Capability Transfer for Service Collaboration

R. Alexander Milowski
ILCC, School of Informatics
University of Edinburgh
Email: alex@milowski.com

Henry S. Thompson
ILCC, School of Informatics
University of Edinburgh
Email: ht@inf.ed.ac.uk

***Abstract*—As the use of collaborative computing in e-Science expands on the Internet, the need to provide access to protected resources (e.g. data sets, storage, web services, or other computational peers) becomes a central issue. Yet, providing access typically requires using authentication credentials and exposing credentials is a breach of security. Simple examples of this conflict arise in the use of "cloud services" for storage. As these services are used by semi-trusted delegates, how do the delegates access your resources, without your credentials, while being restricted to the exact set of operations the task requires? We have developed a new security scheme, called the Capability Trust Exchange Model, that is designed to be layered over existing web services and standards to provide specific capabilities to a semi-trusted delegate without exposing credentials.**

I. Intent between Collaborating Agents

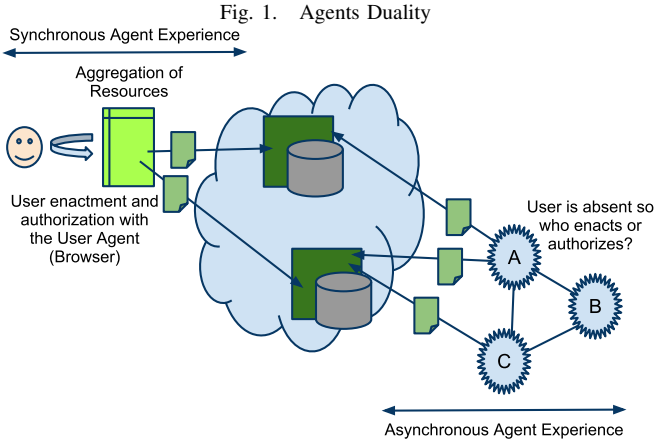
With respect to consumer services, collaboration between different services provided by different organizations is a well used and accepted technique that provides the user with a unified user experience within the context of the web browser. When an HTML document contains correct references to a variety of services from different providers, a user accessing that document via a web browser has access to those services just by viewing the document. The browser, in combination with the user's permission and in compliance with the same-origin policies, acts as a point of aggregation.

In contrast, on the computational web, collaboration is between systems acting as agents of the user who is pushed to the edge of the computation where that user may initiate a process and is subsequently not directly involved. The user typical starts by authenticating and authorizing the system to perform a particular task. These tasks are typically ones that require time and resources beyond the patience and capabilities of the user directly. As a result, the user expects to withdraw from the process and will not necessarily be directly involved in the further processing, recruitment of services, and delegation of tasks.

This contrast represents a duality between the experiences of the user via a web browser and the viewpoint of computational agents operating on its behalf. In figure 1, the user's browser is on the upper left and the computational agents are on the lower right. In the middle of the figure are services and resources with which the user or agents might interact.

On the consumer web, a person is typically seated in front of the web browser and the person's experience with the application shown in the browser is effectively synchronous. While the individual components within the web browser may

Fig. 1. Agents Duality



be communicating asynchronously and in parallel, the user's expectation is synchronous with regards to when the aggregation completes. This means the user expects the document to load and execute in a "reasonable" amount of time.

A user in this system performs an enactment of both intent and authorization within the web browser. While the user has simply asked to visit an address on the web via a link, bookmark, or other such annotation, the user has explicitly expressed intent and implicitly authorized the web browser to load, process, and execute the resource retrieved. Within the security policies of the browser, any number of services may be aggregated in front of the user.

These aggregated services may further ask the user for authorization. Once authorized, cooperation between different web services may ensue. For example, a map may be loaded with the user's annotations or picture might be uploaded to a photo gallery on another service. With respect to whatever the desired behavior has been, the user has watched over and authorized the activities.

While the browser user agent receives information by retrieving documents, the computational agent is typically sent information upon which it is supposed to compute. The agent may interact with a number of different resources or other agents during this computation. The result of this computation may not be available for hours, days, or weeks. The agent's view of the aggregation is asynchronous where the response containing the result of the computation is delayed by the necessary processing time.

When a user instructs a computational agent to act on

the user's behalf, the user is no longer directly involved in the process. That leads directly to the thorny issue of which "entity" enacts intent and authorizes activities. Solving this issue is central to extending collaboration of services from the individual to automated systems.

As collaborative computing in e-Science expands on the Internet, the use of anonymous or semi-trusted agents for computing will only increase. As a result, the need to represent the user's intent and perform enactment and authorization to provide access to protected resources (e.g. data sets, storage, web services, or other computational peers) is a central issue for collaboration between services. Yet, doing so requires going beyond the typical conflicted compromise of exposing credentials in order to provide access to secured services.

II. The Storage Triangle

When tasks are delegated to services provided by different organizations, problems arise as to where to send the results. The interaction between the ***Requestor*** and the ***Delegate*** is usually in the form of a short message that encapsulates a set of instructions for the task. The *Requestor* then leaves the conversation expecting the *Delegate* to complete the task and "send" them the results. Unfortunately, as the *Requestor* has left the conversation, where does the *Delegate* send the results?

While e-mail is one way in which the *Delegate* might return the results of completing the task, if the receiver of the result is yet another service, e-mail is a less than ideal way of receiving data. In response to this problem, the *Requestor* may want to introduce some kind of ***Storage Service*** (i.e. a web-based storage system like Amazon S3 [1]) that it expects the *Delegate* to use. Typically, the *Delegate* must have some kind of privileged access to the *Storage Service* to actually store the results.

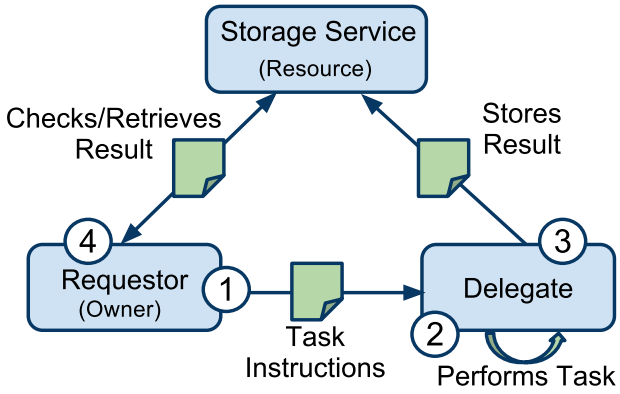
The problem in this scenario is that the *Delegate* may be provided by another organization that is a semi-trusted principal and so the *Delegate* is only semi-trusted. While the *Requestor* has made a number of assessments of the *Delegate* to ensure that the task being delegated will be performed as requested, that trust will degrade quickly with regards to the system policies that might be in place to protect data, credentials, etc. being compromised by an attacking third party. As a result, the *Requestor* is unlikely to want to pass its authentication credentials for a *Storage Service* along to the *Delegate*.

This scenario is shown in figure 2 and is described as follows:

1) The *Requestor* sends a message of instructions to a *Delegate* containing a task to be performed. The message includes a reference to the storage service.
2) The task is performed by the *Delegate*.
3) Upon completion, the *Delegate* stores the results via the *Storage Service*.
4) Later, the *Requestor* can access the *Storage Service* to check for and review the results.

Similarly, if the *Delegate* only receives a reference to the necessary inputs to their task, it must request and receive these inputs. The *Delegate* may need to access information that is protected. While the *Requestor* may want to expose a certain amount of data to the *Delegate*, that access must be limited to exactly the information necessary and no more.

Fig. 2. Storage Triangle Interactions



III. Capability Exchanges

Our use of capability transfers was inspired, in part, by the "Web Keys" proposal described in [2]. While this proposed solution to ***Cross Site Request Forgery (XSRF)*** [3] utilizes fragment identifiers to provide browsers with access to protected resources, we take this idea further and adjust the placement of the "encoded key" or "token" to allow flexible use of web-based protocols for the transferred capability.

Traditionally, a ***Capability*** is an object that encapsulates a reference to a group of data along with the authorization and associated access rights, rules for use, and other attributes that describe how the data can be used within the context of the *Capability*. Within a capability-based system, having a reference to data and having the permission to access or manipulate the data tends to come together. The use of the *Capability* is typically checked by a ***Reference Monitor*** whose role is to check the access by a process to data within a *Capability*. The *Capability* is also granted against a ***Security Policy***, which is a set of rules that determines the types of capabilities that may be transferred for the data [4]. In what follows, these ideas of capability-based security have been applied within the context of distributed computation on the Web to allow agents to transfer capabilities for web resources to task delegates.

When the scenario of the storage triangle plays out on the Web, the "data" is a set of resources that are identified and accessible via some set of URIs [5]. A ***Resource*** is typically a storage unit within some system (e.g. Amazon S3 buckets [1]) that has both identity and manipulative access via a URI. As a result, the capability is a packaging of the URIs that identify the resources and specific access methods allowed against them.

The *Requestor* in the storage triangle must also have privileged access to the *Resource*. Within the context of this scheme, we'll assume that this privileged access is an result of

the fact that the *Requestor* is the resource owner. Although, in fact, the role actor needs only the permissions for operations being delegated. That is, in this specific role, the ***Owner*** has a trusted relationship with the system providing the *Resource*, which is typically via a set of authentication credentials over some protocol, where the operations the *Owner* chooses to delegate must be ones it can perform itself. The term "Owner" was simply choosen because the actor has been granted, and so "owns", the rights to the operations it can delegate. Further, this use of "owner" should not be confused with a "service provider" (e.g. a third party storage provider like Amazon S3) as the "owner" pays for, utilizes, and "owns" data resources that are stored and made accessible by systems owned and provided by the service provider.

Within the process of granting and transferring capabilities against any resource on the Web, there is an additional actor called the ***Resource Monitor***. The role of this actor is the same as the *Reference Monitor* in capability-based security parlance but the *Resource Monitor* is also a resource in itself that provides a web-based API, like OAuth [6], that is utilized by the *Delegate* to negotiate access to the resource. The *Resource Monitor* has a semi-trusted relationship with the actual *Owner* and a trusted relationship with the *Delegate*. While the *Owner* should proceed with caution, the *Resource Monitor* is typically a known entity to which the *Owner* will delegate the task of brokering access to a capability.

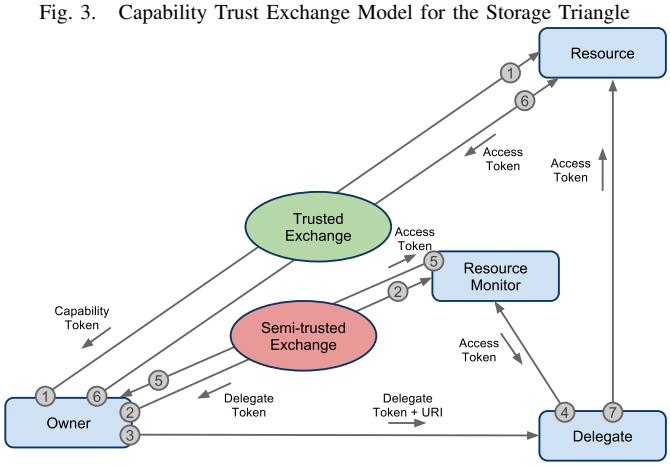
In figure 3, the ***Capability Trust Exchange Model*** is shown. This model uses two exchanges: a trusted exchange between the *Owner* and the *Resource* and a semi-trusted exchange between the *Owner* and the *Resource Monitor*. The trusted exchange provides a means for the owner of the resource to configure capabilities and grant access against those capabilities. This exchange may be through public or private means depending on the level of security needed.

The semi-trusted exchange represents a trust relationship between organizations or individuals. As this exchange exists between organizations, it may be manifested as infrastructure provided by either the *Owner* or *Resource Monitor*. Although, ultimately it is the *Owner* who chooses to establish a relationship with another organization who is providing the *Delegate* and this relationship is negotiated between the *Owner* and *Resource Monitor*.

Finally, within the whole scheme, the *Delegate* has a tenuous relationship to the other actors. The *Delegate* has some registered identity with the *Resource Monitor* that it will use to negotiate access to the capability it has been granted. This negotiation is accomplished by checking the delegate's grant via the semi-trusted exchange and then granting access via the trusted exchange.

At the start of the process, in step 1 of figure 3, the *Owner* first defines the capability against a set of resources by sending a request to the system providing the *Resource*. This exchange with the *Resource* is via some trusted exchange mechanism that may need authentication credentials or may be provided through some other mechanism. In either circumstance, if the *Resource* accepts the capability's definition, a reference in the

Fig. 3. Capability Trust Exchange Model for the Storage Triangle



form of a ***Capability Token*** is generated and returned to the *Owner*.

The *Owner* is now able to do whatever it would like with this capability via the *Capability Token*. As it represents the capability granted against a set of resources, in step 2, the *Owner* can pass a reference (i.e. a generated identifier) to this token, but not the token itself, to the semi-trusted *Resource Monitor* for delegation via some semi-trusted exchange. The actual *Capability Token* should be hidden from parties outside of the *Owner* and the *Resource* to protect it from illicit manipulation by other parties. The *Resource Monitor* uses this reference to negotiate access on behalf of the *Delegate* at a later point and so without the actual *Capability Token*, the capability cannot be changed for the *Delegate*, even if either has access to the *Resource*. Similar to the *Capability Token*, the *Resource Monitor* generates and returns a ***Delegate Token*** to the *Owner* and maintains an association between the referenced *Capability Token* and the *Delegate Token*.

At this point, in step 3, the *Owner* can transfer the capability via some interaction with the *Delegate*. This interaction typically identifies both the *Resource* set and capability by passing a pair containing a URI and the *Delegate Token*, but the URI may be implied by the exchanged message or task. Nevertheless, the *Delegate* now must attempt to exchange the *Delegate Token* for an *Access Token* that can be used to access the actual *Resource*.

The steps 4 though 6 would typically happen in a synchronous manner where steps 5 and 6 are subsumed. At step 4, the *Delegate* attempts to exchange the *Delegate Token* for an *Access Token*. An ***Access Token*** is a token generated by the *Resource* and associated with the capability. The token is used by the *Resource* in the web protocol to access the capability (e.g. as a query parameter) and is checked against the allowed operations associated with the capability when the *Delegate* accesses the *Resource*.

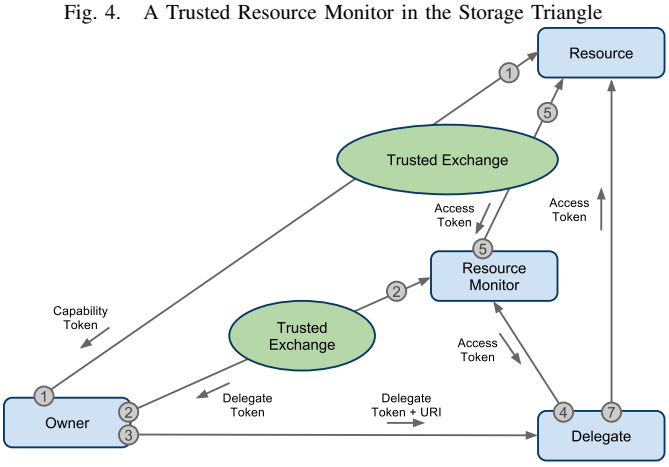
To receive an *Access Token* to return to the *Delegate*, in step 5, the *Resource Monitor* has a semi-trusted exchange with the *Owner*. During this exchange, in step 6, the *Owner* has a trusted exchange with the *Resource* to generate an *Access*

Token. If this sequence is successful, this token is passed back through the same chain of communications to the *Delegate*.

At this point, the *Owner's* agent has been re-inserted into the process–which may seem to violate the scenario where the user has left the process that it originally initiated. The *Delegate* may access the capability at an unpredictable later time at which the capability may have been revoked or its current *Access Token* may have expired. These situations require renegotiation with the *Resource* to ensure the capability is active and available. In addition, it is the *Resource's* system that generates the *Access Token*. Unfortunately, the *Resource Monitor* is not necessarily associated with the *Resource* directly and so requires an exchange directly with some representation of the *Owner*. In an environment where these actors are provided within the same infrastructure, the exchange between the *Resource Monitor* and *Resource* is trusted and the *Owner* can be removed from the process. This scenario is shown in figure 4 where step 6 has been omitted and step 5 is an exchange directly with the *Resource*.

At the final step 7, the *Delegate* can access the capability utilizing some protocol and API associated with the URI of the *Resource* that also must encode the *Access Token* received in step 4. When the *Resource* receives the request, it can look up the capability and check to see if the requested operation is allowed. At any point, the *Owner* or *Resource* can revoke the access token and disallow any access by the *Delegate*.

Fig. 4. A Trusted Resource Monitor in the Storage Triangle



A simple example of using capability transfer is when the *Owner* wants to grant the ability for a *Delegate* to post one picture of an active simulation to a specific picture gallery owned by the *Requestor* ([7] and [8]). The action of posting a picture is a single atomic operation against a URI (an HTTP POST). As such, the *Delegate* would only be granted the ability to POST a message (entity body) containing a single picture and not other methods (e.g. GET) against the gallery. The *Resource Monitor* would then act as a broker between the *Delegate* and the authenticated access to the gallery API.

IV. Describing Capabilities

A capability on the Web is a pairing of an identification of a set of resources and a set of constraints on the kinds of operations that can be performed upon those resources. Over most web protocols, these directly translate into a set of URIs that identify the resources and restrictions on the kinds of methods and messages that can be sent to those resources. For example, one might restrict a capability to a single HTTP POST against a very specific URI.

More formally, we'll define a ***Web Capability*** as the pair of:

- A *Target Resource Set*,
- A set of *Operation Constraints*.

A ***Target Resource Set*** is a collection of URIs that identifies a set of resources. While these resources may be related by a common base URI, the resources are not required to have any such relationship. In fact, the simplest example is a singleton containing a specific URI.

To allow simple modeling of access to resource-oriented web services, a *Target Resource Set* is allowed to be unbounded and identify whole families of resources whose URIs match particular patterns or can be identified with a particular base URI. Such patterns may be specified by inclusion or exclusion mechanisms when the target resource set is defined.

Against this set of resources a set of constraints is declared. A ***Operation Constraint*** is a triple of an operation name, an integer, and a set of constraining facets. The operation name typically matches a request type in a protocol such as the method name in HTTP [9]. An implementation uses the operation name to find the set of *Operation Constraints* that apply to a given request. This operation name may be a wildcard to allow matching any operation.

The integer value in the constraint tuple is a non-zero priority that may be positive or negative. A negative value indicates a "knock-out" constraint such that if the *Operation Constraint* matches, the request does not match the capability. Similarly, a request must match some positively prioritized *Operation Constraints* to match the capability. This means the simplest capability is the tuple `(*,1,{})`.

Each constraining facet describes a matching facet that the request against the resource must have. These constraints allow fine-grained control over the the allowed messages that can be sent to any resource in the *Target Resource Set*. It is also possible to constrain information received about the *Delegate* such as network addresses or other client identity information.

Some possible constraining facets are:

- client identity,
- client network address,
- duration (expiration date),
- entity message size,
- entity content type,
- multi-part messages and constraints,
- number of uses.

For example, in the "upload a single picture" example, the constraints might be:

- content type: image/*
- size less than 1MB
- only once

The whole capability for posting a single picture might be described as:

```
( { http://upload.example.com/gallery/12345 } ,
  { ( "POST", 1, {
            starts-with(content-type(),"image/"),
            size() < 1048576,
            uses() < 1
        }
    )
  }
)
```

where the first part of the pair is the target resource set–which here is a singleton listing a specific URI–and the set of *Operation Constraints*. Here the `POST` operation is associated with a set of constraints on the content type of entity body, the size of the entity body, and the number of previous uses of this capability.

While not described here, the capability could easily be encoded in an XML document for exchange between the *Owner* and the *Resource* during its trusted exchange when the capability is defined. The flexibility of the XML format may allow a broad range of capabilities to be described beyond the abilities of the system hosting the resource and so the system may refuse the capability as it is unable or unwilling to support the request. Even though a system may be able to concisely describe its allowed capability constraints, having a standardized markup language for describing capabilities would allow the *Owner* to have interoperability with a number of different systems without changing its vocabulary.

Regardless of the language used to describe the capability, when the capability is used, a system must perform the same set of checks against the request. When a request against the capability is received, it is checked by:

1) Find all matching tuples where the operation name of the request matches. If there are no matches, stop and refuse the request.
2) Order the matching tuples by the integer priority from lowest to highest.
3) Apply constraints of the tuples in their sorted order. For each matching tuple, if the priority is negative, stop and refuse the request. If the priority is positive, stop and grant the request.

V. Trust Exchanges over XMPP

Central to the *Capability Trust Exchange Model*, shown in figure 3, is the idea of trusted and semi-trusted exchanges. While these could be simply realized as web protocols over HTTP, the actual relationship between the *Owner* and the *Resource Monitor* may be one that is negotiated by a larger organization of which the *Owner* is only a part. Similarly, the trusted exchange between the *Owner* and the *Resource* may also be a part of the infrastructure in the *Owner's* environment.

There are any number of possibilities for these trusted exchanges of which the simplest would be some kind of authenticated access via a web API. In the case of the trusted exchange, this may be an appropriate architectural choice. In contrast, in the case of the semi-trusted exchange, the *Owner* is an entity whose relationship is formed via the semi-trusted relationship that is established by its overarching organization.

As such, the *Owner* may not have direct access or credentials for the *Resource Monitor*.

To complicate things further, the owner is an entity who is transient in the process. Granting access whenever the *Delegate* negotiates with the *Resource Monitor* may be delayed when the agent representing the intent of the *Owner* is not available. Specifically, in our scheme, we would like to know when an *Owner's* agent is online and available to participate in approving the exchange of a *Delegate Token* for an *Access Token*. We would also like to protect the identity of the *Owner* as much as possible. As such, the exchange should maximize the knowledge of the presence of the *Owner* and establish some kind of intermediary where these parties can exchange messages without direct knowledge of its identities (e.g. location, IM address, e-mail) or credentials for access.

One key technology to consider here is Extensible Messaging and Presence Protocol (XMPP) [10]. Inherent in the protocol is the ability to tell when a peer is online via presence messages. If both the trusted and semi-trusted exchanges are able to utilize XMPP, the parties in the exchange would have knowledge of who is available for interaction. In addition, these exchanges could be hosted by the larger organizations to which the various parties belong.

XMPP is an XML-based protocol for peer-to-peer communication typically used for instant messaging, voice, or video chat. The essential idea within the XMPP protocol is that the client sends and receives a sequence of stanzas. A stanza is essentially an XML element whose attributes contain routing information and whose element name and content provides typing and the data of the message, respectively. The stanzas exchanged with the peer's server are such that the XML representation can be routed to any number of peers based on the identity of the target. These messages are routed such that the stanza element arrives on the incoming stream of stanza elements at the recipient's client.

Routing is accomplished via the information encoded in the peer's identity. An XMPP identity is represented by a literal value of the form `user@domain/resource` which is very similar to an e-mail address but has the "/resource" suffix used to identify the actual physical location of the user (e.g. "laptop" vs. "mobile"). The hosting server guarantees that the resource used is unique during the bind phase of the initial connection. Within the network of peers, the servers are truly peer-to-peer and each client must be able to make a connection to its domain's XMPP server.

XMPP servers are located via DNS-based lookup of SRV records using the `domain` part of the XMPP identity. Once a client sends a stanza whose destination is outside of its server, the destination's server must be located via a similar SRV record lookup. Routing between servers is then accomplished by a clever server dial-back exchange that gives modest guarantees on the sender.

As an XML-based protocol, XMPP is extensible and provides a number of optional features. One of these features is Multi-User Chat (MUC) [11] that provides the ability for peers to meet and exchange messages within a "room" without necessarily knowing each other and approving contact between the peers. As such, a MUC room can be used to facilitate exchange between peers.

A MUC room facilitates exchanges between peers that may not "know" each other and so cannot directly communicate. Within the room, each room member has an in-room identity and a real identity outside of the room. Messages within the room are routed to in-room identities and then replicated out to the actual room occupant's identity. As such, peers can meet in a room and exchange messages (essentially, XML content) without necessarily knowing their real identities and pre-defining the route and controls for doing so for each peer.

This provides a basic mechanism for the trusted and semi-trusted exchanges. In each exchange, the room and its members are configured by the infrastructure provider–which is typically a hosting organization provided by agreement between participating organizations. Participating members are assigned their own room credentials by which it may join the room and exchange stanza messages with other online room members.

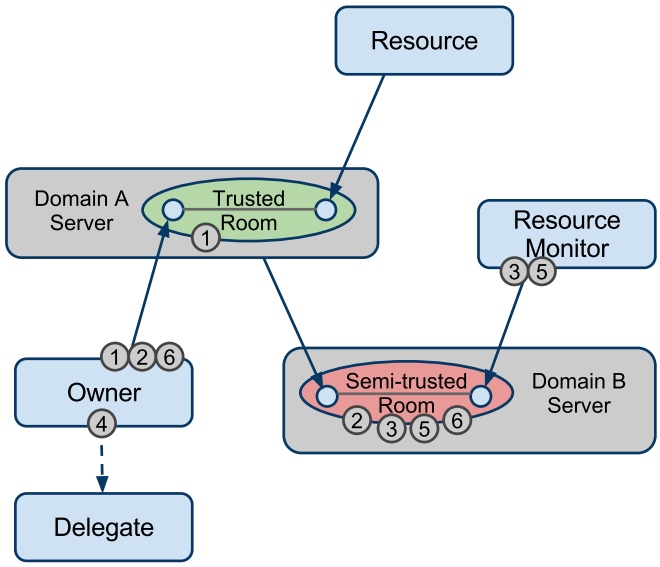
In the case of a trusted exchange (e.g. between the *Owner* and *Resource*), access to the MUC room and the messaging within the room is very tightly controlled. The members and their configuration might be subject to strict security policies to ensure that the exchanges within the room are truly from trusted participants. This may include end-to-end signing of stanza messages [12].

For the semi-trusted exchange, the MUC room is hosted by arrangement between the delegate's and owner's organizations. This manifests itself both as the *Resource Monitor* and the room participant for it. In this case, after a semi-trusted relationship is negotiated, the semi-trusted exchange's MUC room is established and credentials for each participant are created and distributed. It is then the responsibility of the resource *Owner* to participate in both the trusted and semi-trusted exchange MUC rooms to facilitate the capability transfer.

In figure 5, the relationship between XMPP identities, agents, and MUC rooms is shown. The trusted and semi-trusted exchanges are represented by MUC rooms provided by services at two different XMPP servers. The semi-trusted exchange's MUC room has participants from different domains and so must be setup to federate and propagate messages between servers. The configuration of services and location of exchanges is either pre-configured or something that is discovered by the *Owner* when the *Delegate* is discovered.

As XMPP stanzas are XML messages, we could realize this exchange as a conversation of XMPP message elements. We'll assume that all the parties have negotiated their membership in the appropriate MUC rooms. In step 1, the *Owner* creates the capability with an exchange with the *Resource*. In step 2, a reference to the capability is sent to the *Resource Monitor* via the semi-trusted exchange's room which is identified as `semi@conf.capxfer.org`. This stanza uses the standard XMPP message element and contains a custom element containing the reference to the capability:

Fig. 5. XMPP-Based Exchanges




```
<message to='semi@conf.capxfer.org'
         from='semi@conf.capxfer.org/owner'
         xmlns='jabber:client'>
<create-delegation ref='xy23pf2'
                   xmlns='http://capxfer.org/V/exchange'/>
</message>
```

Note that the message is to the MUC room and not the *Resource Monitor*. We do not necessarily know the in-room identity of the *Resource Monitor* at this point even though we could have discovered this previously via service discovery.

In step 3, the *Resource Monitor* in the room responds with a delegate token directly to the in-room identity of the *Owner* that was provided in the previous message in the "from" attribute:

```
<message to='semi@conf.capxfer.org/owner'
         from='semi@conf.capxfer.org/resm'
         xmlns='jabber:client'>
<delegate-token for='xy23pf2' id='d5674322'
                xmlns='http://capxfer.org/V/exchange'/>
</message>
```

Through whatever means provided through the recruitment of the *Delegate*, in step 4, the *Owner* sends the *Delegate Token* to the *Delegate* along with other instructions. When the *Delegate* attempts to exchange this token for access, in step 5, the *Resource Monitor* sends a message to the *Owner* via the room:

```
<message to='semi@conf.capxfer.org/owner'
         from='semi@conf.capxfer.org/resm'
         xmlns='jabber:client'>
<exchange-token ref='d5674322'
                xmlns='http://capxfer.org/V/exchange'/>
</message>
```

Finally, in step 6, if the *Owner* is willing and able to still grant access, an *Access Token* is returned:

```
<message to='semi@conf.capxfer.org/resm'
         from='semi@conf.capxfer.org/owner'
         xmlns='jabber:client'>
<access-token for='d5674322' id='a7488391'
              xmlns='http://capxfer.org/V/exchange'/>
</message>
```

Upon receiving the *Access Token*, the *Resource Monitor* returns this to the *Delegate* through whatever it was requested.

It should be noted that messages sent by the *Owner* in step 1 are routed through the trusted room to the *Resource*, while in step 2 and 6, the *Owner's* messages are routed via the "Domain A Server" to "Domain B Server" and into the semi-trusted room to be received by the *Resource Monitor*. Similarly, in step 3 and 5, messages sent by the *Resource Monitor* are routed first to the semi-trusted room and then via a hop between "Domain B Server" to "Domain A Server" to be received by the real identity of the *Owner*. These routes can be located in the figure 5 by the numbered circles.

VI. Using OAuth for Delegate Access

The strategy throughout this design has been to put the onus on the infrastructure so that the *Delegate* has a simple way to access capabilities that have been transferred. Once the *Delegate* has received the *Delegate Token*, it needs only to exchange this with the *Resource Monitor* to be able to access the *Resource* over standard web protocols.

To keep parity with the web protocols used to access the *Resource*, a standard web-based protocol should be used between the *Resource Monitor* and the *Delegate*. A good fit for this is the OAuth 2.0 protocol [13] as it is a web protocol for transferring access to resources between parties. Specifically, the section on "Obtaining an Access Token" describes what the *Delegate* must accomplish to obtain something it can use with the *Resource*.

As the grant to the *Delegate* has already been received as a *Delegate Token*, the grant is treated as an assertion in terms of OAuth. As described in section 4.1.3 of OAuth 2.0, the *Delegate* converts the *Delegate Token* into an *Access Token* by sending it to the *Resource Monitor* as an assertion. The `assertion_type` parameter is sent with a value of `http://capxfer.org/O/token` and the *Delegate Token* is sent as the `assertion` parameter.

Once the *Resource Monitor* receives a proper request from a known *Delegate*, the process of communicating through the semi-trusted exchange with the *Owner* is initiated. If successful, an *Access Token* is returned. This *Access Token* can be used in further OAuth requests or as specified by the API used to access the *Resource*.

For example, the *Delegate* sends an OAuth access grant request to the *Resource Monitor*, which is defined to be an HTTP POST request:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=assertion&client_id=s6BhdRkqt3&
client_secret=47HDu8s&assertion=xyzzy&
assertion_type=http%3a%2f%2fcapxfer.org%2fO%2ftoken
```

and receives in response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
   "access_token":"SlAV32hkKG",
   "expires_in":3600,
   "refresh_token":"8xLOxBtZp8"
}
```

The *Resource Monitor* may require specific credentials to be provided. For example, a *Delegate* may have credentials with the *Resource Monitor* that have been pre-established and are required to be sent as an `Authorization` header with any access token grant request. These are allowed by the OAuth 2.0 specification and allow additional security measures to be defined by the organization providing the *Resource Monitor*.

Once the *Delegate* has the *Access Token*, it can use the token to access the *Resource*. The OAuth Protocol provides a number of ways to do this via either an `Authorization` header, a URI query parameter, or a form-encoded body parameter. Once received, a resource may refuse the access code and return a `WWW-Authenticate` header describing why.

For example, a *Delegate* might send a `POST` request to a *Resource* as follows:

```
POST /gallery/12345 HTTP/1.1
Host: upload.example.com
Content-Type: image/png
Authorization: OAuth SlAV32hkKG

...
```

In addition to exchanging *Delegate Tokens* for *Access Tokens*, the *Resource Monitor* may provide other services to the *Delegate* such as refreshing tokens. The *Delegate* might need to interact with the *Resource Monitor* more than once to refresh tokens when they expire as specified by the OAuth protocol or other local or remote security policies. Once the *Delegate* has received a response with an error status code and a `WWW-Authenticate` header, the *Delegate* may decide they need to refresh the *Access Token* based on the information received or local policies.

While OAuth is being used only between the *Resource Monitor* and the *Delegate*, there is nothing that prevents OAuth from being used more extensively between the other parties. The only limiting factor is the transient nature of the *Owner* as stated in the asynchronous model of the *Storage Triangle* problem. It is relevant to note that one form of authorization is for a *Resource Monitor* to actually contact the *Owner* over XMPP and send him or her human readable messages asking them to authorize certain delegates. This may integrate well with OAuth 2.0 in a more interactive way than the use described above.

VII. Conclusion

The need for privacy and control over shared data in e-Science has hindered the ability to share data amongst semi-trusted parties. Utilizing capability transfer methodologies allows organizations to model trust relationships in the access they grant to systems performing tasks. As such, having these models directly realized in the way web resources are used is essential.

While organizations and individuals may have strong trust relationships, these human trust relationships break down when systems and agents they operate are expected to collaborate without human involvement. The model presented here recognizes the existence of both the semi-trusted and trusted relationships and allows very specific control to be enacted or revoked. This allows collaboration of systems without user interaction while still preserving its intent.

As the need in services for users to transfer and delegate capabilities against resources increases, services will likely integrate some ability to accomplish the same goals as presented in this paper. This can be seen in the recent development of the "Bucket Policy" facility of Amazon S3 where you can create an access key that can be used to read or write to buckets without the need for authentication. As such, in the realm of service collaboration, it appears these services are ready to adopt the idea of capability transfer and so an opportunity exists to standardize how they are transferred and controlled.

REFERENCES

[1] Amazon Simple Storage Service, Amazon, http://aws.amazon.com/s3/

[2] Close, Tyler, Mashing with permission, W2SP 2008: WEB 2.0 SECURITY AND PRIVACY 2008, http://waterken.sourceforge.net/web-key/

[3] Security Corner: Cross-Site Request Forgeries., Chris Shiflett, December 2004, http://shiflett.org/articles/cross-site-request-forgeries

[4] Kain, Richard Y. and Landwehr, Carl E, On Access Checking in Capability-Based Systems, IEEE Transactions on Software Engineering, 1987, vol 13, pp 202-207

[5] Architecture of the World Wide Web, Volume One, W3C, Jacobs, Ian and Walsh, Norman, December 2004, http://www.w3.org/TR/webarch/

[6] RFC 5849: The OAuth 1.0 Protocol, IETF, April 2010, http://tools.ietf.org/html/rfc5849

[7] Allen, Gabrielle, Exploiting Web 2.0 for Scientific Simulation, eSI Workshop: The Influence and Impact of Web 2.0 on Various Applications, May, 2010, Edinburgh e-Science Institute, Edinburgh, http://www.nesc.ac.uk/esi/events/1078/programme.pdf

[8] Allen, G.; Loffler, F.; Radke, T.; Schnetter, E.; Seidel, E. , Integrating Web 2.0 technologies with scientific simulation codes for real-time collaboration, IEEE Cluster 2009, August, 2009, IEEE, New Orleans, LA, http://www.cluster2009.org/w41.pdf

[9] Hypertext Transfer Protocol – HTTP/1.1, IETF, June 1999, http://tools.ietf.org/html/rfc2616

[10] Extensible Messaging and Presence Protocol (XMPP): Core, IETF, October 2004, http://tools.ietf.org/html/rfc3920

[11] XEP-0045: Multi-User Chat, XMPP Standards Foundation, 2010, http://xmpp.org/extensions/xep-0045.html

[12] End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP), IETF, October 2004, http://tools.ietf.org/html/rfc3923

[13] The OAuth 2.0 Protocol, IETF, July 2010, http://tools.ietf.org/html/draft-ietf-oauth-v2-10