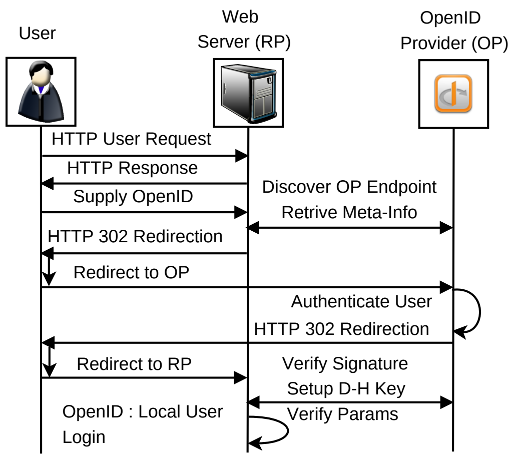


Figure. 2: Stateless OpenID Authentication Mechanism

the user is redirected back to the RP, there is no D-H key exchange. Instead, the stored key from the previous step is retrieved from the database.

V. OpenID in OpenStack

Implementing OpenID at the front-end GUI server as a simple RP is not the target. To apply OpenID authentication mechanism in OpenStack, we needed to combine a dual-PDP scenario into a single-silo formation.The following sections discuss the design considerations, and the solution architecture, followed by the usability and a detailed analysis of our solution. The design is applicable to OpenStack, till the Cactus release, which was the stable version at the time of the ongoing research. Later, it was extended to the second generation of the OpenStack architecture, the Diablo release, which introduced KeyStone, a separate OpenStack Identity Server [**?**].

A. Design Considerations

As shown in figure 3, integrating OpenID in OpenStack would have been a simple task. However, the design would have had multiple flaws according to basic security practices, including a dual-PDP scenario, mentioned in section III.
In an architecture to integrate OpenID with OpenStack, the front-end GUI should be a ”dumb” server, only processing the views for the user. There should not be any requirement for the GUI server to maintain any user credentials for authentication. Furthermore, even though the views on the GUI are based on responses from the API server, the initial authentication on the front-end should be granted by the **Auth Manager** in the back-end server. Additionally, as 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.
Therefore, the process of authentication of a user in the front-end should be realized as a service from the back-end server. Thus, we converged on a solution based upon OpenID-Authentication-as-a-Service for OpenStack. Furthermore, in our design, all interaction between the front-end and the back-end is stateless, as required by the RESTful API server [**?**, **?**].
Additionally, the design should meet all of the specifications of OpenID [**?**, **?**], and ensure all security requirements for OpenID in all phases of interaction. This would allow interoperability between all OpenID providers, thus greatly improving the usability for the users.

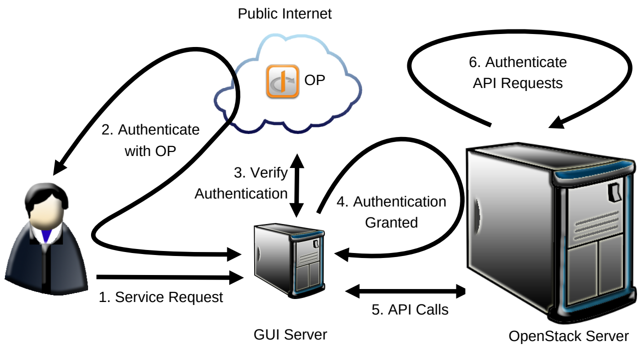


Figure. 3: Incorrect Architecture for OpenID Integration

Finally, the requirements at the front-end server to implement OpenID authentication in OpenStack should be simple, and secured. Also, we need to maintain a modular and distributed structure to comply with the current architecture and scalability of OpenStack.

B. Implementing OpenID as a Service

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 functions of the two APIs have been defined as:

- **Authentication Request API:** 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.
- **Authentication Verification API:** 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.

As shown in figure 4, the user requests for an OpenID based authentication to the front-end GUI server. The GUI server then invokes the initial authentication request API on the API Server in the back end. The back-end server responds to the request with all the necessary OpenID parameters required for the redirection. The GUI server parses the information, and sends a `HTTP 302 Redirection` to the UA, i.e in this case, the web browser.
The UA redirects to the OP, where it is authenticated. Upon successful authentication, the OP sends the UA to the front-end GUI server.
At this point, the front-end invokes the second authentication verfication API on the back-end API server. The back-end communicates with the OP, and completes all the processes for verification. Once verified, the authentication is granted by **Auth Manager**. Based on the authenticated user information, the front-end then allows the user to log into the managerial interface.

C. Design Analysis

HTTPS can be used to secure interaction with the UA. Additionally, it is required to protect the integrity of the assertion messages relayed from the OP through the UA, and includes nonce checking, and signature verification.
All RESTful requests to the API Server 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. Using SSL between the front-end and the API Server is a common practice for confidentiality in RESTful services. Security technologies such as IPSec [**?**] are intended for network level host-to-host security, rather than application-to-application security, and hence is not a recommended security solution

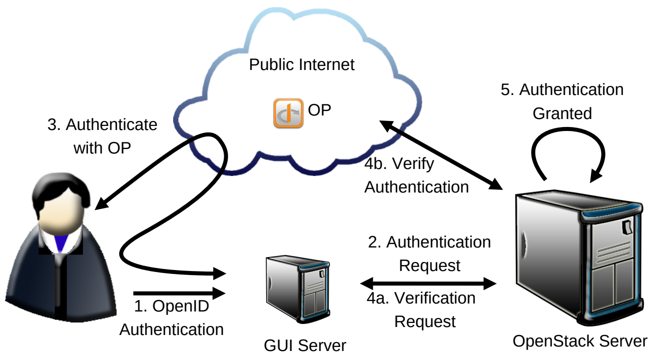


Figure. 4: OpenID Authentication in OpenStack

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 OpenStack server and the OP occurs in the back-end. However, the back-end communication with the OP cannot be considered hidden from an attacker. An attacker can sniff packets from the network to intercept the communication between OpenStack and the OP. Hence, this communication takes place over an encrypted channel with the D-H shared key between OpenStack and the OP.
However, there is still scope for an attacker to manipulate the information. Based on security issues of OpenID, the solution can be vulnerable to Discovery Tampering, Adversary Relay Proxy, and DoS attacks.
Session management between the UA and the GUI front-end is another area where the security should be improved. Services on OpenStack are RESTful services, and no session information is stored, while the front-end is a session-based service point for the UA. It is contradictory with the design principles of RESTful services to maintain such session based security. Therefore, OpenStack needs to trust the front-end GUI to manage the user session.

D. *Usability of OpenID in OpenStack*

A lot of discussions on the usability of OpenID have occurred so far. OpenID is the most widely used open standard for authentication, with many OP providers, such as Google[9], Yahoo[10], MyOpenID[11], and LiveJournal[12].
Integrating OpenID in OpenStack will provide a decentralized user centric authentication delegation for using OpenStack services, where the users will have control over his or her own identity management and authentication. Thus, usability will improve, as OpenID aims for a single user versus multiple service points applicability, and allows users to have a seamless SSO experience. This would allow providers to introduce a federated login architecture, and also reduce the IT maintenance cost by management of user credentials at third-party OPs.
Standard OpenID implementation also includes using the Provider Authentication Policy Extension (PAPE), to allow a flexible authentication framework. PAPE allows an RP to specify different requirements to be implemented at an OP during authentication. Thus, OpenID could utilize PAPE to implement a requirement based security in OpenStack.
Additionally, users will have flexibility in the authentication mechanism, as OPs allow different authentication mechanisms for their users. Most OPs support username/password, and client certificate based authentication for users. Apart from that, Leicher et al. in [**?**] describe a trusted computing environment using OpenID, A. S. Ahmed in [**?**] presents a 3GPP standard authentication mechanism for smart phones, and Watanabe *et al.* in [**?**] illustrate a cellular subscriber ID and OpenID federated authentication architecture. Ericsson Labs also provides an Identity Management service, which uses the 3GPP standard Generic Bootstrapping Architecture [**?**, **?**] based device authentication services with OpenID.

VI. Prototype Implementation

We initially started working 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. However, Diablo was released in October 2011, with a new architecture, which included Keystone, an Identity Server. Our design was then adapted and integrated with the OpenStack Diablo release of Keystone. Additionally, the implementation also included support for OSAPI from the Diablo release.

A. *Integration with the Bexar/Cactus Architecture*

In our design, we introduced the Nova-OpenID Controller module in the OpenStack architecture. The added module is responsible for all operations related to OpenID authentication in the back-end. It has an internal HTTP interface with the Cloud Controller, and a public interface to interact with the OP on the public internet. The implementation also included extension of the Nova-Admin tool. The admin can use the added functionality in Nova-Admin to add/modify OpenID information for existing OpenStack users.
The architecture of OpenStack, including the added modules for the prototype is shown in figure 5. We designed the two APIs according to the specification of EC2APIs on OpenStack API server. Furthermore, we implemented the invocation of the APIs from the Dashboard/Django-Nova web GUI for OpenStack. In our current implementation, we have a one-to-one mapping of existing OpenStack users to the number of enabled OpenID Identifiers a user can use. The implementation was tested successfully against standard OpenID providers on the Internet.
As shown in figure 6, the user provides the OpenID Identifier on the GUI, and subsequently, the front-end invokes the *OpenidAuthReq* API. The response contains the required parameters for redirecting the UA to the OP. After user authentication is completed, the UA returns to the front-end, and invokes the *OpenidAuthVerify* API. The back-end then verifies the assertion URL, links an existing OpenStack user to the verified OpenID Identifier, and returns a success or a failure. The front-end GUI then uses the information to allow or deny login to the user.

[9]Google OpenID Services, http://code.google.com/apis/identitytoolkit/
[10]Yahoo OpenID Services, http://openid.yahoo.com
[11]MyOpenID OpenID Services, https://www.myopenid.com
[12]LiveJournal OpenID Services, http://www.livejournal.com

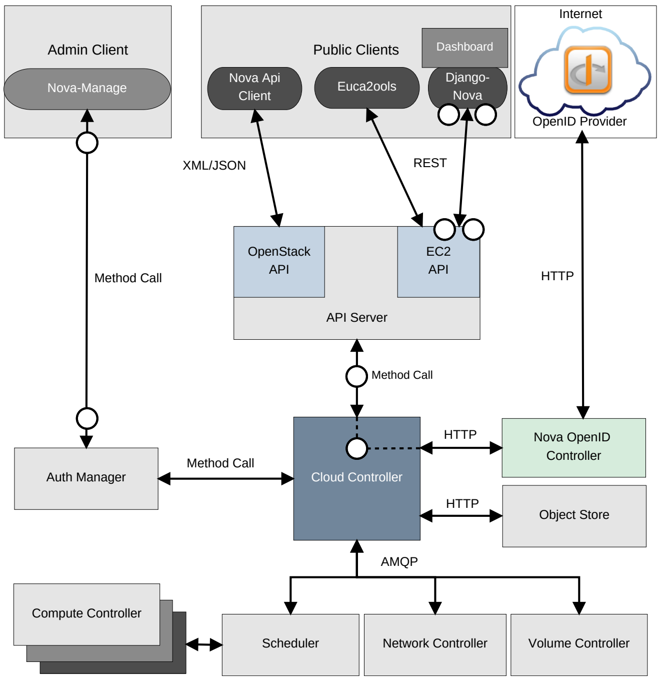


Figure. 5: Prototype Architecture for OpenID in OpenStack in Bexar/Cactus

B. *Integration with the Diablo Architecture*

Diablo, the fourth release of OpenStack, was released in October 2011. However, beginning with the Diablo release, the authentication framework design utilizes a new architecture. It includes the *KeyStone* project [**?**] as the identity authentication module.
As shown in figure 7, we added the OpenID module in the keystone server, the centralized Identity management server for OpenStack. This integration point simplifies the architecture, and retains the organization format of OpenStack, with Keystone being responsible for all authentication processing. Additionally, the same token is being generated, as to all other authentication mechanisms in Keystone. The OpenID module exposes two APIs for OpenID authentication, similar to the ones in the previous architecture. Hence, all authentication requests are received and processed at the Keystone API server.
After authentication process is completed successfully, Keystone returns an authentication token to the dashboard user as a security credential. After that, all service requests made by the client application to the OpenStack API Server include the authentication token. Once the API Server receives a request, it validates the token against Keystone. Upon a successful verification, the API Server replies with a service executed response. An authenticated token provides federation within all OpenStack components.
The signalling sequence among the different entities, when a user wants to use OpenID authentication in OpenStack is shown in figure 8. Initially, the user submits an OpenID URL on the front end GUI server. The GUI server calls the *OpenidAuthReq* API exposed from the Keystone sever. Keystone then performs discovery on OpenID URL to discover all meta data rested to the URL. After that, Keystone sends a browser redirection message against the submitted URL for the OP. The browser redirects to the OP, where the user performs authentication using any standard method. After successful authentication at the OP, the UA is sent back to the GUI server. The GUI server then parses the request, and calls the *OpenIDAuthVerify* API on the Keystone server. Keystone verifies the attached assertion URL included with the API call against the OP. Successful verification in Keystone results in a token and service profile information generated by the Keystone Auth module. This information is sent back to the GUI server. From this point, the GUI server will use this token to call any OS API service.

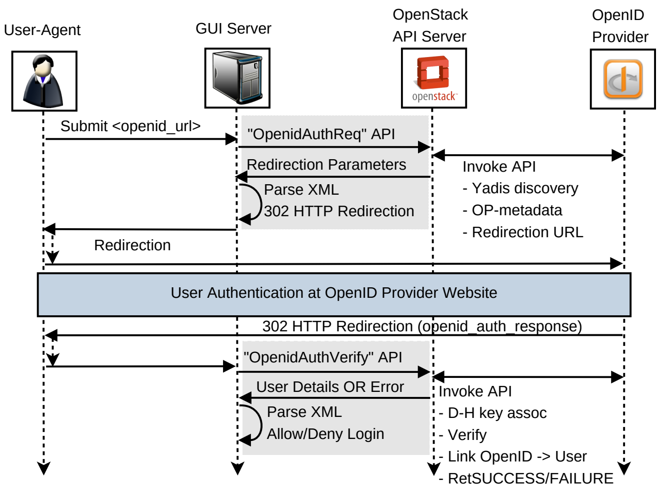


Figure. 6: Signalling Sequence for OpenID Authentication in OpenStack Bexar/Cactus

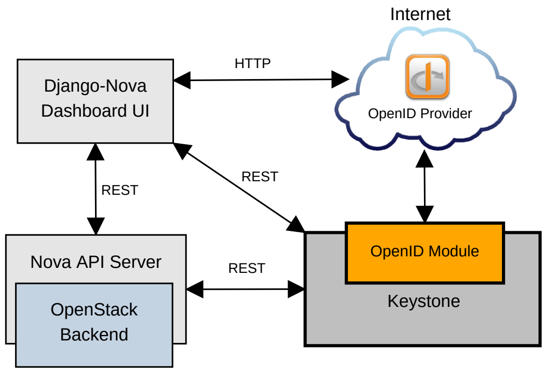


Figure. 7: Interaction Model for OpenID in OpenStack in Diablo with Keystone

C. *Prototype Evaluation*

Our implementation follows all the specifications of OpenID 2.0 [**?**, **?**]. Thus, authentication in OpenStack using OpenID is supported for all standard OpenID providers supporting OpenID 2.0. We verified use cases for authentication, including SSO authentication, using OpenIDs from *Google*, *Yahoo*, *MyOpenID*, and *Ericsson IDM Services* [**?**].
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 sys-

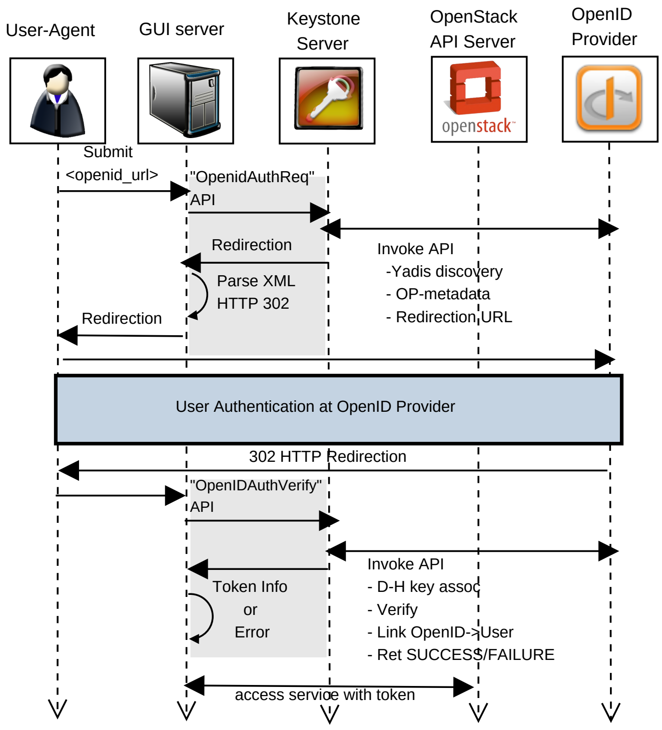
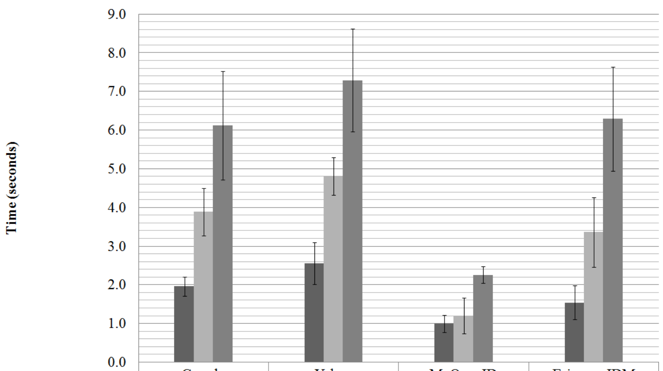


Figure. 8: Signalling Sequence for OpenID Authentication in OpenStack via Keystone

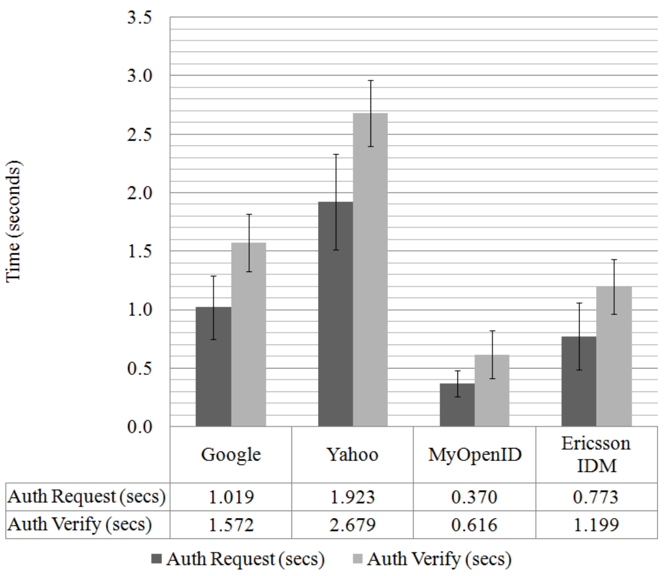
tem, on a Tower MacPro4.1, with 8GB RAM, and a 2.27 GHz Intel(R) Xeon(R) 16 Core 64-bit processor.
We captured network packets and calculated the time differences to evaluate the performance of our prototype. We recorded 30 observations for each OpenID provider. The times were calculated based on two 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 reauthenticate 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 9. The graph shows the mean of each set of readings, along with the population standard deviation for each OP.
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 inside the code, for the OpenID component alone, to evaluate its performance against external OPs. We recorded the duration of time 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 10. 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 10 for the internal timing measurements shows a similar pattern to the external measurements in figure 9. The standard deviations for all the OPs were also consistent. However, figure 10 does not show the timing for the verification phase to be twice the time required for the request phase as in figure 9. Additionally, the sum of



	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

■ Request (sec) ■ Verification (sec) ■ SSO (sec)

Figure. 9: Time Measurements for OpenID Authentication



	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. 10: Time Measurements for Processing API Calls

the internal timings on figure 10 is much lower compared to the external timing in figure 9 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.

VII. Conclusion

To improve usability, OpenStack proposes to utilize its API functions to provide a GUI for its users. For initial authentication, the front-end is required to incorporate a separate username/password validation. Additionally, the Keystone module, from the OpenStack Diablo release, handles all authentication related operations for an OpenStack backend server.
In this paper, we introduced a flexible user-centric decentralized authentication service for the front-end, using OpenID as an open-source authentication platform. A traditional OpenID authentication implementation executes the process at the front-end server. However, this introduces a dual PDP scenario, which is not a recommended practice for web services. In our design, OpenID authentication in the front-end is used as a service from the back-end. 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, and utilized the OpenID-Authentication-as-a-Service APIs from the Dashboard/Django-Nova GUI. The design was successfully implemented on OpenStack, including the new architecture with the separate Identity management with Keystone. The implementation of the prototype was followed by evaluation of its performance. The results obtained illustrates that the performance is well within acceptable limits, and is a successful design and implementation.

VIII. Future Works

The research performed during this work revealed further possibilities. The first objective would be to introduce greater flexibility in the choice of authentication mechanisms for the user. To provide a generic solution for authentication, we aim to design a common Authentication-as-a-Service API in OpenStack. To provide flexibility in the choice of authentication on OpenStack, we suggest that other authentication platforms, such as Shibboleth/SAML, be considered. Additionally, we also aim to introduce open platforms for authorization delegation within OpenStack. OAuth is such an open source authorization delegation platform, which allows a user centric authorization delegation specification, and allows two parties to securely interchange specific information about authorized resources.

Author Biographies

Rasib H. Khan is a researcher at Helsinki Institute for Information Technology, Finland, in the EU FP7 PURSUIT project. He has completed two Master's degrees from Royal Institute of Technology (KTH), Sweden, and Aalto University (formerly Helsinki University of Technology), Finland, as an EU Erasmus Mundus Scholar. His areas of interest include cloud computing, information security, user centric identity management and future Internet technologies.

Abu Shohel Ahmed currently works at Ericsson Research. He is also a doctoral student at Aalto university, Finland. He completed his Masters in Mobile Computing and Security from Aalto University. Ahmed's main research interests include protocol security analysis, identity and access management, and security in the cloud. He is also interested in trend analysis of future Internet technologies.

Jukka Ylitalo received his M.Sc. degree in 2001 and Ph.D. degree in 2008 from Helsinki University of Technology in Finland. Ylitalo's research themes are related to applied security in network architectures and services. His recent research has focused on cloud computing security.