Journal of Universal Computer Science, vol. 20, no. 3 (2014), 385-405
submitted: 24/3/13, accepted: 4/2/14, appeared: 1/3/14

Automatic Authentication to Cloud-Based Services

Mircea Boris Vleju
(Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC)
Hagenberg im Mühlkreis, Austria
b.vleju@cdcc.faw.jku.at)

Abstract: We describe the concept of automatic authentication for cloud-based services via the use of a client-centric solution for small and medium enterprises (SMEs). In previous work we have introduced the Identity Management Machine (IdMM) which is designed to handle the interaction between a client's identity directory and various cloud identity management systems. We now further refine this machine by describing its interaction with various cloud authentication systems. The IdMM is designed to aid SMEs in their adoption or migration to cloud-based services. The system allows SMEs to store its confidential data on-premise, enhancing the client's control over the data. We further enhance the privacy related aspects of a client-to-cloud interaction via the introduction of obfuscated and partially obfuscated identities which allow SMEs to also choose the type of data being sent to a cloud service. Since the IdMM is a single sign-on system capable of automatic authentication the risk of phishing or other social engineering attacks is reduced as an individual user may not be aware of his or her credentials for a given cloud service.

Key Words: Abstract State Machine, Automatic Authentication, Client Centric, Cloud Computing, Identity Management, Small and Medium Enterprises
Category: D.2.10, F.1, H.4, H.5.3, K.6.5

1 Introduction

Our research within the Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC) deals with the client-side aspects of identity management in cloud computing [1]. The adoption of cloud-based services offers many advantages for small and medium enterprises. However, a cloud-based approach also entails certain disadvantages. The authors of [Alpár et al. 2011], [Brunette and Mogull 2009],[Cloud Security Alliance 2010] and [Schewe et al. 2011] outline some of these disadvantages: loss of control, contracting issues, provider lock-in and other security and privacy issues. Such issues imply an extra level of trust between a client and a cloud provider. With respect to identity management, a loss of control implies that the client must trust the cloud provider with sometimes critical or important identity information (such as credit card information). A potential client would prefer such data to be stored on-premise and only be offered to a service on demand. The vendor lock-in issue might be mitigated by the

[1] We define the term *client* as being a small or medium enterprise (SME) that contracts and uses any cloud service. Similarly we refer to a *user* as an identity within the SME using a cloud-based service.

adoption of services across multiple providers. In this scenario a problem arises in maintaining identity data across the providers. Changing a client-side property, such as the user's address for example, entails changing this property on each individual service (a time consuming task especially if the service provider does not adopt open standards). Our research is focused on providing a client-centric identity meta-system which allows a client to maintain a private identity directory while offering individual users automatic authentication to cloud-based services via a single sign-on, privacy enhanced service.

In [Vleju 2012a] we have introduced the Identity Management Machine (IdMM) representing a client-centric, single sign-on tool for small and medium enterprises that want to adopt or migrate to cloud-based services. This concept has been further refined in [Vleju 2012b] where we described the architecture of the IdMM. As mentioned in [Vleju 2012b], the IdMM is composed of six agents [see Fig. 1]: the core agent (comprising the rules described in [Vleju 2012a]), the client agent (managing the interaction with the client's directory), the cloud agent (used for the interaction with a cloud service), the user agent (handling the interaction with a user), the protocol agent (used for protocol-based authentication) and the provisioning agent (managing user provisioning, password resets and user de-provisioning). Apart from the core agent, each agent is defined by further refinement of the abstract functions presented in [Vleju 2012a]. This process is still an ongoing task, with the provisioning and protocol agents still left at an abstract level. In [Vleju 2012c] we have described the $\text{IdMM}_{\text{Client}}$ agent by further refinement of the client-side abstract functions. We also gave an example of the $\text{IdMM}_{\text{Client}}$'s interaction with an ApacheDS LDAP directory.

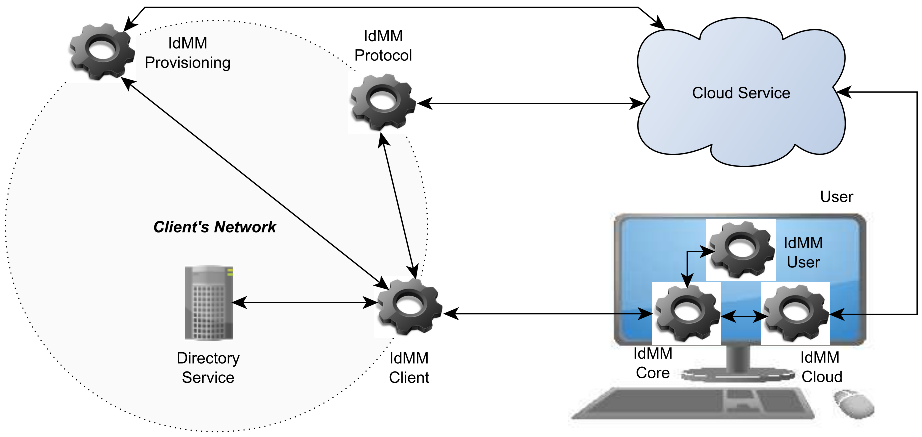

Figure 1: IdMM Architecture

The purpose of this paper is to introduce the rules relating to the cloud and user agents of the IdMM. With these agents specified we can then exemplify the IdMM's automatic authentication capabilities. Since this process is dependent on the specification of the $\text{IdMM}_{\text{Core}}$ and $\text{IdMM}_{\text{Client}}$ agents we will first describe these two agents. In [Section 3] we describe the main concepts of the Identity Management Machine. [Section 4] and [Section 5] describe the $\text{IdMM}_{\text{Core}}$ and $\text{IdMM}_{\text{Client}}$ agents. More information about the key concepts and these two agents can be found in [Vleju 2012a;b;c]. The $\text{IdMM}_{\text{Cloud}}$ agent is described in [Section 6] with the $\text{IdMM}_{\text{User}}$ agent described in [Section 7]. Finally, we describe a proof of concept implementation in [Section 8] and outline our conclusions and further work on the topic in [Section 9].

2 Related Work

As mentioned in [Vleju 2012a] there have been several attempts to define a client-centric approach to identity management. The authors of [Ahn et al. 2009] describe a privacy enhanced user-centric identity management system allowing users to select their credentials when responding to authentication requests. It introduces "*a category-based privacy preference management for user-centric identity management*" using a CardSpace compatible selector for Java and extended privacy utility functions for P3PLite and PREP languages. The advantage of such a system is that it allows users to select the specific attributes that will eventually be sent to a relying party. Such a system works well for enhancing privacy, however it fails to address the extra overhead inflicted on the user. As [Schechter et al. 2007] shows a typical user would tend to ignore obvious security and privacy indicators. For composite services, the authors of [Zhang and Chen 2010] describe a universal identity management model focused on anonymous credentials. The model "*provides the delegation of anonymous credentials and combines identity meta-system to support easy-to-use, consistent experience and transparent security*".

From a client-centric perspective, Microsoft introduced an identity management framework (CardSpace) aimed at reducing the reliance on passwords for Internet user authentication while improving the privacy of information. The identity meta-system, introduced with Windows Vista and Internet Explorer 7, makes use of an "open" XML based framework allowing portability to other browsers via customized plugins. However CardSpace does suffer from some known privacy and security issues, mentioned in [Alrodhan and Mitchell 2007, Oppliger et al. 2007]. The concept of a client-centric identity meta-system is thoroughly defined in [Cameron et al. 2008]. The framework proposed here is used for the protection of privacy and the avoidance of unnecessary propagation of identity information while at the same time facilitating exchange of specific information needed by Internet systems to personalize and control access to services. By defining abstract services the framework facilitates the interoperation of the different meta-system components.

Passwords managers can, in our opinion, reside within the topic of automatic authentication for individual users. Projects such as KeePass [2] and LastPass [3] do offer some similar functionalities to the IdMM. However both fall short in two key criteria: neither is truly automatic (since some input is required by the user upon authentication to a service) and while both work well with individual users they cannot or should not be used by SMEs (since Keepass is designed to be user-centric and LastPass does not offer a client control over where the data is stored).

3 The Identity Management Machine

Identity Management Machine (IdMM), first presented in [Vleju 2012a], is a privacy enhanced client-centric identity meta-system based on the concept of abstract state machines [Börger and Stärk 2003]. The system provides a '*proxy*' between the client and cloud-based identity management solutions, '*translating*' the protocols used by the client to manage identities to a set of protocols used by cloud providers. The IdMM can then authenticate a user to a given cloud service as well as manage any private identity-related data stored on the cloud.

3.1 Interaction Scenarios

The IdMM makes an abstraction of the protocols used by both the client and cloud provider for their identity management systems via the use of the abstract functions presented in paper [Vleju 2012a]. Such functions leave the organizational and implementation aspects of the identity management systems directly into the hands of the end parties. To describe these functions we must first consider the interaction between a client and a cloud provider with respect to identity management, authentication and authorization. We consider three distinct cases of client-to-cloud interaction: the direct case, the obfuscated case and the protocol based case.

3.1.1 Direct Client-to-Cloud Interaction

As showed by the authors of [Alpár et al. 2011], [Brunette and Mogull 2009] and [Dhamija and Dusseault 2008] one of the greatest issues surrounding identity management for cloud providers is the need of the cloud provider to control

[2] `http://www.keepass.info/`
[3] `http://www.lastpass.com/`

the customer experience. Many providers make use of their own custom designed identity systems to which a client must subscribe. This means that the client has no choice but to use the cloud provider's identity system. While this may be an inconvenience from a privacy point of view, the real problem lies in managing the client's information across multiple providers. A simple change, such as changing a user's address, entails changing the value on every single provider the client uses. To combat this problem we view the interaction between the client and a cloud service as a one-to-many mapping. Any change made on the client-side must also be made on the cloud via the synchronization of attributes. Concurrently, any changes made by the provider (such as the addition, replacement or removal of an attribute) must be reflected in the client's directory system.

3.1.2 Obfuscated Client-to-Cloud Interaction

While the direct client-to-cloud interaction allows for an efficient use of cloud services it does suffer from a lack of privacy. Since all information about a client is stored on the provider's infrastructure there is an increased risk that through data leakage or unauthorized access that information could be fall into the wrong hands. We mitigate this threat by introducing the concept of obfuscated identities [4].

We consider an identity as being a *real identity* if the information contained corresponds to the identity's owner and is visible to any external entity (as opposed to the other scenarios where the raw information may be obfuscated to an external entity). Such identities can be used, for example, in on-line stores, where the provider needs to have real information to work with. In such cases the provider needs to know the full name of the user, the address and possibly credit card data. Using any kind of obfuscation in such a case could impede the provider from handling the order request.

As opposed to a real identity, an *obfuscated identity* has its information obfuscated. Depending on the method of obfuscation (some methods are presented in [Bakken et al. 2004]) the information is either undecipherable or can only be deciphered by the owner of the identity. Any free file storage service can be used as an example where obfuscated identities are recommended. In such services, the provider does not need to know the user's full name, height or address. Therefore such obfuscated identities can be successfully used. If the obfuscation method makes the information undecipherable, then the corresponding identity can be viewed as anonymous (as described by the authors of [Zhang and Chen 2010]).

A third kind of identity considered is the *partially obfuscated identity.* The information contained by such identities is a mix of real/visible attributes as well

[4] For the purpose of this paper we consider the definition of an identity as explained in [The Open Group Identity Management Work Area 2004]

as obfuscated ones. If we extend the file storage example by adding the condition that any user must be over 18 years old in order to use the service then the age of the user must not be obfuscated. As such, while the rest of the information can remain obfuscated, the age will contain real user data.

The usage of obfuscated and partially obfuscated identities is dependent on the cloud service. As mentioned, some services do require real identities. Even if the service can be used with obfuscated identities we leave the matter in the hands of the client. The client can choose whether or not to use obfuscation in such cases. For partially obfuscated identities we also allow the client to choose what real data attributes are sent to a service.

3.1.3 Protocol-Based Client-to-Cloud Interaction

In recent years there has been a drive to improve interoperability between cloud providers mostly to prevent vendor lock-in. From an identity management perspective the result has been the adoption of some open-based protocols to facilitate both cloud interoperability as well as identity access management. Protocols such as OpenID or OpenAuth represent an important tool for a client-centric identity management system. They allow the provider to focus on the requirements of the service while allowing access via the standard implementation of these protocols. From the client's perspective, such protocols allow for an easier integration across multiple cloud providers. It must be noted however that such protocols do suffer from a variety of security and privacy issues, as described in [Dhamija and Dusseault 2008] and [Prodromou 2007].

3.2 Architecture

The IdMM represents a client-centric tool used for authentication, authorization and, when needed, attributes synchronization to cloud services. Our research thus far has focused only on the authentication and attributes synchronization part with the authorization planned as future work. The rules for the automatic authentication are described in [Vleju 2012a]. To describe these rules we make use of abstract functions. These functions can be divided into four categories: client-side functions, cloud-based functions, system functions and user-based functions [see Fig. 2]. From these functions we can derive the general architecture of the system [see Fig. 1]. The system is composed of seven agents: $\mathrm{IdMM}_{\mathrm{Core}}$, $\mathrm{IdMM}_{\mathrm{Client}}$, $\mathrm{IdMM}_{\mathrm{Cloud}}$, $\mathrm{IdMM}_{\mathrm{User}}$, $\mathrm{IdMM}_{\mathrm{Protocol}}$ and $\mathrm{IdMM}_{\mathrm{Provisioning}}$. Apart from the $\mathrm{IdMM}_{\mathrm{Protocol}}$ and $\mathrm{IdMM}_{\mathrm{Provisioning}}$ agents which we will detail each agent in the following sections. The full specification of the protocol and provisioning agents is still an ongoing task.

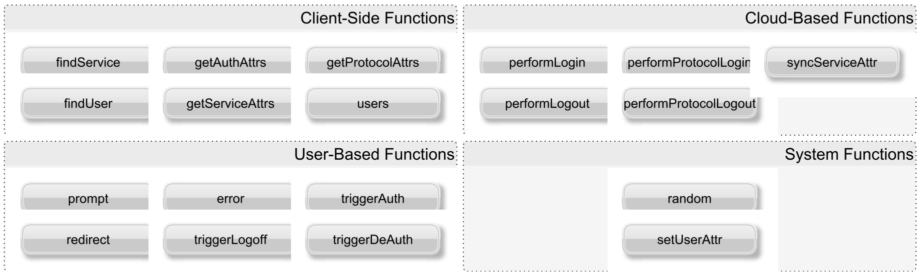

Figure 2: Abstract Functions

3.3 Notational Conventions

For a quick reference we list here some frequently used notations, in particular for list operations:

$T*$ denotes a list of elements of type T.

$[e_1, \ldots, e_n]$ denotes a list containing the elements $e_1, \ldots, e_n$.

$[]$ denotes an empty list. $length(l)$ returns the number of elements in the list l.

$l_1 \cdot l_2$ denotes the concatenation of the lists l_1 and l_2.

$e \in l$ denotes that the list l contains the element e.

$e \notin l$ denotes that the list l does not contain the element e.

$l_1 \subset l_2$ denotes that all the elements in l_1 exist in l_2.

$split(l, n)$ splits off the last n elements from the list l returning a pair (l', n') such that $l' \cdot n' = l \wedge length(n') = n$.

$l_1 - l_2$ removes from l_1 any elements that exist in both l_1 and l_2.

$random(l)$ returns a random element from the list l.

4 The IdMM$_{\mathbf{Core}}$ Agent

The core agent for our client-centric solution is described in [Vleju 2012a]. The resulting ASM has 9 states [see Fig. 3]. The initial state of the IdMM is *UserLogin*. While in this state the user is prompted to input his or her credentials and is subsequently authenticated to the machine (if the user cannot be authenticated the machine halts with the appropriate error). Upon a successful authentication the machine then waits for further input from the user. This input is represented by one of three events: a user logout request, a service authentication request or a service de-authentication request. Upon a user logout request the state is set to *UserLogout*. This will log the user out of the machine set the state to *UserLogoutService* which then logs the user out of every single service he or she was connected to and halts the execution of the machine. The termination of the machine occurs in the *Halt* state. If an error exists the appropriate message will be displayed to the user.

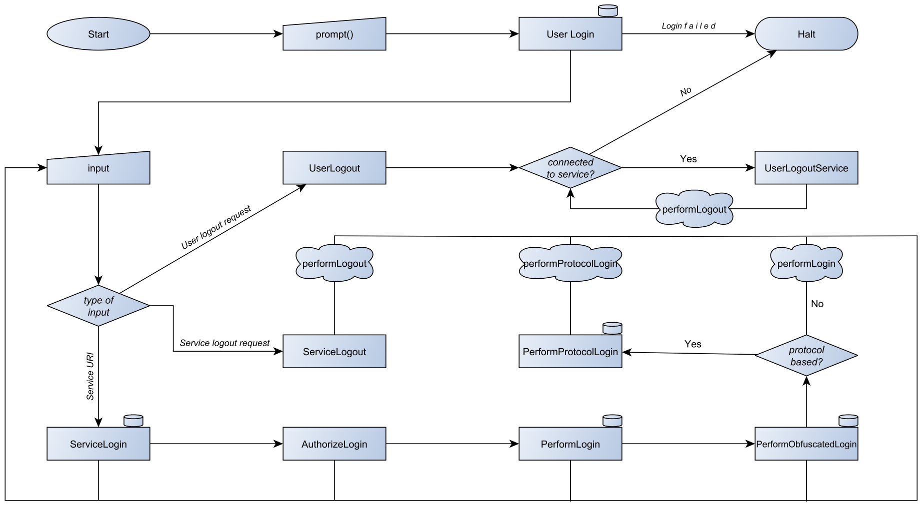

Figure 3: IdMM Flow

When the user wants to use a specific service he or she will specify the service's URI. The event triggered by this action will set the state to *ServiceLogin*. In the *ServiceLogin* state the machine attempts to find the matching service given the URI. If no services can be found an error message is triggered. If a service is found the state will be set to *AuthorizeLogin*. In case the user is already connected to the service he or she will be redirected to the given URI. The authorization is done by the *AuthorizeLogin* state. If the user has the appropriate access rights the state is set to *PerformLogin* where the machine checks the type of identity required and sets the state to *PerformObfuscatedLogin*. If the service supports protocol-based authentication then *PerformObfuscatedLogin* will switch to *PerformProtocolLogin*, which will perform the authentication based on a given protocol. In case the service does not support protocol-based authentication the machine will perform the authentication with the given identity. The machine switches to the *ServiceLogout* state when a log-out event is triggered. In this state the IdMM searches for the matching service and performs the log-out.

To describe the rules of the IdMM we have introduced several types and data structures [see Fig. 4]. We defined the type *Access* which will later be used for access control. Since the addition of access control management is planned as future work the values supported by the type are not yet relevant. To identify whether an identity is real, obfuscated or partially obfuscated we use the type *IdentityType*. The data structure *Attr* is a name-value pair representing an attribute. To describe a protocol we use the data structure *Protocol*. The data structure *User* is used to store any information about an identity (unique id, the attributes, the type of the identity and an access control list for each service). Similarly, the data structure *Service* is used for storing information regarding services. Any service has a unique URI, an access control list and a marker to identify the type of identities it supports. We surmise that each cloud service has an authentication service. Since the authentication service is a cloud-based service itself we include it as being part of the data structure *Service*. A *service* is considered as being an authentication service if the value of *authService* is $\emptyset$. The lists *attrs* and *authAttrs* represent the names of the attributes and authentication attributes used by the service. This list will be used as a parameter in the functions *getServiceAttrs* and *getAuthAttrs* [see Fig. 2]. If an authentication service supports standard protocols then these protocols are stored in the list *protocols*.

$$
\begin{array}{l}
\mathbf{type}\ \ Access\ =\ NoAccess \mid Read \mid Write \mid Execute \\
\mathbf{type}\ \ IdentityType\ =\ Real \mid Partial \mid Obfuscated \\
\\
\mathbf{data}\ \ Attr\ =\ (name,\ value) \\
\quad name \in String\ ,\ \ value \in String \mid \mathbb{R} \mid Attr \mid Attr* \\
\\
\mathbf{data}\ \ Protocol\ =\ (name, protocolAttrs)\ , \\
\quad name \in String\ ,\ \ protocolAttrs \in String* \\
\\
\mathbf{data}\ \ Service\ =\ (uri,\ attrs,\ acl,\ authService,\ authAttr,\ idType,\ protocols) \\
\quad uri \in String\ ,\ \ attrs \in String*\ ,\ \ acl \in Access*\ , \\
\quad authService \in Service \mid \emptyset\ ,\ \ authAttr \in String* \mid []\ , \\
\quad idType \in IdentityType\ ,\ \ protocols \in Protocol* \\
\\
\mathbf{data}\ \ User\ =\ (id,\ attrs,\ sacl,\ idType)\ , \\
\quad id \in String\ ,\ \ attrs \in Attr*\ ,\ \ sacl \in Map\,(Service, Access*)\ , \\
\quad idType \in IdentityType
\end{array}
$$

Figure 4: IdMM Types and Data Structures

As previously stated we make use of abstract functions to describe the IdMM [see Fig. 2]. The client-side functions, cloud-based functions and user-based functions will be further refined by describing the $\text{IdMM}_{\text{Client}}$, $\text{IdMM}_{\text{Cloud}}$, $\text{IdMM}_{\text{User}}$ and $\text{IdMM}_{\text{Protocol}}$ agents. To ease this process we keep the underlying communication layers abstract [5]. The system functions include two important functions which we must detail: *random* and *setUserAttr*. In order to handle an obfuscated identity we use the function *users* to select a list of the obfuscated identities contained in the client's directory. We would then choose a random identity from this list and use it to authenticate to the required cloud service. At present we

[5] In [Vleju 2013] for example, we have included the $\text{IdMM}_{\text{Client}}$ agent as part of a Tomcat server. The communication with the $\text{IdMM}_{\text{Core}}$ agent is done via the Google Web Toolkit RPC framework.

propose that the function *random* should be restricted to only this requirement. More research can be conducted in this topic to further enhance privacy. For instance, we might choose not to include identities that have been previously been used with the give service.

If the service requires a partially obfuscated identity we use the function *setUserAttr* to replace some of the obfuscated values in the identity with real values that correspond to the current user [see Fig. 5] [6]. To achieve this we assume that the result of the function *getServiceAttrs* will contain an attribute named *"RequiredUserAttrs"*. The value of this attribute will be a list representing the names of the real attributes that must be inserted.

```
Boolean setUserAttr(User obf, Service s, User current)=
  let result = false
  let attrs = getServiceAttrs(s, current)
  let requiredAttrs = getAttribute('RequiredUserAttrs', attrs , s.uri)
  if requiredAttrs≠ ∅ and
     requiredAttrs.value ∈ String* then
    result:=true
    forall attrName∈requiredAttrs.value do
      let attr_current = getAttribute(attrName, attrs, s.uri)
      let attr_obf = getAttribute(attrName, obf.attrs, s.uri)
      if attr_current≠ ∅ and attr_obf≠ ∅ then
       attr_obf.value:=attr_current.value
      else
        result:=false
  return result;
```

Figure 5: System Function setUserAttr

5 The IdMM$_{\text{Client}}$ Agent

The refinement of the client-side functions is described in [Vleju 2012c]. To allow a seamless adoption of the IdMM by a potential client we make an abstraction with respect to the client's directory. The refinement of the client-side functions yielded an interface (*client interaction interface*) of 20 directory dependent functions [see Fig. 6]. Each one of these functions depends on the type and the protocols used by the client's directory. For example, if the client uses an LDAP-based directory than the implementation of the client interaction interface must contain a minimum implementation of the LDAP protocol such that the IdMM$_{\text{Client}}$ agent can interact with the directory.

In addition to the client interaction interface the refinement of the client-side functions also yielded a list of parameters for each of the submachines. These

[6] The function *getAttribute* is described in [Vleju 2012c]. It returns an attribute given its name from a list provided as parameter. If no attribute is found the function returns ∅

parameters will include information about the client's directory (location, authentication mechanism, credentials) as well as information about the structure of the client's directory [see Fig. 6]. In [Vleju 2012c] we illustrated how to implement a client interaction interface for an ApacheDS server. We used the Novell LDAP Classes for Java API to facilitate the interaction with the ApacheDS server. The parameters were stored in an XML file. In [Vleju 2013] we switched from an XML file to a *.properties* file.

The $\mathrm{IdMM}_{\mathrm{Client}}$ agent is responsible with the interaction with the client's directory service. Its purpose is to retrieve any relevant information from the directory service and convert it to the IdMM's data structures. To achieve this we further refined the client-side functions presented in [Fig. 2]. Apart for the *getAuthAttrs* function, the refinement involves introducing a new submachine for each client-side function.

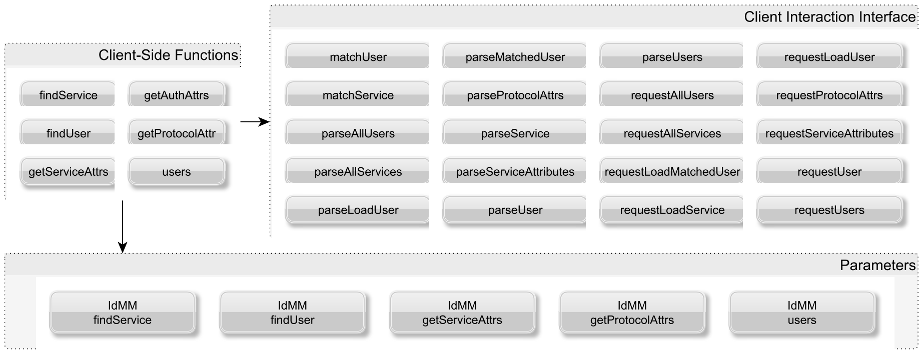

Figure 6: Client-Side Functions Refinement

6 The $\mathrm{IdMM}_{\mathrm{Cloud}}$ Agent

The $\mathrm{IdMM}_{\mathrm{Cloud}}$ agent is responsible with the interaction with any cloud service. Its purpose is to authenticate/de-authenticate the user to/from a given service and to perform attribute synchronization [see Section 3.1.1]. To achieve this goal the cloud-based functions presented in [Fig. 2] must be further refined.

The function $performLogin$ authenticates the user to a given cloud service using the direct and obfuscated scenarios [see Section 3.1]. The function $performLogout$ de-authenticates the user from the given cloud service. Attribute synchronization is done via the function $syncServiceAttr$. To refine these functions we introduce the $\mathrm{IdMM}_{\mathrm{Cloud}}$ submachine.

If the service supports any protocol based authentication we use the function $performProtocolLogin$ for authentication and $performProtocolLogout$ for de-authentication. For the purpose of this paper we only describe the authentication in the direct and obfuscated cases. The refinement of the *performProtocolLogin* and *performProtocolLogout* functions entails further research into the specification of open-based authentication and authorization protocols. The first step in the refinement of these functions is to allow the IdMM to authenticate a user using an external identity provider, regardless of the protocol. Once this step is complete we intend to further refine the functions by specifying the $\text{IdMM}_{\text{Protocol}}$ agent. We intend to have this agent act as an identity provider itself thus eliminating some of the privacy issues related with utilizing external identity providers.

6.1 Types and Data Structures

Apart from the types and data structures presented in the papers [Vleju 2012a;c] we introduce the type *Method* which will allow us to identify the method being used and the data structure *CloudService* [see Fig. 7]. A *CloudService* data type retains any pertinent information about the cloud service. This includes the service's URI, the names of the attributes required to use the service and the list of attributes used by the service.

```
type Method = Login | Logout | Sync

data CloudService = (uri, requiredAttrs, attrs)
   uri∈ String ,
   attrs∈ Attr*
   requiredAttrs∈ String*
```

Figure 7: IdMM_{Cloud} Types and Data Structures

6.2 A Plugin-Based System

In [Section 5] we described the communication between the $\text{IdMM}_{\text{Client}}$ and the client's directory by reducing it to a client interaction interface whose implementation is dependent of the type of the directory used by the client. The description of the interaction with the cloud follows a similar direction. Since the direct and obfuscated interaction scenarios entail the usage of the cloud provider's authentication system we must make an abstraction of this system. As such the refinement of the cloud-based functions will also yield an interface of abstract functions.

In order for the $\text{IdMM}_{\text{Cloud}}$ to authenticate/de-authenticate a user from a cloud service a suitable service-dependent implementation must be provided. As such we describe the interface of abstract functions as a *CloudPlugin* [see Fig. 8].

We define the function $plugins()$ which returns all the cloud plugins currently existing in the $\mathrm{IdMM}_{\mathrm{Cloud}}$ agent's scope. We will now detail each of the functions presented in [Fig. 8].

```
type CloudPlugin = (id, generic, getSyncServices, makeAPIAuthentication,
  makeAPIDeAuthentication, makeAPISync, mapClientAttrsToCloud,
  matchService, parseAuthParameters, parseGenericAuthentication,
  parseGenericDeAuthentication, parseGenericSync, requestAuthParameters,
  requestGenericAuthentication, requestGenericDeAuthentication,
  requestGenericSync, uriAuth, uriDeAuth
  where
    id ∈ String, generic ∈ Boolean, CloudService* getSyncServices(),
    Boolean makeAPIAuthentication(Service s, Attr *attrs),
    Boolean makeAPIDeAuthentication(String uri),
    Boolean makeAPISync(CloudService s),
    Attr* mapClientAttrsToCloud(Attr* attrs, Methodm),
    Boolean matchService(Service s),
    Service parseAuthParameters(Object response),
    Boolean parseGenericAuthentication(Object response),
    Boolean parseGenericDeAuthentication(Object response),
    Boolean parseGenericSync(Object response),
    Object requestAuthParameters(String uri),
    Object requestGenericAuthentication(CloudService s),
    Object requestGenericDeAuthentication(String uri),
    Object requestGenericSync(CloudService s),
    uriAuth ∈ String, uriDeAuth ∈ String
```

Figure 8: CloudPlugin Interface

6.2.1 CloudPlugin Information

The type *CloudPlugin* has several fields which provide some information about the type of the plugin. The field *id* represents the unique identifier for the plugin. The field *generic* is false if the service corresponding to this plugin offers its own custom API for authentication. The fields *uriAuth* and *uriDeAuth* represent the entry point URIs for authentication and de-authentication processes.

To identify the correct plugin for a given service the function *matchService* is used. It returns *true* if the plugin can be used in conjunction with the service or *false* otherwise.

6.2.2 Mapping Directory Attributes to Cloud Attributes

Since the IdMM makes an abstraction with respect to the structure and mechanisms of both the client's directory and the cloud service there has to be a mapping between the attributes stored in client's directory and the ones used by the cloud service. This mapping is done via the *mapClientAttrsToCloud* function. The function takes a list of attributes representing the values from the client's directory and renames them so the values correspond to the required cloud-based names.

In [Vleju 2013] we used a special attribute called *'map'* to store the mappings. Each call of the *getServiceAttrs* or the *getAuthAttrs* function would include such an attribute. The value of the map attribute is a list of attributes where the name corresponds to the cloud service attribute name and the value represents the name used in the directory. For example, if *getServiceAttrs* would return the following list:

$$[('username','j.smith') \cdot ('password','12345') \\ \cdot('map', [('user','username') \cdot ('pass','password')])]$$

then the result of the *mapClientAttrsToCloud* function applied to this list should be:

$$[('user','j.smith') \cdot ('pass','12345')]$$

6.2.3 Authenticating to a Service

There are two means of authenticating to a cloud service. If the service has its own API implemented the function *makeAPIAuthentication* is used. This function returns *true* or *false* depending on whether the authentication was successful. If the service does not have an API implemented then a generic one must be provided. We use the functions *requestAuthParameters* and *parseAuthParameters* to obtain the necessary information required for authentication. To make the actual authentication we use the functions *requestGenericAuthentication* and *parseGenericAuthentication*, the latter of which will return *true* if the authentication was successful.

6.2.4 De-Authentication from a Service

As with the authentication process there are two ways of performing a deauthentication. If the service has a custom API implemented we use the function *makeAPIDeAuthentication*. Otherwise we use the functions *requestGenericDeAuthentication* and *parseGenericDeAuthentication*. As before the function *parseGenericDeAuthentication* will return *true* if the de-authentication was successful.

6.2.5 Synchronizing Service Attributes

Attribute synchronization is needed when the value of an attribute changes in the client's directory. To synchronize service attributes we first use the function *getSyncServices* to retrieve all the synchronization endpoints for a particular service. Afterwards we make the actual synchronization using the *makeAPISync* function if the service has an API or the *requestGenericSync* and *parseGenericSync* functions otherwise. The function *syncServiceAttr* returns true if all synchronizations succeeded.

6.3 Dynamic Frame and States

The $\text{IdMM}_{\text{Cloud}}$ submachine has 11 states [see Fig. 9]. The submachine's dynamic frame is composed of: the current service, the eventual result that will be returned, the list of attributes provided as parameters, the current method, the *CloudPlugin* that will be used and a halt flag. The machine will halt its execution when *halt* gets a defined value.

States	Dynamic Frame
type State= DetermineType	service∈Service
| FindPlugin	result∈Boolean
| MakeAPIAuth | MakeAPIDeAuth	attrs∈Attr∗
| MakeAPISync | MakeGenericSync	state∈States
| MakeGenericAuth | MakeGenericDeAuth	halt∈String
| PerformLogin | PerformLogout	method∈Method
| SyncServiceAttrs	plugin∈CloudPlugin

Figure 9: $IdMM_{Cloud}$ Dynamic Frame and States

6.4 Rules

The rules for the $\text{IdMM}_{\text{Cloud}}$ can be divided into three categories. The setup rules [see Fig. 10] are used to initialize the machine. While in the *PerformLogin*, *PerformLogout*, *SyncServiceAttrs* states the values for *plugin* and *method* are set and the state is set to *FindPlugin*. The *FindPlugin* state is responsible for finding the appropriate plugin for the given service. If a plugin is found the machine switches to the *DetermineType* state where based on the method and the value of the plugin's *generic* field it determines the next state. The initial state of the machine is determined by the function that started the machine [see Fig. 11].

Authentication is handled while in the *MakeAPIAuth* or *MakeGenericAuth* state [see Fig. 12]. The machine uses the plugin's functions described in [Section 6.2] and performs the authentication. In both cases the directory attributes are mapped to the corresponding cloud values. Similarly the de-authentication is done via the *MakeAPIDeAuth* or *MakeGenericDeAuth* state [see Fig. 12].

The synchronization is done while in the *MakeGenericSync* or *MakeAPISync* state [see Fig. 13]. The machine retrieves all the synchronization endpoints, maps the directory attributes to the cloud values and then performs the synchronization for each endpoint.

```
PerformLogin →
  plugin:=∅
  method:=Login
  state:=FindPlugin

PerformLogout →
  plugin:=∅
  method:=Logout
  state:=FindPlugin

SyncServiceAttrs →
  plugin:=∅
  method:=Sync
  state:=FindPlugin

FindPlugin →
  forall p ∈ plugins() do
    if p.matchService(s) then
      plugin:=p
  if plugin=∅ then
    halt:='No plugin found'
  else
    state:=DetermineType
```

```
DetermineType →
  if plugin.generic then
    case method of
      Login →
        state:=MakeGenericAuth
      Logout →
        state:=MakeGenericDeAuth
      Sync →
        state:=MakeGenericSync
  else
    case method of
      Login →
        state:=MakeAPIAuth
      Logout →
        state:=MakeAPIDeAuth
      Sync →
        state:=MakeAPISync
```

Figure 10: $IdMM_{Cloud}$ Setup Rules

```
Boolean performLogin(Services,
          Attr * attributes)=
  result:=false
  halt:=∅
  service:=s
  attrs:=attributes
  state:=PerformLogin
  while halt=∅ do
    fire IdMM_Cloud rules
  return result
```

```
Boolean performLogout(Services)=
  result:=false
  halt:=∅
  service:=s
  state:=PerformLogout
  while halt=∅ do
    fire IdMM_cloud rules
  return result
```

```
Boolean syncServiceAttrs(Services, Attr * attributes)=
  result:=false
  halt:=∅
  service:=s
  attrs:=attributes
  state:=SyncServiceAttrs
  while halt=∅ do
    fire IdMM_cloud rules
  return result
```

Other Functions

```
Plugin* plugins()

Attr* extractAttributes(Attr* attributes, String* toExtract)=
  let result=[], result∈Attr*
  forall name∈toExtract do
    let attr = findAttribute(name, attributes)
    if attr≠ ∅ then
      result = result·[attr]
  return result

Attr findAttribute(String name, Attr* attributes)=
  if ∃a | a ∈ attrs ∧ a.name = name then
    return a
  else
    return ∅
```

Figure 11: Cloud-Based Functions

```
MakeAPIAuth →
  let attributes = plugin.mapClientAttrsToCloud(attrs, method)
  result:=plugin.makeAPIAuthentication(service, attributes)
  halt:='halt'

MakeGenericAuth →
  let response = plugin.requestAuthParameters(plugin.uriAuth)
  if response=∅ then
    halt:='No auth parameters found'
  else
    let s = plugin.parseAuthParameters(response)
    let attributes = plugin.mapClientAttrsToCloud(attrs, method)
    s.attrs = [s.attrs]·[extractAttributes(attributes, s.requiredAttrs)]
    response:=plugin.requestGenericAuthentication(s)
    if response=∅ then
      halt:='Failed to make authentication'
    else
      result:=plugin.parseGenericAuthentication(response)
      halt:='halt'

MakeAPIDeAuth →
  result:=plugin.makeAPIDeAuthentication(plugin.uriDeAuth)
  halt:='halt'

MakeGenericDeAuth →
  let response = plugin.requestGenericDeAuthentication(plugin.uriDeAuth)
  if response=∅ then
    halt:='Cannot make deauthentication'
  else
    result:=plugin.parseGenericDeAuthentication(response)
    halt:='halt'
```

Figure 12: IdMM$_{Cloud}$ Authentication and De-Authentication Rules

```
MakeGenericSync →
  let services = plugin.getSyncServices()
  let attributes = plugin.mapClientAttrsToCloud(attrs, method)
  forall s ∈ services do
    s.attrs = [s.attrs]·[extractAttributes(attributes, s.requiredAttrs)]
    let response = plugin.requestGenericSync(s)
    if response=∅ then
      halt:='Failed to sync'
    else
      if plugin.parseGenericSync(response)=false then
        halt:='Failed to sync'
  if halt=∅ then
    result:=true
    halt:='halt'

MakeAPISync →
  let services = plugin.getSyncServices()
  let attributes = plugin.mapClientAttrsToCloud(attrs, method)
  forall s ∈ services do
    s.attrs = [s.attrs]·[extractAttributes(attributes, s.requiredAttrs)]
    if plugin.makeAPISync(s)=false then
      halt:='Failed to sync'
  if halt=∅ then
    result:=true
    halt:='halt'
```

Figure 13: IdMM$_{Cloud}$ Synchronization Rules

7 The IdMM$_{\text{User}}$ Agent

The IdMM$_{\text{User}}$ agent is responsible for the interaction with an individual user. Its purpose is to handle inputs from the user and to display the appropriate messages. This goal is achieved by implementing the user-based functions presented in [Fig. 2]. Since the implementation of these functions is software-dependent we opted to keep the functions abstract. In this section we will detail only the specification of the functions and exemplify their implementation in our proof of concept implementation presented in [Vleju 2013].

The function *prompt* is used to prompt the user into typing his or her credentials. The function returns a list of attributes representing the credentials. In [Vleju 2013] the credentials are stored in the browser's *localStorage* variable. The user inputs the data as part of the extensions options. In this case there is no user interaction upon calling the function as its implementation entails reading the values from the *localStorage*.

The function *error* is used to notify the user of an error that occurred during the authentication process. The function takes one parameter, the error message, and displays this message to the user. In [Vleju 2013] the error message is displayed using Chrome's extension popup mechanism. Upon an error the extension icon is changed and the message will be displayed only when the user clicks this icon.

The function *redirect* is used to redirect the user to a specific URI, specified as a parameter. Since the implementation described in [Vleju 2013] is browser-based the call of this function has two effects. If the user is already authenticated to the service then the function will be void. If the user was authenticated by the IdMM then the window's URL will be set to the provided URL. This will have the effect of reloading the page, since the URL does not change.

$redirect(String\ \ uri)$ $error(String\ \ message)$ $Attr*\ \ prompt()$	$triggerLogoff() =$ $\mathtt{instr} := UserLogout$
$triggerAuth(String\ \ uri) =$ $\mathtt{instr} := ServiceLogin(uri)$	$triggerDeAuth(String\ \ uri) =$ $\mathtt{instr} := ServiceLogout(uri)$

Figure 14: User-Based Functions

When the user enters a URI the underlying event will call the function *triggerAuth* [see Fig. 14]. This function simply changes the state of the $\mathrm{IdMM_{Core}}$ agent to *ServiceLogin* [7]. When the user wants to de-authenticate from a service the underlying event will call the function *triggerDeAuth*. This function sets the state of the $\mathrm{IdMM_{Core}}$ agent to *ServiceLogout*. Similarly, the de-authentication from the IdMM is done via the *triggerLogoff* function, which sets the state to *UserLogout*.

[7] The $\mathrm{IdMM_{Core}}$ agent fires IdMM rules until its dynamic frame variable *halt* gets a defined value. As such changing the state to *ServiceLogin* has the effect of triggering the *ServiceLogin* rule.

In [Vleju 2013] there are two events for triggering a service authentication. One event monitors the creation of new pages: the user opens a new tab and enters a URL. The second event monitors the page's URL. If this value changes then the service authentication is triggered. The functions *triggerAuth* and *triggerDeAuth* are called synchronously. This is to ensure that an authentication or de-authentication event is successfully completed. The de-authentication from a service is done once every window used by the service is closed. The de-authentication from the IdMM is done when the browser itself is closed. Since the $\text{IdMM}_{\text{Core}}$, $\text{IdMM}_{\text{Cloud}}$ and $\text{IdMM}_{\text{User}}$ agents are implemented as a Google Chrome Extension the domain of single sign-on service is the browser itself. Moving the $\text{IdMM}_{\text{Core}}$ agent outside the browser (as a separate service, for example) will switch the domain (to the user's device).

8 Proof of Concept Implementation

The specification of the $\text{IdMM}_{\text{Core}}$, $\text{IdMM}_{\text{Client}}$, $\text{IdMM}_{\text{Cloud}}$ and $\text{IdMM}_{\text{User}}$ agents allows for an automatic authentication tool for cloud-based services in the direct and obfuscated interaction scenarios. Concurrently we have created a proof of concept implementation, presented in [Vleju 2013], which follows the architecture presented in [Fig. 1]. An on-line demo of the implementation can be found at `http://youtu.be/DoM36DOydkA`. The IdMM core, cloud and user agents run in a Google Chrome browser as an extension. The implementation was written in Java with Google Web Toolkit (GWT) being used for JavaScript compilation. The $\text{IdMM}_{\text{Client}}$ agent runs separately on top of a Tomcat server. In this case the communication between the $\text{IdMM}_{\text{Core}}$ agent and the $\text{IdMM}_{\text{Client}}$ agent is done via the GWT RPC framework.

During the implementation phase we became aware that some web-based applications do require the user to enter dynamically generated data for the sole purpose of disallowing automated requests. Most often this is achieved via captcha messages. As such we were forced to introduce a new user-based function, *String captcha*(*Object message*), to allow the user to input captcha messages.

9 Conclusions and Further Work

The IdMM represents a single sign-on service that performs automatic authentication and authorization to cloud services. The IdMM is composed of six agents (four of which are described above). The refinements of the $\text{IdMM}_{\text{Cloud}}$ [see Section 6] and the $\text{IdMM}_{\text{User}}$ [see Section 7] agents allow the system to automatically authenticate a user to cloud services. In [Section 8] we described a proof of concept implementation of the IdMM. With the refinements presented in this paper a user can be automatically authenticated to a service using either the direct or obfuscated client-to-cloud interaction scenarios presented in [Section 3.1].

Because of the single sign-on capability of the IdMM, an individual user may not be aware of the credentials used to authenticate him or her to a cloud service, thus mitigating the risk of phishing attacks. In addition, all identity related data is stored on-premise allowing the client to have better control over what information is sent to cloud services. This is further enhanced via the introduction of obfuscated and partially obfuscated identities allowing a client to connect to services without using (or partially using) the real identity-related data.

To satisfy the requirements of the protocol-based interaction scenario the *performProtocolLogin* and *performProtocolLogout* functions must also be refined. This entails the introduction of the $\text{IdMM}_{\text{Protocol}}$ agent which will act as an identity provider for a given protocol. With the successful introduction of the $\text{IdMM}_{\text{Protocol}}$ agent we can then focus on the authorization and access management part. We intend to study the existing identity access control solutions and adapt them to our use cases.

Having completed the addition of access control mechanisms to the IdMM we can then focus on introducing of the $\text{IdMM}_{\text{Provisioning}}$ agent. This agent will allow for an easier management of the identities and access rights stored both on the client as well as the cloud provider's systems. The agent will be responsible for the creation, modification and deletion of identities on the client's directory as well as account creation, synchronization and deletion on the necessary cloud services. The system will also be responsible for periodical passwords resets on cloud services. As a further step in enhancing the functionalities of the IdMM we plan to introduce an auditing and logging system.

References

[Ahn et al. 2009] Ahn, G.-J., Ko, M., Shehab, M.: "Privacy-enhanced user-centric identity management"; Communications, 2009. ICC '09. IEEE International Conference on; 1 –5; 2009.

[Alpár et al. 2011] Alpár, G., Hoepman, J.-H., Siljee, J.: "The identity crisis. security, privacy and usability issues in identity management"; CoRR; abs/1101.0427 (2011).

[Alrodhan and Mitchell 2007] Alrodhan, W., Mitchell, C.: "Addressing privacy issues in cardspace"; Third International Symposium on Information Assurance and Security, 2007. IAS 2007; 285 –291; 2007.

[Bakken et al. 2004] Bakken, D., Rarameswaran, R., Blough, D., Franz, A., Palmer, T.: "Data obfuscation: anonymity and desensitization of usable data sets"; Security Privacy, IEEE; 2 (2004), 6, 34 – 41.

[Börger and Stärk 2003] Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis; Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Brunette and Mogull 2009] Brunette, G., Mogull, R.: "Security Guidance for critical areas of focus in Cloud Computing V2. 1"; (2009).

[Cameron et al. 2008] Cameron, K., Posch, R., Rannenberg, K.: "Proposal for a Common Identity Framework: A User-Centric Identity Metasystem"; (2008).

[Cloud Security Alliance 2010] Cloud Security Alliance: "Top threats to cloud computing"; (2010).

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

[Oppliger et al. 2007] Oppliger, R., Gajek, S., Hauser, R.: "Security of microsoft's identity metasystem and cardspace"; Communication in Distributed Systems (KiVS), 2007 ITG-GI Conference; (2007), 1 –12.

[Prodromou 2007] Prodromou, E.: "Openid privacy concerns"; (2007).

[Schechter et al. 2007] Schechter, S., Dhamija, R., Ozment, A., Fischer, I.: "The emperor's new security indicators"; Security and Privacy, 2007. SP '07. IEEE Symposium on; 51 –65; 2007.

[Schewe et al. 2011] Schewe, K.-D., Bósa, K., Lampesberger, H., Ma, J., Rady, M., Vleju, M. B.: "Challenges in cloud computing"; Scalable Computing: Practice and Experience; 12 (2011), 4, 385–390.

[The Open Group Identity Management Work Area 2004] The Open Group Identity Management Work Area: "Identity management"; (2004).

[Vleju 2012a] Vleju, M. B.: "A client-centric asm-based approach to identity management in cloud computing"; Advances in Conceptual Modeling; volume 7518 of Lecture Notes in Computer Science; 34–43; Springer Berlin Heidelberg, 2012a.

[Vleju 2012b] Vleju, M. B.: "A client-centric identity management tool for small and medium enterprises using cloud services"; 4th Workshop on Software Services; 15–19; Bled, Slovenia, 2012b.

[Vleju 2012c] Vleju, M. B.: "Interaction of the idmm with a client-side identity management component"; Technical report; Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC), Johannes Kepler University Linz; Hagenberg, Austria (2012c).

[Vleju 2013] Vleju, M. B.: "A practical implementation of a client-centric identity management tool for cloud computing"; EUROCAST-Computer Aided Systems Theory; Gran Canaria, Spain, 2013.

[Zhang and Chen 2010] Zhang, Y., Chen, J.-L.: "Universal identity management model based on anonymous credentials"; Services Computing (SCC), 2010 IEEE International Conference on; 305 –312; 2010.