J Grid Computing (2014) 12:3–27
DOI 10.1007/s10723-013-9283-2

Adding Federated Identity Management to OpenStack

David W. Chadwick · Kristy Siu · Craig Lee · Yann Fouillat · Damien Germonville

Received: 6 December 2012 / Accepted: 29 October 2013 / Published online: 5 December 2013

Abstract OpenStack is an open source cloud computing project that is enjoying wide popularity. While many cloud deployments may be stand-alone, it is clear that secure federated community clouds, i.e., inter-clouds, are needed. Hence, there must be methods for federated identity management (FIM) that enable authentication and authorisation to be flexibly enforced across federated environments. Since there are many different FIM protocols either in use or in development today, this paper addresses the goal of adding protocol independent federated identity management to the OpenStack services. After giving a motivating example for secure cloud federation, and describing the conceptual design for protocol independent federated access, a detailed federated identity protocol sequence is presented. The paper then describes the implementation of the protocol independent system components, along with the incorporation of two different FIM protocols, namely SAML and Keystone proprietary. Finally performance measurements of the protocol independent components, and the two different protocols dependent components are presented, before the paper concludes with the current limitations.

D. W. Chadwick (✉) · K. Siu · Y. Fouillat · D. Germonville
School of Computing, University of Kent, Canterbury, UK
e-mail: d.w.chadwick@kent.ac.uk

C. Lee
The Aerospace Corporation, El Segundo, CA, USA

Keywords Federated identity management · Federated access · OpenStack · Cloud

1 Introduction

OpenStack is a relatively new open source cloud computing project. It has rapidly become very popular since its first release on 21st October 2010. It currently boasts over 6600 members, comprising technologists, developers, researchers, and cloud computing experts from 87 countries [1]. Over 140 organisations are members of the OpenStack foundation, including many well-known multinational computer companies such as HP, Intel, IBM, AT&T, Cisco and Dell.

OpenStack provides several cloud services, which are all accessible via RESTful APIs [2]. There are no proprietary hardware or software requirements placed on installing these services and they are all customizable to different environments. Swift is the storage service which provides both block storage and object storage. Nova is the compute service used for provisioning and managing large networks of virtual machines. Neutron is the networking service used for managing networks and IP addresses. Glance is the image service which provides for the registration, discovery and delivery of virtual machine images. It can use Swift to store its images, Neutron to transfer them, and Nova to run them. Horizon is a web based graphical user interface (or dashboard) which allows administrators to manage their OpenStack installation.

Keystone is the identity service which authenticates users and provides them with an authorisation token to use the various OpenStack services. The services use the token to get authorisation information about the user, in terms of the user's ID, the project and domain the user is a member of, and the role he has in this project. The services use either role based access control to determine which privileges are available to each role, or access control lists to give direct access to users.

The current Keystone implementation is centralised, in that all users need to be enrolled in its database, either manually by the OpenStack administrator (via a command line interface or Horizon), or via bulk loading from a corporate database such as LDAP, before they can access any of the services. There are a number of well-known limitations with this design, such as users having accounts in each system they access and having to remember different credentials for each one.

This paper focuses on the addition of federated identity management (FIM) to OpenStack, which facilitates both flexible authentication methods and federated authorisation management. Federated identity management is a necessary but insufficient step in implementing federated clouds. Cloud federation is concerned with cloud service providers federating together in order to share their resources, so that when one of them is overloaded, it may, transparently to its users, move their work to another cloud provider that has spare capacity. We do not address this latter aspect of cloud federation in this paper.

Instead we concentrate on how we have added *protocol independent federated identity management* to OpenStack Keystone, so that any different FIM protocol can be plugged into the protocol independent infrastructure. In our design, authentication and authorisation become separate functions, and users no longer need to be pre-registered in Keystone. Instead, users may use their existing corporate credentials to access OpenStack services, and their existing identity attributes are mapped into OpenStack authorisation attributes. Users are automatically enrolled as temporary Keystone users, through the process of auto-provisioning.

The rest of this paper is structured as follows. Section 2 describes a motivating example for our research. Section 3 describes related research in the area of federated identity management and its application to clouds and Grids. Section 4 describes the current Keystone implementation for authentication and authorisation. Section 5 describes the conceptual model for federated identity management, authentication and authorisation. Section 6 describes the conceptual model for federated access to Keystone, and the important design choices we made. Section 7 describes the implementation, which includes plugins for two different federation protocols. Section 8 describes the performance evaluation of our implementation whilst Section 9 discusses the current limitations and concludes the paper.

2 A Motivating Example

Our motivating example for federated identity management in OpenStack is an *international geospatial data community cloud for disaster response*. Effective international disaster response requires the coordination of many stakeholders around the world. Natural disasters, such as massive earthquakes and tsunamis, are currently unpredictable with any significant accuracy or lead time. Hence, to respond, stakeholders must collaborate in real-time, on-demand. A critical part of any such collaboration is information sharing. While some IT resources within a disaster area may be destroyed or non-operational, many stakeholders and responders will be outside this immediate area. Clearly, the challenges of disaster response include *continuity of operations* and *communication within the disaster area* (most likely mobile devices). Nonetheless, as more and more IT resources and information becomes cloud-hosted, it is only a matter of time until collaboration for disaster response requires on-demand cloud federation.

Conceptually speaking, when a disaster occurs, a dynamic virtual organisation (VO) or federation needs to be instantiated whose member organisations are stakeholders in the disaster response effort. Each VO user should be able to access the new federation's resources using his existing organizational credentials, without having to wait for new credentials to be issued to him. However, not all users from each member organization will need to access the new federation, only those who have been assigned roles in the relief effort. These roles should define what authorisations

they have, e.g., reading satellite data archives, executing codes to identify damaged civic infrastructure, and storing intermediate data products where they can be accessed by first responders in the field. A federated authentication and authorization infrastructure is needed that can be rapidly constructed simply by configuring existing cloud infrastructures.

Such fundamental cloud federation use cases, as disaster response, have been recognized in the NIST US Government Cloud Computing Technology Roadmap. This is a large document in three volumes, giving guidance to US federal agencies on cloud adoption. Volume I of this document [3] identifies ten, high-priority requirements. Requirement 5 targets "Frameworks to Support Federated Community Clouds". Federated community clouds are necessary to support all manner of international, government-to-government, agency-to-agency, or business-to-business collaborations. Such efforts have been reported as part of NIST's on-going cloud workshop series [4, 5] and promoted within the Middleware And Grid Interagency Coordination (MAGIC) working group of the Networking and Information Technology Research and Development (NITRD) Program [6], whose charter is to coordinate IT research, development, and adoption across US federal agencies.

All of these activities clearly recognize the fundamental importance of federated identity management in cloud computing. Adding federated identity management to an open source cloud system such as OpenStack, will greatly facilitate further experimentation and evaluation of practical systems to enable an essentially *global identity ecosystem for federated inter-clouds.*

3 Related Research

Identity management in distributed, networked environments has been a recognized challenge for many years. The advent of cloud computing, and specifically federated inter-clouds, is just creating another instance of this challenge, only on a much greater scale and requiring more general solutions. Many groups and projects have identified the needs, motivations, issues and challenges surrounding FIM [7–11] and there are a few that are making concrete efforts.

Kerberos [12] was one of the earliest, being developed at MIT starting in the late 1980s. Kerberos uses an Authentication Server and protocol to manage secure interactions between a client and a *principal*. The Authentication Server initially issues a *Ticket Granting Ticket (TGT)* to the client. When the client wishes to communicate with a principal, it sends the TGT to the Ticket Granting Service that verifies the TGT and returns a Ticket and session keys to the client. These are then used to access the principal. The use of a third-party Authentication Server in Kerberos was a fundamental development of network security, and the Keystone identity service in OpenStack is modelled on Kerberos. However, Kerberos did not address the issues of discovery or interoperability in a heterogeneous environment.

In the late 1990s, Globus toolkit was developed to protect high performance computer (HPC) Grids. The Grid Security Infrastructure (GSI) [13] relies on an X.509 Public Key Infrastructure (PKI) [14] in which public key certificates are issued to users by trusted Certification Authorities (CAs). A HPC authenticates a user by requiring the user's client to sign a message, and then validating the signature with the user's certificate. Mutual authentication can be accomplished if the client and HPC trust the CAs that signed each other's certificates. SSO and delegation (or more precisely impersonation) are accomplished by using proxy certificates issued by the user's original certificate [15]. Authorisation is provided by embedding X.509 attribute certificates in the proxy certificate [16].

The Security Assertion Markup Language (SAML), an OASIS standard [17], built on X.509 by defining authentication and attribute assertions in XML rather than in ASN.1, and by also defining request and response protocols for carrying these assertions. Shibboleth [18], an implementation of the SAML standard, was developed as part of the Internet2 Middleware Initiative, starting in 2000, to address resource sharing among organizations that use differing authentication and authorization mechanisms. Shibboleth uses a third-party *Identity Provider (IdP)* to provide security assertions to the *Service Provider (SP),* as described in Section 5. SAML based federations have proved to be very successful in the academic community. For example, InCommon [19], which is operated by Internet2, provides a Shibboleth-based federation as a service to US universities, whilst

the UK Access Management Federation [20] provides a similar service in the UK. EduGAIN is a more recent development that is inter-connecting these national federations into one large global federation. By the fall of 2013 there were 20 national federations interconnected together, with another 5 in the process of joining, and an additional 5 as candidates for future participation [21]. These federations span the globe from Canada to Brazil, include most European countries, as well as Australia and Japan. EduGAIN is funded by GEANT, the 40 million euro pan-European research and education networking project.

Due to the relative success of SAML and Shibboleth, compared to X.509, the Grid world has embraced SAML in various ways. Globus toolkit built in support for SAML by allowing it to retrieve attributes from an IdP by using the SAML protocol [16], whilst the *Shibboleth Enabled Bridge to Access the National Grid Service* (SHEBANGS) project at the University of Manchester [22] enabled Shibboleth users to access the NGS using their institutional usernames and passwords instead of being required to obtain X.509 certificates first. The latter are generated on the fly by the Shebangs components, once the user has been authenticated by Shibboleth.

Meanwhile in the commercial world, IBM and Microsoft jointly developed the WS-Federation protocol standard [23] to build on the WS-Trust and WS-Security standards. In WS-Federation, a federation is a collection of *realms*, i.e., security domains. A resource provider in one realm can make authorization decisions based on *claims* about a principal asserted by an IdP in another realm. WS-Federation specifies how this can be accomplished by the brokering of identities, attribute discovery and retrieval, and the secure transport of claims among realms. WS-Federation is widely supported in Microsoft products. Ironically, one of the most common formats for WS-Federation claims are SAML assertions, although the WS-Federation and SAML protocols are incompatible, and therefore do not interwork.

OpenIDv1 and v2 [24] follow the same model of clients, SPs/relying parties (RPs), and IdPs as described in Section 5. When a client is requesting a web service, they can specify their preferred OpenID IdP, which the SP/RP can then use to authenticate the user. OpenID also offers an Attribute Exchange facility whereby different user attributes can be sent to the RP, depending on the user's preferences and the RP's requirements. In comparison, OAuthv1 and v2 [25] is a delegation of authority protocol, which allows a user to grant an SP access to the user's resources stored at another SP. One can immediately see if the "other SP" is actually an IdP, then the user could grant the SP access to its identity attributes at the "other SP", and thus gain authorised access to the SP's resources. However OAuth was not designed for this type of use case, and does not standardise the protocol details that are needed to implement FIM securely. For this reason OpenID Connect [26] is being developed, which is a profile of OAuthv2 specifically designed for federated identity management and SSO.

The latest FIM protocol to be standardised is the ABFAB protocol suite from the IETF [27]. Whereas most of the previous FIM protocols were developed for users who would be using web browsers as the client software, and were therefore capable of being redirected from one site to another, ABFAB is an extension of the Radius protocol [28] that is used extensively by the eduroam (**edu**cation **roam**ing) network [29] for wireless access. It does not require redirects, and is therefore suitable for use by command line and other non-browser clients. Eduroam is a federated authentication system widely used in the academic sector, which allows users from any participating university to authenticate to any other participating university's wireless network (the SP) by authenticating via their home university (the IdP). Whilst eduroam does not provide federated authorisation, ABFAB has added this by allowing SAML assertions to be transferred from the IdP to the SP as Radius attributes.

While such tools address a large segment of users and applications, they do not address the key issue of attribute management in the context of general cloud federations. One previous approach to this is the Virtual Organization (VO) concept [30], developed in the Grid computing community over the last ten years [31–33]. A VO is essentially a security context where each member of a VO can be associated with a set of authorization attributes (typically roles). These authorisation attributes can be managed for users from multiple administrative domains by using an external VO Membership Service (VOMS) server [34] that maintains all status information for a set of VOs. Once created, each VO has its own VO administrator. This administrator can define any number of groups or roles within that VO. The VO administrator can grant,

deny or revoke a user's membership in the VO. In current VOMSs, such as used by the Open Science Grid, a user authenticates to a VOMS for a specific VO. The VOMS replies with a SAML assertion defining the user's authorizations. The user's client uses this information to build an X.509 proxy certificate, based on the user's primary certificate, and this is used for authorization at the protected service. The Policy Enforcement Point (PEP) that is protecting the service consults a Policy Decision Point (PDP) to make the actual authorization decision. This established model of PEPs and PDPs that separates concerns, provides benefits for both manageability and scalability. To date, support for VOs has been integrated into the lowest levels of system software [35] and VOs are used operationally [30, 36]. Detailed information describing the operation of VOMS is available in [34]. The VO concept has been implicitly integrated into our OpenStack federated identity management design, through the use of an Attribute Mapping service, so that a separate VOMS server is no longer needed. Instead an administrator can say which IdP asserted identity attributes should be mapped into which OpenStack authorisation attributes. In this way, different groups of VO users can be created. The Attribute Mapping service addresses the semantic interoperabiity problem of mapping user identity attributes from potentially thousands of different IdPs into Keystone's model of tenants/projects and roles, thereby enabling proper authorization decisions to be made. This capability is fundamentally missing from Kerberos, Shibboleth and GSI. It is also REST-based, as is all of OpenStack, in contrast to WS-Federation, which is SOAP-based (even though a RESTful profile of WS-Federation could presumably be defined).

An important related issue is how to manage a user's authorization attributes that are asserted by several different IdPs and VOMS servers, for example, an employer IdP asserts an organisational role, whereas a bank IdP asserts possession of a credit card, and a VOMS server asserts group membership. Collecting these various assertions together and proving that they all belong to the same user, is the concept of *attribute aggregation*. It can be architected in several different ways. Cantor [37] has developed a simple attribute resolver for Shibboleth, whereby the SP pulls attributes from different IdPs based on a globally unique identifier that they all hold for the user. As Cantor admits, this is not a good solution for either user privacy or control, but it is simple to implement. However, ABFAB is taking the same approach, since the user's Network Access Identifier is globally unique. Chadwick and Inman [38] on the other hand have developed a privacy protecting intermediate service that sits between the client and the attribute providers, which performs the aggregation on the user's behalf, based on policies set by the user. They call this the linking service, and it is managed by the users themselves, without ever learning the true identities of the users. However in user trials carried out by Watt et al. [39], they found that users did not feel to be sufficiently in control of the aggregation, since the linking was done in real time during login based on policies that had been set sometime in the past. Consequently a revised system was built, called the Trusted Attribute Aggregation Service (TAAS), which allows the user to dynamically select his or her attributes from multiple issuers during session authentication with the SP [40].

While nearly all of the above protocols and services were developed prior to cloud computing, many of the issues they were addressing and the capabilities they were developing are directly relevant. Another decade of development and deployment of networked devices and systems, and the advent of on-demand, cloud resources, only underscores the need to integrate federated identity management into cloud software. However, the mechanism must be FIM protocol independent, due to the number of protocols that are now available. Our design allows for this.

At we noted in the introduction, cloud federation is a much broader topic than federated identity management. With regards to other cloud software stacks that are endeavouring to support cloud federation, the Reservoir project supports federation in at least two different contexts. First, Reservoir's *Virtual Execution Environment Managers (VEEMs)* can federate among themselves, allowing a VEEM to flexibly place Virtual Execution Environments (VEEs) locally or at any remote site [41]. This would be analogous to allowing a Nova-Scheduler to instantiate VMIs at other OpenStack sites. At the identity level, however, Reservoir is defining an *InterCloud Identity Management Infrastructure (ICIMI)* that is adopting the SAML model and architecture [42]. Thus it is limited in its federated identity management capabilities.

The EGI-InSPIRE project represents an existing, operational Grid that is incorporating cloud software

stacks [43, 44]. By leveraging much of the existing Grid infrastructure for authentication, authorization, monitoring, accounting, etc., a federated cloud testbed has been deployed that incorporates OpenStack, OpenNebula, and StratusLab sites. One of the goals of the EGI-InSPIRE project is to allow researchers to deploy the compute environments they want, where they need to. This includes traditional Grid software stacks. While this goal is laudable, the approach of EGI-InSPIRE is largely providing *brokerage services* across the different cloud platforms, rather than achieving a "native" federation among heterogeneous IdPs and SPs. It is the external X.509 PKI infrastructure that provides the SSO and distributed authentication functionalities, rather than a federated identity management capability that is integrated into the cloud services themselves.

The EU Contrail Project [45] is also pursuing the cloud brokerage model for cloud federation, but with a much more complete level of integration. Contrail users have a single interface whereby cloud resources from multiple cloud providers can be acquired. This is done by structuring Contrail into a *Federation Layer, Provider Layer*, and *Resource Layer*. When the user submits a job, the best provider can be selected based on the user's requirements. Applications can be scaled and migrated, using support for SLAs, the XtreemFS file system, virtual networks, and other capabilities. With regards to identity management, Contrail maintains an external database of SAML attributes and uses OpenID to verify user identities for the various resource providers. OAuth delegation is the primary mechanism whereby Contrail uses a user's credentials to manage the various cloud resources that have been acquired on the user's behalf. While this achieves an important and useful capability, Contrail is not actually integrating FIM into any cloud stacks, but is providing an interoperability layer across them.

Such work in cloud federation dovetails with broader goals in both North America and Europe. NIST's National Strategy for Trusted Identity in Cyberspace (NSTIC) program [46] is working to achieve improved security, privacy, scalability, and ease-of-use in a national "identity ecosystem". Likewise, the European Network and Information Security Agency (ENISA) is working to achieve resilience for critical information infrastructures which necessarily depends on federated identity management in clouds [47]. In September 2013 the UK Cabinet Office's Identity Assurance program signed contracts with five commercial IdPs for them to assert the identities of users for e-government services (http://digital.cabinetoffice.gov.uk/2013/09/03/identity-assurance-first-delivery-contracts-signed/). This uses SAML assertions between the IdPs and the government's trusted hub that acts as an attribute aggregator. These governmental efforts are supported by international organizations, such as the Open Identity Exchange (http://openidentityexchange.org/about), the Cloud Security Alliance [48] and the Kantara Initiative [49].

Finally, while not specifically targeting cloud environments, the ABC4Trust project [50] is an ambitious project to address federation and interoperability using Attribute-Based Credentials (ABC) created from either U-Prove or Idemix tokens. The Privacy-ABC model defines additional entities beyond the usual three: User, Issuer, Verifier, Revocation Authority, and Inspector, even though these functions could be combined in some deployments. An important goal of the ABC4Trust project is to preserve the user's *privacy*. The Privacy-ABC model provides *verifiable, certifiable*, and *scope-exclusive pseudonyms* to allow users to remain private while nonetheless ensuring authenticity. This applies for IdPs as well as SPs, such that IdPs can verify identity, yet not be able to track a user's every move. Clearly such goals are beyond that of the current work, yet its credentials could be integrated into our protocol independent FIM system by plugging in a new ABC4Trust protocol handling module.

4 Existing Keystone

4.1 User Authentication

Keystone is the identity management service of OpenStack. It is trusted by the OpenStack services to handle the creation and management of users' identities and credentials. Users are enrolled in a Keystone domain and made a member of one or more tenants (or projects—the two terms are synonymous with project being the newer term). A project is the account holder of a set of OpenStack services. Each user is given one or more roles within a project, and a role is used by a service to determine the user's access rights to

the service. The concept of a domain was only added in the April 2013 release of Openstack, termed Grizzly, so that a cloud service provider can partition its resources into different domains, and assign different client organisations to different domains.

When a user wishes to access an OpenStack service, the following steps are taken (these are shown pictorially in Fig. 1):

1. The user enters his credentials to the service's client, along with the request he wishes to make. Currently the OpenStack open source code only provides command line clients for each of its services, although a web based administrative client called Horizon is also available.
2. The client sends an authentication request to the Keystone server using the user's credentials.
3. If the credentials are correct, and the user specified the project he wishes to use, Keystone sends back a scoped token and a list of endpoints to the different services that are available to him/her. If the project was not specified, Keystone sends back an unscoped token. An unscoped token can only be used with Keystone to exchange it for a scoped token, by providing a project ID. The user needs a scoped token to access any of the OpenStack services.
4. The client chooses the endpoint of its service and sends the scoped token along with the user's request to the chosen service.
5. Assuming this is an opaque token and not a PKI based token, the service sends the scoped token to the Keystone server asking for it to be validated. (If it is a PKI based token the service can validate it itself by using the public key of the Keystone server.)

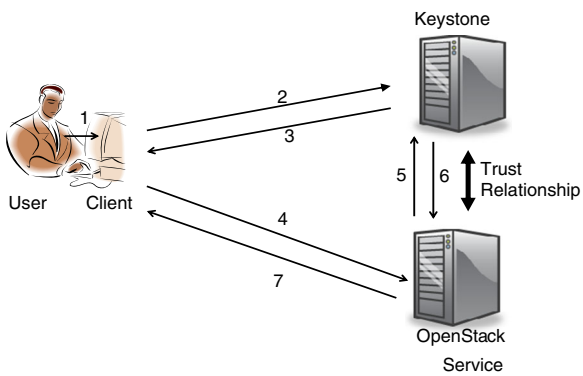


Fig. 1 Using keystone to access an OpenStack service

6. The Keystone server checks if the token is valid. If it is, it sends back the user id, domain, project, and the roles the user has in the project. The user is then authenticated with the service.
7. The service now makes an authorisation decision based on either the role and the project (and domain) that the user has (i.e. RBAC), or user id (i.e. ACL). If the user is authorised, then the service processes the request and sends the response to the client, otherwise the user's request is rejected.

4.2 Keystone Internals

Keystone, like all the other OpenStack services, is RESTful. REST is an architectural style for the client-server model [2]. In this architecture, a client performs an operation on a resource (server) by using a representation of the resource (identified by a URL). Each URL represents a different state of the resource. RESTful architectures like OpenStack often use the HTTP protocol because it provides the essential operators (or methods) for resource manipulation: GET (read), POST (create), PUT (update) and DELETE (remove). Thus all requests sent to Keystone are HTTP operations on a set of URLs. Each request is handled in two phases: firstly in a pipeline during which the requests are pre-processed, and secondly after the pipeline when the requests are directed to the right service module for handling them, see Fig. 2. The pipeline is composed of several middleware components, which are all configurable. Middleware components can modify both the incoming request and the outgoing response. A middleware component can also stop the processing of a request and can send a response instead of the main service module. New middleware components are easy to create and easy to add into the pipeline thanks to the in-built features of the Python language in which OpenStack is written. At the end of the pipeline, the request is sent to the Router module. The Router module sends the request to the appropriate service module of the Keystone code (e.g. for authenticating the user, listing the different users, validating a token, etc.) based upon the HTTP method and the path of the URL (the path is the last part of the URL the request was sent to e.g. for http://keystone:5000/v2.0/tokens, the path is /tokens).

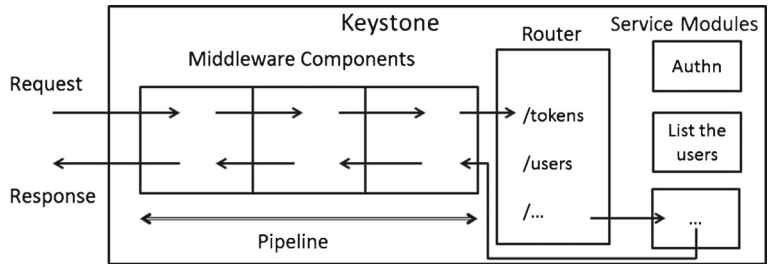


Fig. 2 Keystone internal architecture

In the current release of Keystone, the public service's pipeline is made up of the following middleware components:

- Token Auth: Initializes the context of the request for the next middleware component by copying the token ID in the header of the request into the request context.
- Admin Token Auth: Checks if a valid Administration Token is provided in the request and if so, updates the context so that the Keystone service doesn't need to do further authentication.
- XML Body: Detects if the content type of the body of the request is XML and if so converts it into JSON formatted content. When a response comes back, it will convert the JSON content back to XML, if the original request was formatted in XML.
- JSON Body: Parses and extracts the parameters of the JSON body.
- Debug: Logs all information about the request and the response.

The middleware components of the pipeline are configured in the Keystone configuration file.

5 Conceptual Model of Federated Access

Underpinning federated access is a trust relationship between the SP and the IdP: the SP trusts the IdP to authenticate and identify the user, whereas the IdP trusts the SP to privacy protect the user's identity attributes. Federated access is a three party protocol between the user, the IdP and the SP. The SP initiated variant is shown in Fig. 3.

1. The user wants to access a resource on a service provider.
2. The service provider determines the identity provider of the user. This is known as the *IdP discovery problem*, and can be solved by a variety of means e.g. manual selection by the user clicking on an IdP icon (known as the NASCAR solution, but also the NASCAR problem when there are too many icons to choose from), redirection to a separate Where Are You From Service (as used by Shibboleth), or automatic discovery (e.g. from the IP address or OpenID of the user, or because the SP only supports one IdP as was the case with Microsoft's Passport service).
3. The service provider redirects the user to his IdP. A request for authentication and the user's identity attributes is sent with the redirection.
4. If the user is already authenticated with the IdP, this step may be skipped if single sign on has been implemented. Otherwise, the user authenticates with his IdP. The authentication phase is usually not part of the identity management protocol and can differ between different IdPs, as each is free to choose its own preferred method.
5. After successful authentication, the IdP redirects the user back to the SP. An authentication response is sent with the redirection. The response

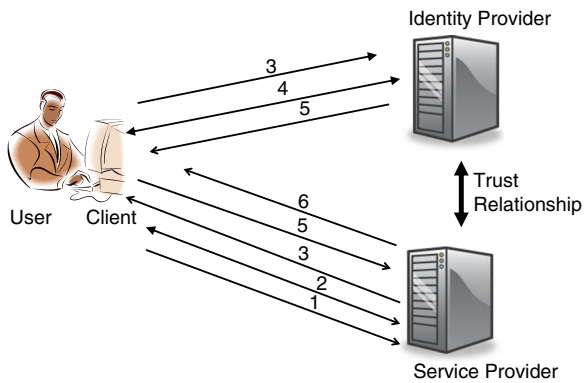


Fig. 3 SP initiated federated access to a resource

may contain the identity attributes of the user and may be encrypted and/or signed.
6. Now that the user has been authenticated and identified, he is able to access the SP's resource based on his identity attributes.

6 Conceptual Design of Federated Keystone

6.1 Design Principles and Choices

Several different design choices had to be made for the OpenStack FIM implementation. These were guided by some overarching design principles that we formulated at the outset. The chief one of these was *keep it simple* for the cloud service providers. The existing OpenStack services should not need to change when federated access is introduced. They already trust Keystone to authenticate and identify the users, so if the introduction of federation only effects Keystone, then the existing services should be able to use centralised or federated identity management without seeing any difference. This means that each cloud service can keep its existing projects and roles for authorisation and trust Keystone to correctly issue these to the users.

A second similar principle was *make it easy for the existing clients* to integrate federated access into their code or way of working. This posed a significant challenge due to the fact that the vast majority of today's IdPs assume that the user will be using a web browser for authentication, whereas all the OpenStack clients are command line clients. We investigated whether a command line client could sensibly interact via the HTTP protocol with existing IdPs that assume they are talking to a web browser, but this is either very difficult or impossible to achieve, since the authentication interaction is not standardised. Consequently the client cannot know what type of HTTP response content it will receive to its GET requests. The browser manufacturers already have similar difficulties today in interpreting web pages and knowing which fields can be auto-populated using the user's previously stored answers to similar questions. This is why in some cases today you will find that your email address attribute, say, may be auto-populated by your browser when you complete a form from one web site, and not, when you complete a similar looking form from another web site. If browser manufacturers, with all their resources, are not able to fully solve this problem today, then it is clearly beyond the scope of our small project. Thus our only practical solution is to use a web browser for the authentication phase (at least) of those FIM protocol implementations that expect it. This is because a web browser can easily render any type of IdP response into a web page that is viewable by the user, and the user, using his native intelligence, can easily see where he should enter his username and password, or other authenticating information, onto the web page. However, this left us with a major design choice. Should we use the existing command line clients for interaction with Keystone and the cloud services, and launch a web browser just for the authentication phase, or should we use a web browser throughout as the user's client for interacting with Keystone, the cloud service and the IdP. The latter approach would require us to produce a new web service that can act as both a service client to Openstack and a server to the web browser by providing the service client functionality to the user via a series of web pages. We decided that the former approach would both be much easier to implement and provide users with a similar experience to the one they are already accustomed to, whereas the latter approach would offer users a completely new access method to OpenStack services, which may not suit everyone. We therefore decided to implement the former approach, and to provide a single plugin module for the command line clients, that would handle the federated authentication exchange for them.

Another important design choice was 'what identity management protocol should be used? We decided that the actual FIM protocol is not that important, since FIM protocols are always being introduced and revised. Our design should be impervious to them. The design challenge then, is how to integrate them in a common way, given that they interact in very different ways. For example, the Shibboleth SAML implementation expects the IdP and SP to always communicate indirectly via the user's browser. OAuth expects the IdP and SP to have a back channel for direct communication, whereas the latest IETF ABFAB work expects the client to talk directly to the SP, the SP to talk directly to the IdP, and all requests from the client to the IdP to be routed via the SP. So how should all these different protocols be plugged into Keystone without effecting the overall architecture or conceptual model of federation? We decided that we would have one protocol dependent module that is responsible for both

creating the authentication and attribute request message(s) and getting this(them) to the IdP in a protocol dependent manner, and for validating the response(s) from the IdP, again in a protocol dependent manner, before finally returning the user's identity information to the Keystone code in a standard format. All protocol messages between the client and the protocol dependent module will be treated as opaque messages by Keystone, so that it does not have to process or understand them in any way. It will simply act as a relay, passing them to and from the protocol specific module. This module should be replaceable as necessary, and indeed, multiple different modules should be supported simultaneously, so that different IdPs that support different federation protocols, can be supported by the same Keystone implementation. The details of how this was achieved are given in the implementation section.

All the remaining federation functionality can then be made protocol independent. Firstly this needs to intercept the client's initial request and determine if federated authentication is requested or not. If it is, a new discovery component is needed that can determine which IdP (and hence federation protocol) the user should use. Once the IdP has been chosen by the user, its pre-configured supported protocol can be determined, so that the protocol independent federation code can call the appropriate protocol specific module to issue a request to the chosen IdP. This protocol specific module should now be responsible for all subsequent communications between Keystone and the IdP until the final IdP response is received and validated. The protocol specific module can then return the set of validated identity information to Keystone. However, there are thousands of identity providers in existence today, issuing possibly thousands of different attributes to their users, so secondly, Keystone must be capable of a) determining which of these are trusted and b) mapping between the trusted asserted attributes and the existing cloud service projects and roles which are used to make access control decisions. Thus a) an Attribute Issuing Policy (AIP) is needed to say which IdPs are trusted to assert which identity attributes and b) an Attribute Mapping (AM) capability is required that can take a user's validated identity attributes and map them into a set of authorisation attributes, namely domains, projects and roles, which the services can understand. These policies and mappings must be easily configurable by a Keystone administrator, for example, through configuration files or rest APIs.

An important design choice was how to handle the IdP discovery problem. We decided that a conceptual solution is for Keystone to have access to a directory service (Dir) of known and trusted federation IdPs. Querying the directory service will return the list of known and trusted IdPs, so that these can be presented to the client, allowing the user to choose his favoured one. The directory service approach can be used to provide each of the solutions to the discovery problem previously identified in Section 5. The returned list could be turned into a set of icons by the client (NASCAR approach), or displayed as a list or be searchable (the Shibboleth WAYF approach), or automatically selected if only one IdP is returned or the client is configured to know which IdP it should use. The only feature it cannot provide is automatic discovery of one IdP from a set of IdPs, since in the general case this is an intractable problem if user identifiers are independently chosen by the IdPs. Consequently we cannot be sure which IdP a user wants to use based solely on his identifier. For example, if an IdP A uses email addresses as user identifiers, and one of its users uses the e-mail address "example@gmail.com", the user will probably want to use IdP A to authenticate to the cloud and not Google (even though Google does act as an IdP for some service providers). In this case, the Keystone discovery service cannot automatically determine, based solely on the user's identifier, that the user wants to use IdP A as his IdP. Automatic discovery can only work if all user identifiers are globally unique and conform to a standard that is employed by all IdPs in a federation. Whilst this is the case for OpenID, EduRoam [29] and ABFAB [27] identifiers, it is not for federated identifiers in general. Thus automatic discovery will have to be left up to the protocol specific code for those protocols which support it.

Because trust is at the heart of federated identity management, the Keystone administrator has to trust the IdPs to correctly authenticate the users (*authentication trust*), and to assert the correct identity attributes for them (*identification trust*). Furthermore the OpenStack services trust Keystone to assign the correct authorisation attributes to users (*authorisation trust*). We should enable the Keystone administrator to limit this trust as far as is possible, so as to reduce the harm that would occur to an OpenStack service if the IdP made an error, or had imperfect

attribute assignment procedures. By controlling which IdPs are present in the directory service, we have a way of controlling the authentication trust. By introducing an Attribute Issuing Policy (AIP), we have a way of controlling the identification trust. By introducing the Attribute Mapping functionality, we have a way of controlling the authorisation trust. Providing all this functionality in the protocol independent code will ensure that Keystone only assigns authorisation attributes to end user in conformance to these policies regardless of the federation protocol.

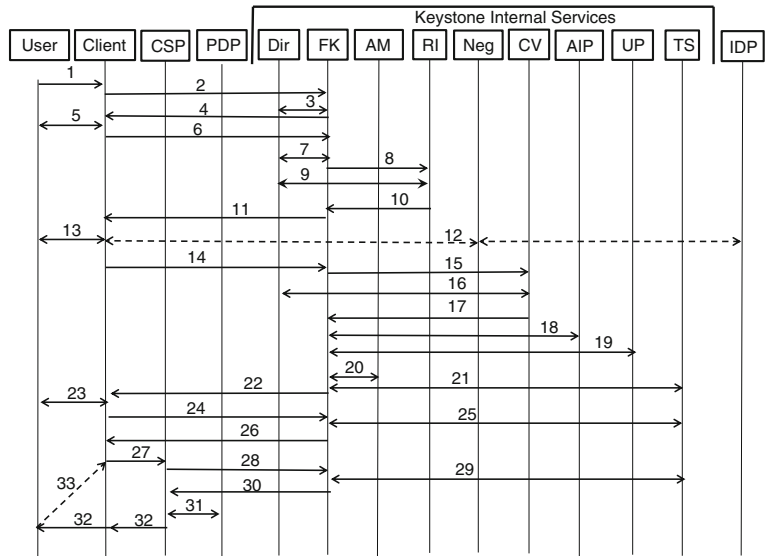
Provisioning of users is a massive task that is undertaken by all organisations today. When users join an organisation their identities have to be vetted, their qualifications have to be verified and their references followed up. They then have to be entered into the organisation's IT systems, and their credentials, roles and other attributes assigned to them. This information needs to be properly managed and kept up to date. Most importantly, when they leave the organisation, their credentials and other privileges need to be removed from the IT systems, otherwise they would continue to have (unauthorised) access. Much the same procedure has to take place when provisioning users in Keystone today. Their entries need to be created, credentials have to be assigned to them, then they need to be placed in the appropriate domains with the correct projects and roles. Their privileges need to be continually kept up to date, and when they have finished using OpenStack their entries need to be deleted. When we add federated identity management to Keystone, the potential number of users could explode exponentially. How are all these users to be efficiently managed? After much discussion, we decided that automatic provisioning and de-provisioning of users is the ideal solution. Since IdPs are already provisioning their users and keeping their identity information up to date, then we should leverage this effort in Keystone, rather than duplicate it by requiring the Keystone administrator to add and remove entries in its database. But automatic User Provisioning (UP) posed certain challenges, for example, how to know when an entry should be deleted, and how to ensure that the same user can gain the same access rights to the same cloud services on subsequent accesses, especially if his Keystone entry is different each time? Further details of how we solved these and other issues are given in the implementation section.

6.2 Protocol Sequence

Given the set of new functionality added to federated Keystone (FK): Directory (Dir), Request Issuing (RI), Federation Protocol Negotiation (Neg), Credential Validation (CV), Attribute Issuing Policy enforcement (AIP), User Provisioning (UP) and Attribute Mapping (AM), coupled with the existing Token Issuing and Validation Services (TS), the call sequence for federated access becomes that shown in the swimlanes of Fig. 4. The steps are described as follows:

1. The user types in the command to the cloud service client e.g. C:\Python27\Scripts>python swift -F -A http://fedkeystone.sec.cs.kent.ac.uk:5000/v2.0/ list textFiles. This command is calling the Swift client, asking for federated login (-F), giving it the address of the Keystone service (-A <address>) and asking for a listing of the files in the textFiles directory.
2. The client calls federated Keystone asking for the list of IdPs.
3. Keystone calls the Directory Service to obtain the list of federation IdPs.
4. Keystone sends the set of IdPs to the client.
5. The user chooses which IdP he wishes to use.
6. The client returns the chosen IdP to Keystone and asks for an IdP request message.
7. Keystone performs a directory lookup on this Identity Provider, to determine its supported protocol
8. Keystone asks the appropriate protocol dependent Request Issuing function to create an authentication and attribute request in the appropriate format for the chosen Identity Provider.
9. RI makes a request to the Directory Service to obtain the IdP metadata.
10. RI returns the IdP request message to Keystone.
11. Keystone returns the request message transparently to the client.
12. The client passes the request to the IdP (either directly or via Keystone, both mechanisms are supported). Any negotiation or other protocol changes that are needed are also transparently supported (see implementation section for details).
13. The IdP asks the user to authenticate by its chosen method (out of scope of this proposal). It could be PKI based, Kerberos based, simple UN/PW, or OTP etc. How strongly the user is

Fig. 4 Call sequence



authenticated is reflected in the Level of Assurance (LoA) that is returned in the response (providing the protocol supports this). After the user's credentials have been validated, the IdP creates its response, which may be returned directly or indirectly to Keystone's Credential Validation function (both mechanisms are supported).

14. The client passes the IdP's response to Keystone.
15. Keystone passes the IdP's response to the protocol dependent Credential Validation function to validate it.
16. Credential Validation calls the Directory to obtain the meta data of the IdP in order to validate the response.
17. Credential Validation returns the user's ID, set of identity attributes and IdP, and validity time to Keystone.
18. Keystone calls the Attribute Issuing Policy checker to ensure that only allowed attributes are asserted by the trusted IdPs.
19. Keystone calls the User Provisioning module, deletes any expired entries, and then creates (or updates) the temporary entry for the user.
20. Keystone calls the Attribute Mapper to obtain the local set of authorisation attributes that are equivalent to the IdP provided identity attributes.
21. Keystone updates these attributes in the temporary user entry then calls the Token Service (TS) to obtain an unscoped token for the user.
22. Keystone returns the unscoped token to the user, along with a list of domains and projects available to the user.
23. The user choses which domain and project he wishes to use.
24. The client passes the unscoped token and chosen domain and project to Keystone.
25. Keystone calls the Token Service to validate the unscoped token and issue a scoped token for the user, for either the domain or the project.
26. Keystone returns to the client the scoped token and list of cloud services at which this can be used.
27. The client contacts the Cloud Service Provider with the scoped token, requesting the service.
28. The Cloud Service Provider passes the scoped token to Keystone for validation.
29. Keystone contacts the Token Service to validate the token and gets the response.
30. Keystone sends the response to the Cloud Service Provider.
31. The Cloud Service Provider asks the PDP if the user is authorised for this request.
32. Assuming the PDP's reply is granted, the Cloud Service Provider provides the requested service to the client.
33. The client makes additional requests to the Cloud Service Provider using the same scoped token, goto 27.

7 Implementation

7.1 The Directory Service

Keystone already has a simplistic directory service which it calls the Service Catalog. This is used to store details about the various OpenStack services. A catalog entry consists of a service and one or more endpoints for (or instances of) the service. The data stored about a service is its type, service_id and an extra field (that is a JSON data structure that can essentially store anything, but this is not indexed in the backend database). In the Grizzly release the extra field contains the service's name and description. Multiple endpoints can be added to define which service instance should be used depending on the region of the cloud deployment. The data stored about an endpoint is its endpoint_id, region, service_id, interface, URL and an extra field (that again is a JSON data structure). In the Grizzly release, the interface represents the type of the URL of this endpoint, either 'public', 'admin' or 'internal'. (In older versions of OpenStack, the URLs for a given service were stored in the extra field of the endpoint entity with all three types stored in a single entity.) There are RESTful operations for creating, deleting and reading services and similar operations for creating, deleting and reading endpoints.

The details needed to define an Identity Provider in the conceptual directory service are similar to the service details in the service catalog. For example, the service name in the catalog can be used to hold the ID of the IdP that is used to uniquely identify it within a federation; the public URL can be used to hold the endpoint of the IdP that the client must contact (in step 12 of the protocol), whilst the service type can be used to indicate that the service is an IdP. Consequently an enhanced service catalog can be used to provide the required directory service. Because the JSON data structures are unconstrained, then different federated identity management protocols can store different fields in the service and endpoint entries, as the need arises.

In order to allow for different types of IdP, we have used the format <type>.<subtype> for service type, where <type> is set to 'idp' and <subtype> is set to the protocol used by the IdP. In this way the federation protocol independent code will know which federation protocol specific module to call, and the latter will know how to formulate the correct protocol requests to the IdP. Our initial implementation supports the SAML 2.0 protocol and Keystone's proprietary protocol for authenticating and authorising users, so we have defined the service types 'idp.saml' and 'idp.keystonev2' to denote these. A new display name extra field in the service catalog is used to hold the user friendly display name of the IdP that is shown to the user (in step 5 of the protocol), although the existing service description field can be used instead.

Protocol specific extensions to the catalog are currently as follows: for SAML, the X.509 public key certificate of the IdP endpoint, which is needed by the Credential Validation function to validate the signed SAML responses from the IdP, is stored in a new certdata JSON field; for KeystoneV2, either the public key certificate of the Keystone IdP and its Certification Authority required for PKI token validation, or the administrator username and password required to authenticate to the Keystone IdP for opaque token validation, or both, are stored in a JSON creds field i.e. {creds: {certdata:"x509data", cacert:"x509cadata"}} or {creds:{username:"un", password:"pass"}}. Corresponding parameters have been added to the Keystone endpoint and service creation APIs so that the information can be added to the service catalog.

The service retrieval API does not need extending as the service read API method allows the enhanced catalog module to retrieve the list of IdPs alongside the standard list of services. All information in the extra fields is already retrieved so no additional code is needed for this. However we had to modify the retrieval code that so that the administrator password of the Keystone credential is not retrieved. Applications or modules using the extended service catalog are responsible for handling the additional data in the extra fields. So when a request for a list of trusted IdPs is received by federated Keystone (step 2 of the protocol) the protocol independent code retrieves the list of IdPs via a call to the service catalog to retrieve the list of all services, and sorts these by type. The resultant list is used to produce a new list of IdP {'displayname': 'service _id'} pairings which are returned to the client. When a request is received by federated Keystone either for an IdP request message (step 6 of the protocol), or to negotiate with an IdP (step 12) or to validate an IdP response (step 14), the protocol independent code makes a call to the service catalog to obtain the type of the service, and from this can call the correct

protocol dependent module. The protocol dependent modules can store whatever additional information they need in the service catalog, and retrieve it when they are called.

7.2 The Federation Protocol Dependent Modules

Each federation protocol specific module must support three Python functions, although one of them (negotiate) can effectively be null.

The first, "getIdpRequest", will be called after the user has chosen the IdP that he wishes to use. The protocol module should create an authentication and attribute request message to be sent to the chosen IdP.

The second function is "validate", which will be called after the IdP's response has been received. It will be passed two parameters: the ID of the IdP used to authenticate the user, and the (untouched) response from the IdP. Each of the protocol specific validate functions must return the user's identity information in the same format. This should comprise three parameters:

- a federation wide Unique ID of the user. This allows the protocol independent code to ensure that the same user entry is always created for the same end user, regardless of how many times it may be created and deleted (see Section 7.4). If an IdP is not able to issue a unique identifier, for example, for privacy protecting reasons, then it means that any cloud services that use user IDs in their access control lists won't work correctly with federated users. However, cloud services that use PDPs and identity attributes for authorisation, should not be affected by this lack of Unique ID.
- a set of {set of user identity attributes and the name of IdP that asserted them}. By using the 'set of set of' construction, we have built in support for future attribute aggregation, whereby the user may obtain multiple sets of attribute assertions from different IdPs. Once these attributes have been validated (see Section 7.3), they will be mapped into OpenStack authorisation attributes of domains, projects and roles.
- the validity time of the asserted identity. FIM protocols typically include a validity time with their assertions. The returned value should be the intersection of all the aggregated IdP asserted validity times. It will be used by the auto-provisioning module to control the lifetime of the automatically created user entry. If no validity time is returned, then the user entry that is created will have an infinite lifetime (as in the existing Keystone implementation) and the entry will subsequently need to be deleted by administrative means.

The third optional function, "negotiate", is only needed when multiple message exchanges are needed between the SP and the IdP. The IETF ABFAB protocol is an example of this type of federated access protocol. This function has two parameters: the ID of the IdP, and the message from the IdP to the SP. The module processes the message, computes the response to the IdP, and returns it to Keystone which returns it to the client. There is no limit to the number of such messages that can be exchanged via the client, so that any arbitrarily complex FIM protocol can be supported. Once the negotiation has been completed, the client signals this by sending the validate message to Keystone.

Protocol modules which do not require negotiation are still expected to implement the negotiate function with a body which simply raises an exception to denote that this is not implemented or supported by the current protocol. This is to ensure that the Keystone server can handle erroneous requests from a client.

7.2.1 The SAML Implementation

The SAML version we have implemented is SAML v2.0 [17]. The Request Issuing function supports the SAML Web Browser SSO Profile [51]. In this profile, the service provider sends a HTTP redirect message to the browser, which causes the browser to send it to the IdP. The request contains the "return to" address, which in our case is set to "localhost"—this is explained in step 6 of Section 7.7. Request Issuing requires a PKI private key for signing the requests that it creates. The corresponding public key certificate needs to be inserted into the SAML federation metadata so that IdPs can validate the request signatures, and optionally encrypt any responses. It also requires a service provider name, which similarly needs to be inserted into the federation metadata so that IdPs can identify it. All of this information is specified in the Keystone configuration file.

Credential Validation needs the IdP's public key in order to validate the signature of the returned response. It obtains this from the extended service catalog. It also requires access to the same private key as

Request Issuing, so that it can decrypt any encrypted SAML responses. This is similarly obtained from the Keystone configuration file.

7.2.2 The Keystone Proprietary Protocol Implementation

Once we had designed and implemented the overall federation architecture and the first SAML implementation, the Keystone proprietary protocol implementation was relatively trivial to perform. No changes are needed to the Keystone implementation that is to act as the IdP. A standard OpenStack release is sufficient. All the Keystone administrator needs to do is add the federated Keystone to its service catalog, as a new type of cloud service. The user can then choose this service as the recipient of the scoped token that is issued to him/her after authentication.

The federated Keystone's protocol specific module needs to be capable of validating both types of scoped token that Keystone supports: PKI based and opaque. PKI based tokens are digitally signed by the issuing Keystone, and can be validated by the recipient through the standard procedure of digital signature validation. Opaque tokens on the other hand have to be passed back to the Keystone issuer for validation. The requestor, typically a cloud service such as Swift or Nova, must have an administrator username and password for Keystone. This stops end users from validating opaque tokens. The federated Keystone administrator therefore needs to add the standard Keystone to its service catalog as an IdP (of type idp.keystonev2) along with the credentials that are needed for validating its tokens (as described in Section 7.1). The protocol specific code can now validate any PKI tokens directly, whilst opaque tokens have to be passed back to the Keystone IdP for validation, along with the shared administrator username and password. Once the token has been validated, its contents (user id, domain, projects and roles, IdP name and validity time) are passed back to the protocol independent code for further processing (policy validation and attribute mapping). Consequently, the federated Keystone administrator must set up some attribute mapping rules which map from the Keystone IdP issued attributes (projects/roles etc.) into the local ones that it understands, and then a local scoped token can be issued to the user allowing him/her to access the local OpenStack services.

This process of federating Keystones together is of course recursive, in that multiple Keystones can be chained together in this way. The token issued by one, can be validated by another. Alternatively no Keystone need act as an IdP, as one could use an external SAML IdP for authentication and then pass the user and its scoped token off to another Keystone for the latter to validate and grant access to its resources. Furthermore, the trust relationship between two Keystones can also be made commutative, in that users could authenticate to either Keystone as an IdP in order to gain access to the services of both Keystones.

7.3 The Attribute Issuing Policy

As described in the previous section, the Attribute Issuing Policy is a means for the Keystone administrator to limit the amount of identification trust it must place in each IdP. This policy lists pairs of IdPs and attribute types. An IdP is only trusted to issue the identity attribute types it is paired with. The protocol independent code checks the list of identity attributes returned by the protocol dependent module for each IdP, and discards those attributes that are not paired with the given IdP in the policy. In this way, we can tightly control which identity attributes are deemed to be valid for subsequent mapping into OpenStack authorisation attributes. A REST API has been defined for setting and reading the Attribute Issuing Policy. This allows administrators to define which IdPs are trusted to issue which identity attributes.

7.4 User Auto-Provisioning

We decided to base the lifetime of the user's auto-provisioned Keystone entry on the validity time of the identity assertion provided by the IdP(s) and returned by the protocol dependent module. In this way, the protocol independent code can automatically create a new Keystone entry for a user when he/she first accesses the system, and this entry can be automatically purged after it has expired. There was however a number of implementation issues that we had to address with this design. Firstly, every Keystone user entry needs to have a username, user id and password. The user id is automatically created by Keystone when the entry is first created, to guarantee its uniqueness. Clearly we did not know any of these values

for the federated users. So we decided that the username and user id would be automatically created from the Unique ID returned by the protocol dependent module. Because the Unique ID could be of any arbitrary length, it is first hashed using SHA1 to create a 20 byte string, then it is converted into a 40 hexadecimal character string. This hex string is used for both the username and user id. If the protocol dependent module is capable of returning the same Unique ID for the same user, each time she authenticates via the same IdP, then the same entry will be created in Keystone's database each time. If it is not, then different entries will be created for the same user, though they will all have the same set of authorisation attributes, and therefore the user will still get the same access to OpenStack resources, provided they use RBAC policies and not ACLs.

The password proved to be more problematical. At first we simply generated a new random password for each new user entry, and then immediately discarded it, since the user would never need it for authentication. However, we subsequently realised that Keystone's Token Service API requires that both the username and password are presented each time a scoped or unscoped token is created. Consequently, the protocol independent code needed to access each user's password, but this is not retrievable from Keystone's database. We therefore decided to generate a strong random password during initialization of the Keystone server, hold this in memory (only) and use it for all the temporary users' passwords. The strong password is generated as follows: a randomly generated number of size 128 bits is created, concatenated with the current system timestamp in milliseconds, reduced to 20 bytes by using the SHA1 algorithm, then base 64 encoded to produce a 28 character password. This is stored in memory and is lost when the system is stopped. A side effect of this, is that any existing Keystone tokens that were created by the federated middleware before a shutdown, will be invalidated at restart time if the user authenticates again, as a new password will have been inserted into its entry. It is a built in mechanism of Keystone user management to invalidate existing tokens when a user's password is changed. However, we do not think that this is a significant inconvenience to users, since their tokens will still be valid if they do not authenticate again, but if they do, then they will need to do it for every token that they hold (and not just for one of them).

We had to modify the existing Keystone user creation function by adding two new optional parameters to it: user id and validity time. If neither are present, an entry with a new randomly generated user id and no validity time is created, as per the existing code. If either or both are present, then the newly created entry will contain these values. The process of auto-provisioning now runs as follows. Search the user database for entries whose validity time is before the current time. Delete these entries. If a user entry with the computed user id already exists, then update its validity time to match the latest value, else create a new temporary user entry with the name and user id created from the Unique ID and the validity time returned from the protocol specific module.

7.5 Attribute Mapping

In order to fully integrate federated access into Keystone it was necessary to implement an identity attribute to domain/tenant/role mapping service, which can perform many to many mappings. This required adding several new data entities to the Keystone backend storage, as well as adding several new API methods. We added a new path to the Router of '/mappings' so that requests to add, delete or read mapping entities can be routed to the new processing module. The attribute mapping service makes use of the existing backend entities domain, tenant and role. In addition, the following new entities were added:

i) **Organisational attribute set** represents a set of IdP asserted identity attributes, which can be mapped to a set of Openstack authorisation attributes i.e. domains, tenants and roles.
ii) **Organisational attribute** represents an identity attribute that can be assigned by an IdP to a cloud user.
iii) **Organisational attribute set association** represents an association between an organisational attribute set and an organisational attribute.
iv) **OpenStack attribute set** represents a set of Openstack authorisation attributes, which can be mapped to a set of organisational attributes.
v) **OpenStack attribute set association** represents an association between an OpenStack authorisation attribute set and either a domain, tenant or role.
vi) **Attribute mapping** maps an OpenStack attribute set to an organisational attribute set.

New RESTful APIs have been specified for creating, deleting and reading each of these entities.

Figure 5 illustrates two attribute mappings:

1. The Openstack "admin" role, "member" role and "kentusers" tenant are mapped to two organisational attributes: organisation = kent and accountType = staff.
2. The Openstack "member" role and "kentusers" tenant are mapped to two organisational attributes: organisation = kent and accountType = student.

7.6 Integrating the Components into Keystone

As described in Section 4.2, the first place where we can process a federated request is in a new middleware component of the pipeline. The second place where we can process a federated request is at the end of the pipeline, by modifying the Router module. To add a new route, e.g. "/federated", Keystone must be modified by adding a new rule to the Router and a new module to the Keystone implementation to handle the request. To add a new middleware component the Keystone configuration file (keystone.conf) must be altered. The main distinction between a new middleware component and a new core Keystone module is that the middleware only needs to be enabled in the configuration of Keystone, whereas the core module must be integrated into the Keystone code base.

Our federated solution uses both approaches. Since the existing service catalog is a core module, then the modified version that stores IdPs also has to be. The attribute mapping function, as well as the attribute issuing policy function, were also implemented as new core modules, whereas the protocol specific modules and the protocol independent module are combined into a new pipeline middleware module called the federated middleware module. In order to signal to this module that federated access rather than centralised access is required, each REST message from the client to Keystone needs to indicate this. We represent this

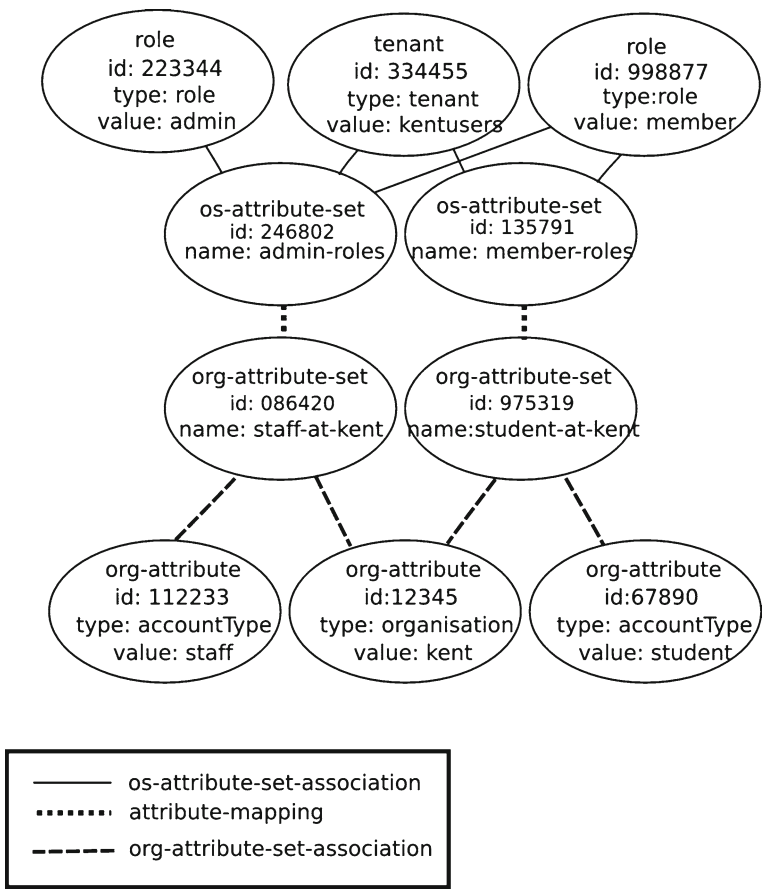


Fig. 5 An example of attribute mapping for the default domain

by inserting a new header *X-Authentication-Type: federated* into the HTTP messages. This signals to the new pipeline module that federated access is needed, and that the request should be trapped and processed by it. Different actions are taken by this module depending upon the contents of the HTTP body. When the body is empty, this indicates that the client is just starting the federation process, and that the list of trusted IdPs should be returned from the directory service. When the body contains an idpRequest JSON element, this indicates that a request message to the identified IdP is required. When the body contains an idpNegotiation element this indicates that the encapsulated message is from the IdP and that a further protocol negotiation message is required from a protocol dependent module. The protocol specific module is then called to produce the next message in the sequence, which is returned to the client by the pipeline module. When the body contains the idpResponse element it indicates that federated authentication has completed and the encapsulated message is the final one from the IdP.

7.7 Modifying the Clients

We modified three of the Openstack command line clients (for Swift, Glance and Nova) to support the SAML 2 Web Browser SSO Profile [51]. The clients did not have a standard implementation structure, so in order to minimise code repetition and ensure a consistent user experience, we decided that it was wise to implement a single module to manage federated access, which could be easily integrated into each and every client. We added two new command line flags to the entry point scripts of each client:

- –federated or -F – to denote that federated authentication should be used. Unlike standard authentication methods, this does not require the username and password to be given.
- –realm or - R – the federated realm to use (Optional)

When the federated flag is encountered the script delegates authentication to our new module, which manages the federated authentication before returning control to the standard client code once a token has been successfully obtained. In order to allow the Keystone middleware to detect when federated authentication is being attempted, the module attaches a HTTP header to each request containing the key:value pair "X-Authentication-Type:federated".

Once the federated authentication process begins the following steps are followed by our new client module:

1. It requests a list of available IdPs from the specified Keystone server (step 2 of the protocol).
2. When the list is returned, if the user has provided a realm then this is automatically selected from the list otherwise it displays a numbered list of available IdPs to the user (step 5 of the protocol), reads the user's choice and uses this to select the IdP from the list.
3. It calls Keystone to obtain an authentication request and IdP URL (step 6 of the protocol).
4. When the response is received it opens the user's default browser passing it the obtained message. This causes the browser to send the message to the SAML IdP.
5. The user authenticates to the IdP in the usual manner via the browser.
6. It opens an HTTP server on the user's localhost to receive the returned authentication response from the IdP (step 12 of the protocol), which is redirected there via the browser.
7. It captures the IdP's response, stops the HTTP server, then calls Keystone with the response in order to retrieve a list of available tenants and an unscoped token.
8. If the user specified the tenant he wishes to use as a command line parameter, then the client picks this from the list, otherwise it presents the user with a numbered list of available tenants (step 23 of the protocol) allowing him to choose one.
9. Finally it returns control to the standard client code, which exchanges the unscoped token and chosen tenant for a scoped token and a list of service endpoints.

The HTTP server that is opened has to work over SSL/TLS in order to conform to the SAML protocol. This means that it needs a PKI key pair. Consequently the new module requires a small amount of configuration data. This is accomplished by reading a configuration file (federated.cfg) at runtime. The file contains three parameters:

- TIMEOUT—the maximum time in seconds to wait for an IdP response before closing the HTTP server and exiting the authentication process.

- CERT—the public key certificate to be used by the SSL HTTP server.
- KEY—the private key to be used by the SSL HTTP server.

We modified the clients' standard installation scripts to copy this configuration file into the correct locations during client installation, so that the configuration parsing code can locate it at runtime.

8 Performance Measurements

Federated Keystone has been implemented as open source code in Python, and is freely available from the OpenStack Git repository (https://github.com/kwss/keystone.git). It was originally planned that this code would only be an optional add-on to the next OpenStack release, called Havana, due in September 2013. However, it is now likely that federation will also be an integral part of the core OpenStack release due in April 2014. Two different FIM protocols are currently implemented: SAML and Keystone proprietary, and the performance of each has been measured. We carried out the performance tests in order to determine a) the overhead of using federated identity management in Keystone instead of centralised Keystone authentication, and b) the overhead of each FIM component, for both protocols. In the Keystone proprietary case, we only used X.509 tokens, rather than opaque ones. We did not measure the performance of either of the federated protocols in communicating with the IdPs, or the IdPs themselves, since these are external to Keystone and will vary with both the federation protocol and IdP implementation that are used. Comparing and contrasting the performance of different federation protocols and different protocol implementations is beyond the scope of this paper.

The performance tests were carried out with Ubuntu 12.04 running on a virtual machine with an Intel(R) Xeon(R) CPU E5520 @ 2.27 GHz and 4 GB RAM. The times were measured in two ways:

1. across the network using Apache's JMeter software on the client. This measured the time taken from sending the various HTTP requests to and receiving a response from either the federated or centralised Keystone;
2. internally to the federated Keystone server, by outputting a timestamp from the system clock both before and after the various functions were called.

Each test was repeated 500 times in a session, and two sessions were run consecutively over one night. The mean and standard deviations of each session were calculated. Any results which were more than a number of times the standard deviation were discarded as anomalous outliers, and the mean and standard deviation were recalculated. When the original standard deviation was low (< 15 %) then 3 times the value was used for outliers. When it was high (>50 %) then 1 times the value was used, when it was intermediate then 2 times the value was used.

The results are shown in Table 1. The first column lists the component whose performance is being measured in the remaining columns. **Discovery** refers to trapping a user's initial request for federated authentication, accessing the Service Catalog to retrieve the list of trusted IdPs, and preparing a response (Federated) or sending the initial request from the client and receiving the response (J Meter). The difference between them is the time taken by the remaining middleware components in the Keystone pipeline and the network overhead to carry the request and response messages. **Get IdP Request** refers to trapping the users request that has selected an IdP, calling the FIM protocol specific module to create an authentication request to the IdP and preparing the response (Federated) or sending the request from the client and receiving the response (J Meter). **Validate Response** refers to trapping the user's request containing the IdP's response, passing it to the FIM protocol specific module for validation, obtaining the set of identity attributes from it, and validating them against the Attribute Issuing Policy. In the J Meter column this is the time taken from submitting the IdP's response up to receiving the unscoped token along with the set of projects and domains. **User Provisioning** refers to the process of deleting any existing expired user entries, then either creating a new temporary user entry, or updating an existing one by extending its validity time. **Map Attributes** refers to the process of mapping the user's identity attributes into OpenStack domains, projects and roles. **Update User Attributes** refers to the process of updating the user's existing OpenStack roles etc. to match the current set, in case the user's entry already existed. **Get Unscoped Token** measures the performance of the pre-existing Keystone code which creates a token and stores it in the database.

Table 1 Time (ms) for each component of federated identity management in Keystone

Component	SAML				Keystone Proprietary				Centralised	
	Keystone Server		J Meter		Keystone Server		J Meter		Keystone Server	J Meter
	Run 1	Run 2	Run 1	Run 2	Run 1	Run 2	Run 1	Run 2	Run 1	Run 1
Discovery	2.03 ± 0.18	2.03 ± 0.17	5.48 ± 0.61	5.47 ± 0.53	2.0 ± 0.0	2.02 ± 0.17	6.16 ± 0.40	6.25 ± 0.5	↑	↑
Get IdP Request	9.15 ± 0.56	9.14 ± 0.43	13.2 ± 0.8	13.2 ± 0.6	6.04 ± 0.24	6.09 ± 0.29	9.78 ± 0.65	10.0 ± 0.56	|	|
Validate Response	72.3 ± 2.9	73.1 ± 1.8	↑	↑	28.5 ± 0.9	29.2 ± 1.3	↑	↑	|	|
User Provisioning	372 ± 35	373 ± 33	|	|	12.3 ± 9.7	10.7 ± 7.7	|	|	|	|
Map Attributes	69.6 ± 2.9	70.0 ± 2.0	|	|	78.0 ± 2.1	79.6 ± 3.3	|	|	193 ± 9	202 ± 10
Update User Attributes	94.2 ± 5.3	101 ± 7.5	972 ± 50	984 ± 52	68.8 ± 7.0	72.7 ± 8.1	555 ± 46	558 ± 50	|	|
Get Unscoped Token	337 ± 7	338 ± 9	|	|	339 ± 11	338 ± 8	|	|	|	|
Get Projects	8.20 ± 0.44	8.44 ± 0.50	↓	↓	4.36 ± 0.48	4.69 ± 0.69	↓	↓	6.13 ± 0.47	13.7 ± 0.8
Set User Domain	26.5 ± 2.7	26.6 ± 1.9	80.0 ± 5.8	81.0 ± 5.8	23.5 ± 0.9	24.3 ± 1.49	75.4 ± 7.0	77.1 ± 6.8	Not measured	Not measured

This provides a baseline to compare our federated code against. **Get Projects** is similarly pre-existing Keystone code that fetches a user's projects from the database, and provides another baseline measurement to compare against. **Set User Domain** (Keystone server) measures the time taken to check and modify the user's domain if necessary, then forward the request on to the core Keystone code, so that it can subsequently process the unscoped token and project ID or domain ID, create a scoped token for the user, and retrieve the user's authorised services and endpoints from the service catalog. In the J Meter case it is the time taken to submit an unscoped token and domain or project ID from the client and obtain a scoped token and authorised services in response. So the difference between these two sets of measurements is the network overhead.

In the centralised tests, we measure the time taken for the user's initial request for an unscoped token (by passing in a username and password) to be processed and the token returned. The existing centralised Keystone code does not return the list of projects, so the client has to immediately make a second call to Keystone to get the list of projects. This seems to be an inefficient way of working, so in the federated implementation we have combined the two processes together by automatically returning the list of projects along with the unscoped token, in order to save the client from having to make a second call.

Analysing the results we can see that **Discovery** is very fast (2 ms) and is independent of the federation protocol being used, as expected. The overhead of the other pipeline components and the network (4 ms) is double that of the Discovery module. **Get IdP Request** on the other hand is also fast (6–9 ms) but is dependent upon the particular federation protocol in use. SAML takes approximately 50 % longer than Keystone proprietary since it has to create and sign an XML message. The latter merely has to tell the client the location of the remote Keystone IdP, since Keystone clients already know how to make authentication requests to Keystone servers. The pipeline and network overhead remain at 4 ms. **Validate Response** is also protocol dependent, but takes significantly longer than getting the IdP request (8 times and 5 times for SAML and Keystone respectively). It takes over twice as long to validate a signed SAML XML message as a signed X.509 Keystone proprietary token, even though both are validating digital signatures. This could be due to several reasons. Firstly SAML messages are encoded in XML whereas Keystone's are in native JSON, so parsing could take longer. Secondly the SAML message contains many more attributes than the Keystone token so there is significantly more Attribute Issuing Policy checking to do. Finally the code we used for Keystone response validation is already part of the OpenStack code base, so has been quality assured by the OpenStack community, whereas the SAML code was written by ourselves and has not yet had such scrutiny. **User Provisioning** is fundamentally different in the two cases, since the Keystone proprietary IdP always returns the same User ID, hence the user entry in the federated Keystone server only needs to have its validity time updated, whereas SAML returns a random ID each time (for privacy reasons) hence a new user entry has to be created for each test. If one compares the J Meter columns for SAML and Keystone proprietary one will see that the increase in SAML (555 ms to 972 ms) is mostly accounted for by the user provisioning overhead. **Map Attributes** performed different attribute mappings for each protocol. There were three mappings for the Keystone user, but only two for the SAML user. **Update User Attributes** only had to do a comparison for Keystone proprietary users, and found that the stored attributes were equal to the current set, as the same user ID is used in each test. Consequently no database update was needed. SAML users, in comparison, had different user IDs in each test, so the code had to do a database update as none of the attributes were stored in the user's entry. **Get Unscoped Token** has identical results for both federation protocols as expected. The anomalous result is for the centralised Keystone server, which performs much faster than both of the federated cases, even though the same code is being executed (200 ms vs 338 ms). We experimented with many different performance tests to try to account for the difference. We modified the validity time of centralised users from NULL to a date longer than the token expiry time and to one shorter than the token expiry; we modified the username of a centralised user to be the same number of characters as the generated usernames for federation; and we modified the username and userID to be the same for the centralised user. But none of these made any difference to the performance gap. However, when we used a 28 character password in the centralised tests, instead of

the original 9 character one, the performance deteriorated to be identical to that of the federated tests. We tracked down this poor performance behaviour to the authenticate function, which is existing Keystone core code which authenticates the user prior to issuing the unscoped token (keystone.auth.plugins.password called from keystone.auth.controllers). It seems hard to believe that simply increasing a password by 19 characters would induce a performance penalty of over 100 ms, but our experiments proved this to be the case. We have reported this to the Keystone developers. **Get Projects** had two projects to retrieve for SAML but only one project for Keystone, so this explains the higher figure. In the centralised case, even though only one project was being retrieved, a new client request has to be received and processed in addition to actually getting the project from the database, whereas in the federated case this is done automatically as part of returning an unscoped token. **Set User Domain** checks if the user's chosen project or domain matches the domain stored in Keystone's database entry for the user. When a user is auto-provisioned, she is initially put in the default domain. But she may choose another domain, or a project that is not associated with the default domain, in which case her entry will need to be updated. The authorised user/project/role assignments are held in a metadata table, one row for each role the user has in a project, whilst the authorised user/domain assignments are held in another metadata table. Finally the project/domain associations are held in the projects table. Since the SAML tests assert two different projects for the user, whilst the Keystone tests only assert one project, we think the different database sizes account for the slightly longer time being taken by the SAML tests. The large network overhead (∼50ms) is due to submitting an unscoped token and obtaining a scoped token and set of authorised services.

One can see from the federated results that any component which simply reads from the backend Keystone SQL database, or updates a field, is relatively fast (i.e. the majority of components) whilst those that create a new database record are relatively slow (User Provisioning and Get Unscoped Token which creates a token and stores it). We were pleased that Validate Response for SAML and Keystone proprietary, which cryptographically checks the digital signature of the (XML) response message, were comparatively fast compared to the database updates, given the fact that cryptographic operations are usually regarded as being notoriously slow.

Since User Provisioning is the most time consuming of all the federated processes, we wanted to determine how each of its subcomponents performed so we measured the time taken for each internal component to execute. Table 2 shows the results for this.

We engineered these tests as follows. We modified a set of the original 500 SAML performance tests to ensure that each test had a user with a different unique ID and an assertion validity time of 24 hours, so that a new user entry was created in each test that would not expire. This gave us the execution time for creating a new user. We then modified the assertion validity time to 2 secs, and paused 1.5 secs between each test, so that each user entry that was created in the previous test had expired when the next test was run. We then measured the execution time for the code that checked for and removed expired users. We ignored the first measurement since there was no user to remove in the first test. This gave us the second set of results in the table. The final result in the table was measured by re-running the Keystone proprietary tests, because in these tests the same user ID is used each time, and we set the validity time to 24 hours. We simply measured the time taken by the code to update the validity time of the user's entry. We excluded the first test from the results as the user entry was not updated.

We conclude that the addition of federated access to Keystone performs reasonably well, given the fact that authentication is being performed remotely, and users are automatically provisioned in the system. The actual performance is highly dependent upon the federation protocol that is used, the length of the user password that is stored, and whether the same user is given the same unique ID or not each time she authenticates. Compared to the existing centralised authentication mechanism, the current design of federated authentication may take between 2.75 and 5 times as long.

Table 2 Time (ms) for each component of user provisioning

Sub Component	Mean ± Std. Dev. (ms)
Creating a new user	331.9 ± 11.2
Removing one expired user	1.60 ± 0.50
Updating an existing user	6.95 ± 2.60

9 Limitations and Conclusion

There are a number of limitations in both the current design and the implementation as described below.

9.1 Design Limitations

Most existing federated identity management systems which use usernames and passwords for authentication (i.e. the vast majority) are open to phishing attacks. In this attack, a user is tricked into contacting a malicious cloud service provider, for example, by the offer of a free or very low cost service. When the user contacts the service, he is redirected to a fake identity provider which displays a web page that appears to be that of his real identity provider. Consequently the user unwittingly enters his username and password, and therefore has been successfully phished. The solution to this is either for the IdP to use a zero-knowledge proof authentication mechanism, such as public key cryptography, so that if the user is phished, the attacker gains zero knowledge, or to use a protocol sequence that does not involve the client being redirected to the IdP by the service provider. One such protocol sequence is the IdP initiated sequence, typically used by portals, where the user contacts the IdP first, rather than the SP. However, the disadvantage of this method is that the IdP must have a list of SPs that the user is allowed to contact, so that it can redirect the user there. This is why this solution is typically used by portals. An alternative is to make use of an intelligent client that knows how to contact the user's IdPs directly, without being redirected there by the service provider. We would like to implement and experiment with this intelligent client in a future project.

The current design suffers a significant performance penalty by automatically provisioning temporary users and automatically deleting them once they have expired. Over one third of the time spent on federated login is consumed in this part of the code. If we were to create permanent users rather than temporary users, then we could remove this login overhead (except for the very first time the user logs into Keystone). The downside of this is twofold. Firstly, Keystone's user database would grow over time and would not be cleaned without administrator intervention. However, a housekeeping tool could be created to periodically purge all old entries of users who had not logged in for longer than a certain period of time. Secondly, this does not solve the problem for federation protocols like SAML that allow for random IDs to be issued for users instead of persistent ones. However, the SAML protocol does allow the SP to require the SAML IdP to issue a persistent ID, and the Keystone SAML module could be configured to mandate this option.

The current design has no way of controlling the level of authentication trust that Keystone has in an IdP. We could easily add this by enhancing the service catalog to hold the maximum Level of Assurance (LoA) that the Keystone administrator has in each IdP. Then when the authentication statement is received from an IdP, the implementation could check that the asserted LoA is less than or equal to the stored LoA, and if not, either downgrade or reject the assertion. Once the LoA starts to be implemented and regulary sent by IdPs, then this enhancement will be quite easy to add to Keystone.

The current Attribute Issuing Policy (identification trust) only controls the attribute types and not the types and values that an IdP is trusted to issue. We currently do not have sufficient practical experience to know if this might be a problem in the future or not. But we envisage that if an attribute whose values confer a range of privileges from low to very high (such as a role with values from porter to CEO), and this is issued by a set of IdPs of different LoAs, then one might wish to not accept high value identity attributes from low assurance IdPs. A future extension could be to limit the set of identity attribute values that an IdP is trusted to issue.

9.2 Implementation Limitations

The current implementation supports the SAML Web Browser SSO profile which causes the existing command line clients to launch a web browser for the authentication phase with the IdP. Consequently the user experience is neither pure command line nor pure browser based. There are two problems with this. Firstly some clients may be applications which require a pure command line or programmable interface. Secondly some users may prefer a complete browser based experience. Both of these are possible with the current design. The former requires the protocol dependent module, client and IdP to support the SAML Enhanced Client and Proxy (ECP) Profile [52], so that direct communications between the client

and IdP and SP are used rather than browser redirects. The reason we did not implement ECP in the current project, is that currently, many IdP's do not support it. The latter requires a three tier architecture, with an intermediate web server that acts as the client to Keystone and acts as the SP to the user's browser. We would like to build this web server in a future project.

The current design supports attribute aggregation, although there are no standard federation protocols that support this yet. Once attribute aggregation is supported, it will be interesting to plug in such a protocol in order to validate the attribute aggregation design.

9.3 Conclusion

In this paper we have described how OpenStack currently performs authentication and authorisation using internally stored information, and how we have extended its functionality to provide federated identity management. We have presented a user case to support the need for federated access. The existing limitations of the current design and implementation have also been described. We expect these will be addressed in due course as user demand for more fine grained and complex authorisation functionality increases.

Acknowledgments This research has been funded by the UK's EPSRC under grant ref. n° EP/J020354/1 (Sticky Policy Based Open Source Security APIs for the Cloud).

References

1. http://www.openstack.org/ (Accessed 26 Nov 2012)
2. Fielding, R.T.: Representational State Transfer (REST). Chapter 5 of Architectural Styles and the Design of Network-based Software Architectures. PhD Dissertation submitted to University of California, Irvine. Available from http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (2000)
3. Badger, L., et al.: NIST US Government Cloud Computing Technology Roadmap, vol. I, Release 1.0, SP 500-293 (2011)
4. Lee, C.: Cloud Federation—Establishing Trust, NIST Cloud Computing Forum and Workshop V. Washington DC, 5 June 2012. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/ForumVAgenda/5_NIST-Forum-V-Lee-v4.pdf (2012)
5. Greer, M., Lee, C.A.: Cloud-Based Disaster Response, NIST Joint Cloud and Big Data Workshop. Gaithersburg, MD. Jan 15, 2013. Available from http://www.nitrd.gov/nitrdgroups/images/5/5a/Disaster_Response%3B_Craig_Lee_CCWG.pdf (2002)
6. The Networking and Information Technology Research and Development (NITRD) Program. See http://www.nitrd.gov/about/about_nitrd.aspx
7. Naqvi, S., et al.: From Grids to Clouds—Shift in Security Services Architecture, CGW'09—Cracow Grid Workshop. Krakow, Poland (2009)
8. Nunez, D., et al.: Identity Management Challenges for Intercloud Applications, 1st International Workshop on Security and Trust for Applications in Virtualised Environments (STAVE 2011) (2011). doi:10.1007/10.1007/978-3-642-22365-5_24
9. Bernstein, D., Vij, D.: Intercloud Security Considerations, 2nd IEEE International conference on Cloud Computing Technology and Science, pp. 537–544 (2010)
10. Emig, C., et al.: Identity as a service—Towards a service-oriented identity management architecture. In: EUNICE'07 Proceedings of the 13th Open European Summer School and IFIP TC6.6 Conference on Dependable and Adaptable Networks and Services, pp. 1–8 (2007)
11. Rak, M., et al.: Security Issues in Cloud Federations, Chapter 10 in Achieving Federated and Self-Manageable Cloud Infrastructures. Theory and Practice, IGI-Global (2012). doi:10.4018/978-1-4666-1631-8.ch010
12. Neuman, B.C., Ts'o, T.: Kerberos: An authentication service for computer networks. IEEE Commun. **32**(9), 33–38 (1994)
13. Foster, I., et al.: A security architecture for computational Grids. In: Proceedings of the 5th ACM Conference on Computer and Communication Security, pp. 83–92. ACM (1998)
14. Housley, R., et al.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile http://www.ietf.org/rfc/rfc3280.txt (2002)
15. Tuecke, S., et al.: Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile, IETF RFC 3820. http://www.ietf.org/rfc/rfc3820.txt (2004)
16. Barton, T., Basney, J., Freeman, T., Scavo, T., Siebenlist, F., Welch, V., Ananthakrishnan, R., Baker, B., Goode, M., Keahey, K.: Identity Federation and Attribute-based Authorization through the Globus Toolkit, Shibboleth, GridShib, and MyProxy. 5th Annual PKI R&D Workshop.NIST, Gaithersburg MD. Available from http://middleware.internet2.edu/pki06/proceedings/welch-idfederation.pdf (2006)
17. OASIS: Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard (2005)
18. The Shibboleth Consortium. http://shibboleth.net
19. InCommon, http://incommon.org
20. UK Access Management Federation, see http://www.ukfederation.org.uk/
21. eduGAIN membership status, see http://www.edugain.org/technical/status.php

22. Details of the Shebangs project: http://www.rcs.manchester.ac.uk/research/shebangs
23. Goodner, M., et al.: Understanding WS-Federation. http://www.msdn.microsoft.com/enus/library/bb498017.aspx (2007)
24. OpenID Authentication 2.0 – Final. Available from http://openid.net/specs/openid-authentication-2_0.html (2007)
25. Hart, D.: The OAuth 2.0 Authorization Framework, RFC 6749 (2012)
26. Sakimura, N., et al.: OpenID Connect Standard 1.0—draft 18. Available from http://openid.net/specs/openid-connect-standard-1_0.html (2013)
27. Howlett, J., et al.: Application Bridging for Federated Access Beyond Web (ABFAB) Architecture, IETF. http://tools.ietf.org/html/draft-ietf-abfab-arch-06 (2013)
28. Rigney, C., et al.: Remote Authentication Dial In User Service (RADIUS), IETF RFC 2865 (2000)
29. Florio, L., Wierenga, K.: Eduroam, Providing Mobility for Roaming Users. Proceedings of the EUNIS 2005 Conference, Manchester (2005)
30. Open Science Grid, Virtual Organization Summary, http://myosg.grid.iu.edu/vosummary?all_vos=on&active=on&active_value=1&datasource=summary
31. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: enabling scalable virtual organizations. Int. J. High Perform. Comput. Appl. **15**(3), 200–222 (2001). doi:10.1177/109434200101500302
32. Nasser, B., Laborde, R., Benzekri, A., Barrère, F., Kamel, M.: Access control model for inter-organizational Grid virtual organizations. In: Meersman, R., Tari, Z., Herrero, P. (eds.) Proceedings of the 2005 OTM Confederated International Conference on On the Move to Meaningful Internet Systems (OTM'05), pp. 537–551. Springer, Berlin Heidelberg (2005). doi:10.1007/11575863_73
33. Cummings, J., Finholt, T., Foster, I., Kesselman, C., Lawrence, K.: Beyond Being There: A Blueprint for Advancing the Design, Development, and Evaluation of Virtual Organizations, Final report from the NSF workshops on Building Effective Virtual Organizations. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.205.6126&rep=rep1&type=pdf (2008)
34. Enabling Grids for E-SciencE (EGEE): EGEE User' Guide—VOMS Core Services. http://egee.cesnet.cz/en/voce/voms-guide.pdf (2005)
35. Coppola, M., Jégou, Y., Matthews, B., Morin, C., Prieto, L.P., Sánchez, Ó.D., Yang, E.Y., Yu, H.: Virtual organization support within a Grid-wide operating system. IEEE Internet Computing **12**(2), 20–28 (2008). doi:10.1109/MIC.2008.47
36. The European Grid Infrastructure (EGI), http://operations-portal.egi.eu/vo
37. Scott Cantor. NativeSPAttributeResolver, available from https://wiki.shibboleth.net/confluence/display/SHIB2/NativeSPAttributeResolver
38. Chadwick, D.W., Inman, G.: Attribute Aggregation in Federated Identity Management. IEEE Computer, pp. 46–53 (2009)
39. Watt, J., Sinnott, R.O., Inman, G., Chadwick, D.: Federated Authentication and Authorisation in the Social Science Domain. Sixth International Conference on Availability. Reliability and Security (ARES), pp. 541–548 (2011)
40. Chadwick, D.W., Inman, G.: The Trusted Attribute Aggregation Service (TAAS)—Providing an attribute aggregation layer for federated identity management. In: Proceedings of the Eight International Conference on Availability, Reliability and Security (ARES 2013), Regensberg (2013)
41. Rochwerger, B., et al.: The reservoir model and architecture for open federated cloud computing. IBM J. Res. Dev. **53**(4), 4:1–4:11 (2009)
42. Celesti, A., et al.: Security and Cloud Computing: InterCloud Identity Management Infrastructure. In: Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, pp. 263–265 (2010)
43. Ferrari, T.: On behalf of P. Solagna/EGI.eu, EGI Services for Distributed e-Infrastructure Access, ISGC 2013, Taipei. http://indico3.twgrid.org/indico/getFile.py/access?contribId=95&sessionId=20&resId=0&materialId=slides&confId=370 (2013)
44. Drescher, M., Collier, I.: EGI Federated Clouds: Task Force & Infrastructure, OGF37, Cloud Federation BoF (2013)
45. The EU Contrail Project, http://contrail-project.eu
46. The National Strategy for Trusted Identity in Cyberspace, http://www.nist.gov/nstic/index.html
47. Hogben, G.: Privacy, Security and Identity in the Cloud, http://www.enisa.europa.eu/activities/Resilience-and-CIIP/cloud-computing/Cloud_Identity_Hogben.pdf
48. Cloud Security Alliance, http://cloudsecurityalliance.org
49. The Kantara Initiative. http://kantarainitiative.org
50. Camenisch, J., et al.: H2.1—ABC4Trust Architecture for Developers. https://abc4trust.eu/download/ABC4Trust-H2.1-Architecture-for-Developers.pdf (2012)
51. OASIS, Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard (2005)
52. OASIS, SAML Enhanced Client or Proxy (ECP) Profile. Version 2.0. Committee Specification. Available from http://docs.oasis-open.org/security/saml/Post2.0/saml-ecp/v2.0/cs01/saml-ecp-v2.0-cs01.pdf (2013)