draft-internet2-mace-mlist-middleware-integration-issues-03.html

Authors: Jill Gemmill, John-Paul Robinson
University of Alabama at Birmingham

Comments to: mace-mlist-contact AT internet2 DOT edu

Middleware Integration with Existing Applications: Current Design Issues, with a Focus on Mailing Lists

This material is in part based upon work supported by the National Science Foundation under Grant No. ANI-0330543. "NMI Enabled Open Source Collaboration Tools for Virtual Organizations". Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Additional support was provided by the University of Alabama at Birmingham and Internet2.

Table of Contents

Abstract

Recently, portals have been used to provide a user-friendly interface to a set of grid services and related applications such that user identity is shared across those applications. NMI components such as web initial sign on and Shibboleth provide another way to assemble a suite of applications that share identity and attributes.

Middleware can provide data (eg: subscriber list), identity, user attributes, and rules from external data stores. Applications no longer need to maintain and manage all this information on their own. It is now possible to establish distributed environments that allow a person to glide seamlessly across applications without having to manually log in to each one; the required authentication and authorization can be performed transparently and securely behind the scenes. Taking this approach, it is possible to convert a "silo solution" into a tool that can be used as a pluggable component. Open Source tools are especially interesting candidate applications because of access to the source code and also because of the community development approach.

The Mailing List application is widely used and well-understood as a means for collaborative communication, which makes it a good candidate for understanding the interface of applications with middleware. This document examines middleware integration issues by focusing on the Mailing List application, with a focus on the requirements needed to build new message systems and points for application developers to consider while designing their application that would increase the malleability of existing solutions.

Introduction

Middleware is usually defined as a service that lies between the network and application layers; middleware crosses single machine, operating system, and even domain administrative boundaries. The growing middleware infrastructure is a trend that should be understood and utilized by application developers. Middleware can provide data (eg: subscriber list), identity, user attributes, and rules *from external data stores*. Applications no longer need to maintain and manage all this information on their own. It is now possible to establish distributed environments that allow a person to glide seamlessly across applications without having to manually log in to each one; the required authentication and authorization can be performed transparently and securely behind the scenes. Important middleware technologies include LDAP for directory services and emerging technologies such as Shibboleth for authorization. We are especially interested in standards developed by OASIS, the Liberty Alliance Project, and the Internet2 Middleware Working Groups which are producing working code for this infrastructure.

We believe that **Middleware** (infrastructure services such as: **authentication, authorization, groups, attributes, directories**) can and should be leveraged to provide systems that enable a user to easily read and review all their mailing lists via web interface without appearing to require any special authentication. The Mailing List application is widely used and well-understood as a means for collaborative communication, which makes it a good candidate for understand the interface of applications with middleware.

Mailing lists have traditionally assumed that list members have no organizational affiliation other than their membership in the list itself. While this simplification lowers barriers to collaboration, it also introduces an undesirable subscription management burden when users participate in

collaborations across a variety of lists. As use of mailing lists to support inter-institutional collaborations has increased, the numbers of user logins and passwords has exploded accordingly. Mailing list software often extends its membership assumptions to their web interfaces, making it difficult to integrate those interfaces into enterprise authentication and authorization environments. Such "stand-alone" assumptions can also make it difficult to integrate lists into a larger collaboration environment built using a collection of best-of-breed applications. This How-To is intended as a guide for leveraging these technologies within a mailing list system environment with a focus on the requirements needed to build new message systems and points for application developers to consider while designing their application that would increase the malleability of existing solutions.

1. What Is a Mailing List System?

At the heart of the mailing list application are the processes that distribute individual messages sent to a list address on to a collection of subscriber addresses. As an application that has been around for almost twenty years, the mailing list application has typically evolved into a monolithic and complex software application with tools for creating mailing lists, managing subscribers, and supplying archives. List management software was designed as a closed system, having list definitions and subscribers identified exclusively by data collected and stored directly by the system. In contrast, modern application environments can provide an improved messaging solution, one that is capable of responding to data collected and supplied by other applications, even applications that may be running on entirely different systems.

A modern mailing list software system could be constructed by leveraging components such as a web interface for message submission, a message acceptance processing engine, and a web-based archive interface; each component could be supplied by distinct vendors. In order for these tools to operate as a unified system environment, they must rely on common data collections for defining lists and identifying subscribers. For example, it is desirable for a list archive web front-end to enforce the subscriber access rules that govern the list and, potentially, the web-interface for message submission. The topic of "Modern mailing list systems" could be extended into a more general topic of integrating visual appearance, administrative function, and content. For example, phpbb's focus is forum discussion and its features are geared toward that purpose; it would be nice to use that tool as a nice (and interactive!) web interface to mailing list archives. Having provided this broad brush stroke view of the future, we focus the remainder of our discussion on integration with **middleware**.

2. What is Middleware?

First, some important definitions:

2.1 Directories and Identity Management

A middleware infrastructure begins with authoritative and accessible directories of people. Most universities and large corporations are now building such central directories. These directories can aggregate information about who is associated with the institution and in what capacity into a central list. For example, information about who is a student is maintained by the university Registrar's office; information about who is an employee is maintained by the Human Resources Office.; information about who are alumni may be maintained by an Alumni office.

The aggregated list serves as the core of a central **Identity Management System**. If you are listed in the directory, you are provided with an identifier (a "name" that is unique within the institution). An identity management system can also be used to provide a central authentication service; this is sometimes called "Single Sign On" (SSO) and refers to use a single username/password pair, for example, to authenticate to any number of services.
By using the one, external authentication system the user needs only one password (authentication credential); the authentication service architects can focus all their energy on building a rock-solid, secure system - it's their job and they are the most expert in this area. The application developer does not need to even handle this important credential, and can instead focus on the specific application requirements. Using this approach, a suite of applications can be combined to present a consistent experience for the end user, including a consistent name throughout all applications. In short, benefits for everyone using this approach.

2.2 The Traditional Triple-A

The traditional triple-A's for systems are **Authentication, Authorization, and Accounting**. These terms are valuable in the middleware space to understand what data is needed at different points. Some familiar triple-A's (eg: Kerberos) rely on a root authority; the emerging middleware infrastructure is based instead on Federations, which are voluntary trust relationships among a set of root authorities.

Authentication

Authentication should be viewed exclusively as the act of identifying the current user of the system, ie. the owner of the request for action. There are many possible authentication mechanisms, including: none at all (eg: a publicly accessible list archive); email address verified by reply-to; username with password, and digital certificates. In each case, the purpose of the authentication step is to associate a specific identity with a specific action.

A good example of this can be found in the Unix system environment. Here a user is authenticated by the login program. Once the identity is known it simply becomes part of the context for all subsequent processes and is transmitted as part of the action request. Subsequent processes trust the login application to have sufficiently validated the identity of the requestor and don't attempt to re-implement the services supplied by the login program.

Authentication should never be considered identical to the concept of "having an account".

Authorization

Authorization is about who can do what. Once the owner of a request can be identified via an appropriate authentication, their allowed actions can be determined. A "big picture" view of authorization is this: **Attributes + Rules --> Allow/Deny Decision.**

Attributes further describe the owner of the request. For example, the owner of the request could be part of an admin group that has certain privileges above that of an ordinary user. As an alternate and most simplistic example, the only attribute that may be needed to make a decision is the identity (name) of the requestor. It's worth emphasizing that identity is simply another attribute. From the middleware integration

perspective, this is an important point, since it enables the separation of the act of authenticating from the act of gathering attributes about the requestor. In a middleware rich environment, the authentication systems are often distinct applications that are focused exclusively on validating the identity of the user and supplying that identity as an attribute to the integrated system environment.

Rules express the policy of who can do what. For example: the person with the identity identical to that of the account owner may change the account's preference settings; the listowner can delete messages from the list archives.

Accounting

Accounting has two parts. First, there is the actual accounting for resource utilization. It is the act of logging and reporting what actions were performed and by whom. The "by whom" is the second part. In order to account for system utilization you want to distinguish different users of the system. An "account" is considered the identity of an entity to "charge" for utilization.

In most systems we think of "setting up an account" as the first step to using the system. In today's distributed Internet environment, each tool has the tendency to want to manage account creation authoritatively. Each tool typically requires the user to go through some steps to either self-register or request an account. This situation has tended to confuse authentication, identity, and account, when they are really 3 very different components. Your identity is the unique name in the central list of users; authentication is the process of successfully presenting the secret associated with your name; and account creation is the process of provisioning resources needed on a user's behalf by the application: for example, disk space called "mailbox" or a new record added to an application-specific database.

In other words, if a user is able to authenticate successfully, resulting in an identity that is authorized to have a user account, the mailing list should be able to automatically provision user accounts as needed.

2.3 "Big Picture" Summary

The middleware we're describing here really does exist. Understanding where to hook middleware into an application involves visualizing the points within the system that could potentially rely on general-purpose, external data and services. This is an extension of the familiar task of finding appropriate abstractions for procedures and data in applications. For example, knowing that a collection of usernames and passwords exists in a central database that supports LDAP makes it desirable to find the places in your application where identity is required to perform an action so that the external data source can be used directly, without replication or duplication. Middleware is an external data source that can be used across systems and across domain administrative boundaries.

The developer should keep in mind what system designers are looking for in a tool: they want a tool that will be fit into their environment. A good parallel to keep in mind is the malleability of a mature system environment such as Linux. In any application there are reasonable points at which reference to external data stores make sense. Providing a structure in your application that makes these hooks possible is the best way to ensure that a system integrator can leverage you application to the fullest.

See our "big picture" illustration of a mailing list system interacting with middleware (below) .

3. Application Use Cases

3.1 Automated List Creation and Dynamic Subscriber Listing

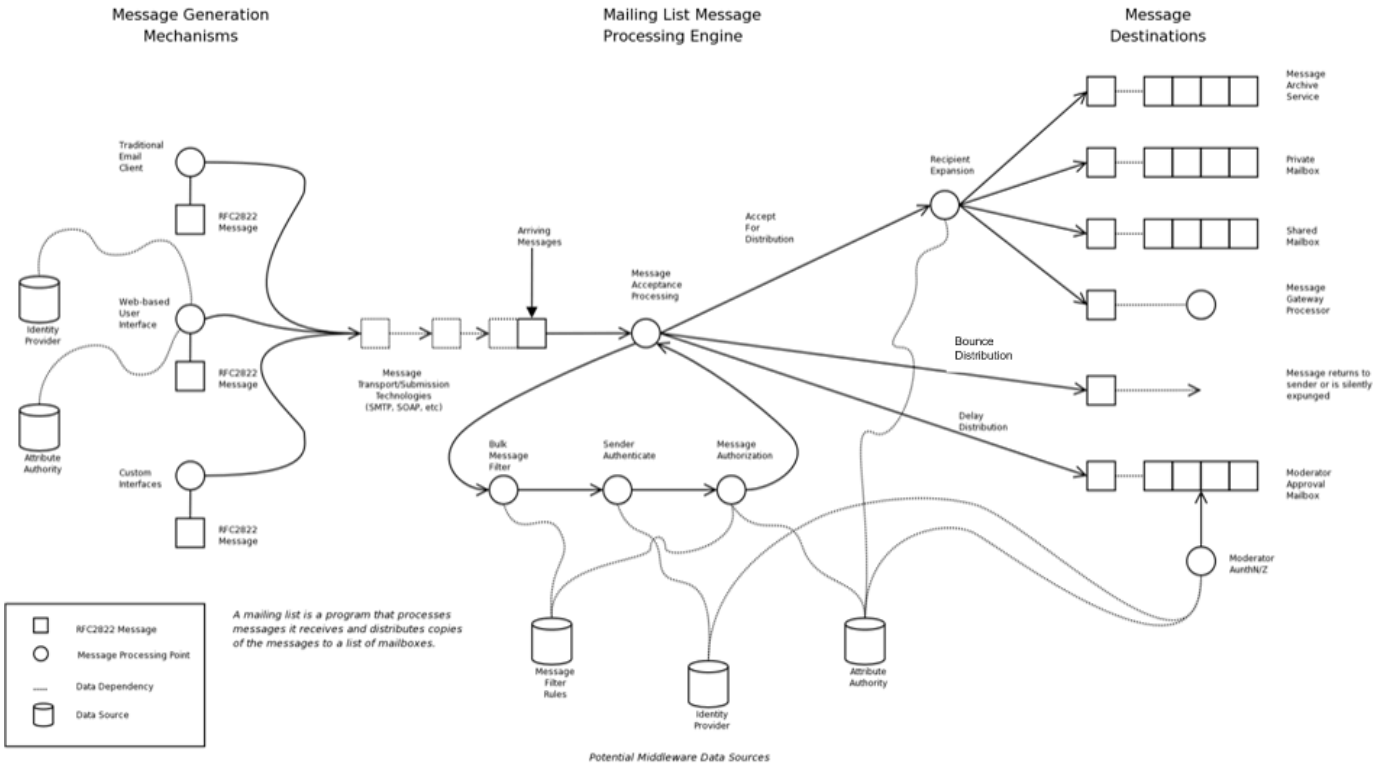
Consider a university with a central directory. The directory contains the people associated with the institution; their role(s) at the institution, such as "faculty" or "student"; and also specific course information. As a student enrolls in particular course the Course Number (e.g.: CS101) becomes an attribute associated with that student. When the student drops the course, the attribute is removed. The faculty member who is responsible for teaching this course may be assigned the same course number.

Once the course is scheduled for a particular semester, a list named something like "CS101Fall2005" could be created automatically, with the faculty member automatically assigned as list owner or list moderator. Rather than maintaining its own subscriber list, the mailing list application can query the directory for the email addresses of the students who are enrolled at the time a course announcement needs to be sent to the class. The figure labeled "External Data Store" (Page 6) illustrates mailing list functions that could interface with external data stores for all the capabilities described in sections 3.1-3.3 of this document.

3.2 Shared Group Subscriptions

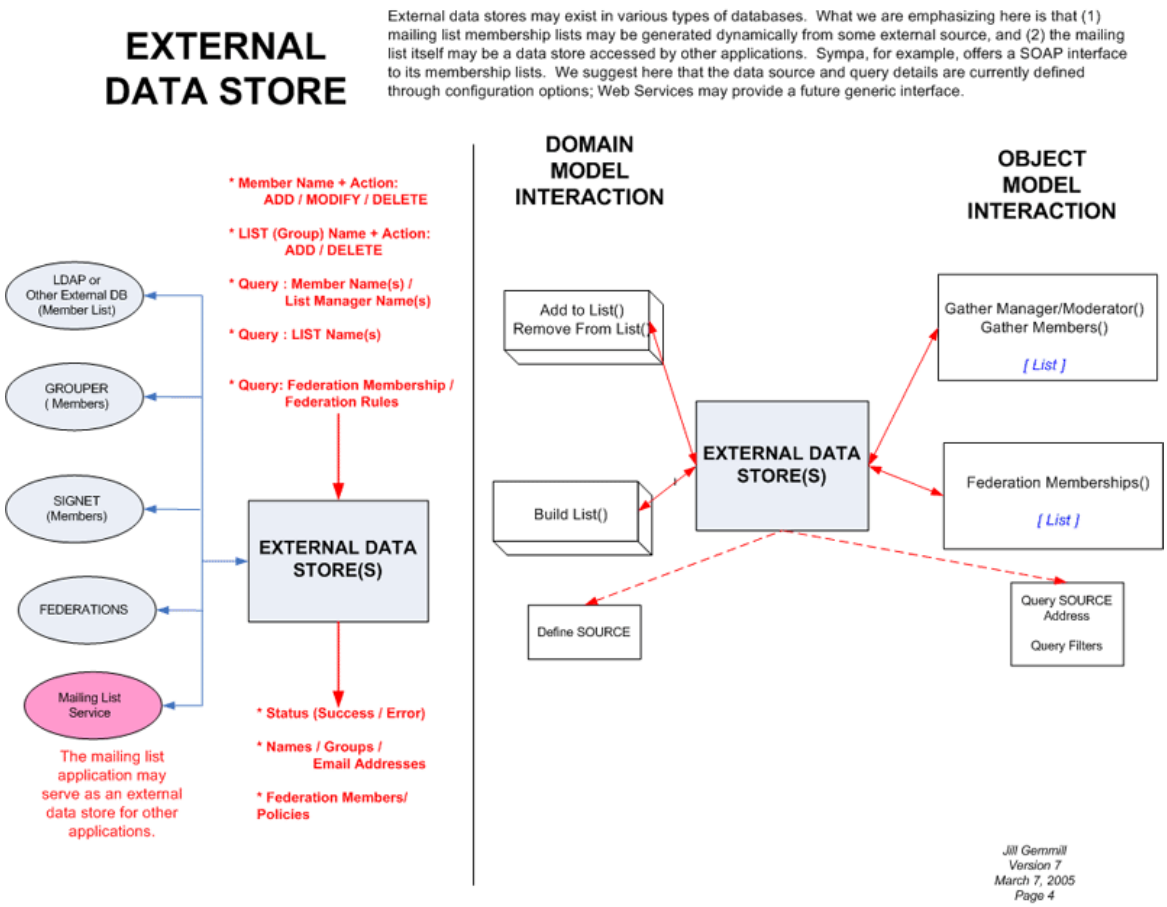
Multiple organizations may be interested in sharing a common list. For example, suppose the chancellor in charge of a multi-university system would like to send communications to the faculty at each of 6 campuses. These campuses are so large that each one has its own directory. It would be desirable to create a chancellor-level list with only 6 subscribers: University1Faculty, University2Faculty, and so on. Each subscriber address is converted into the set of email addresses returned from a query to the appropriate directory for its list of current faculty members.

The communications are not intended for the public at large, so the list archives which are available via web interface need to be appropriately protected. As the faculty browse to the location of this web archive, they are routed by middleware to home institution authentication (or verification of some current session credential), followed by release of their "Faculty" attribute. Then they are permitted to access the mailing list software, and they can do so without having to be subscribed to the list. A student at one of these



[*Link to larger version of diagram*]

universities tries to access the archive; she can authenticate successfully but without the necessary "faculty" attribute, her access is denied.



[*Link to larger version of diagram*]

3.3 The Mailing List as Dynamic Source for Attributes

In a truly distributed environment each application may become the source for information in addition to querying other sources. For example, suppose you wanted to make a wiki available for writing to only those people subscribed to the Mailman Developer list. It should be possible to query the mailing list, obtain each subscriber's identifier, and provide those identities with the desired access. The subscribers, having authenticated elsewhere, would be automatically logged in upon arriving at the wiki. Sympa open source mailing list software, as an example, provides a SOAP interface for this purpose.

4. Modeling the Mailing List / Middleware Interaction

Recall our "Big Picture" diagram, above. We have developed two more detailed models of the application interface with middleware, again using the mailing list application as an example.

- The "Domain Model" represents a non-ordered process flow and identifies middleware interaction points. This model was developed primarily by Jim Phelps (U Wisconsin). It is available here:
  The Domain Model http://middleware.internet2.edu/mlist/docs/draft-internet2-mace-mlist-domain-model-08.pdf
- The "Object Model" represents the hierarchical nature of the mailing list application (application administrator; multiple mailing lists; multiple subscribers/moderators/administrators per list; etc) and identifies middleware interaction points. It is available here:
  The Object Model http://middleware.internet2.edu/mlist/docs/draft-internet2-mace-mlist-object-model-02.pdf

We found that both models identified similar middleware interactions, and that it would be useful to examine that interaction in further detail. In section 8.3 you will see our Domain Model used to describe the architecture of Sympa, an existing middleware enabled Mailing List application. The Sympa developers found this model to be an excellent fit.

5. Middleware API for Applications: A Preliminary Description

5.1 Message Filter Rules

Message filtering includes functions such as anti-virus checking, spam-tagging, checking for valid originating server etc. etc. These services might be considered pre-processing, and might not event be part of the application. The pre-processing might determine whether or not the incoming message was even sent to the mailing list application at all, or might modify the message sent. If, however, the application did directly interact

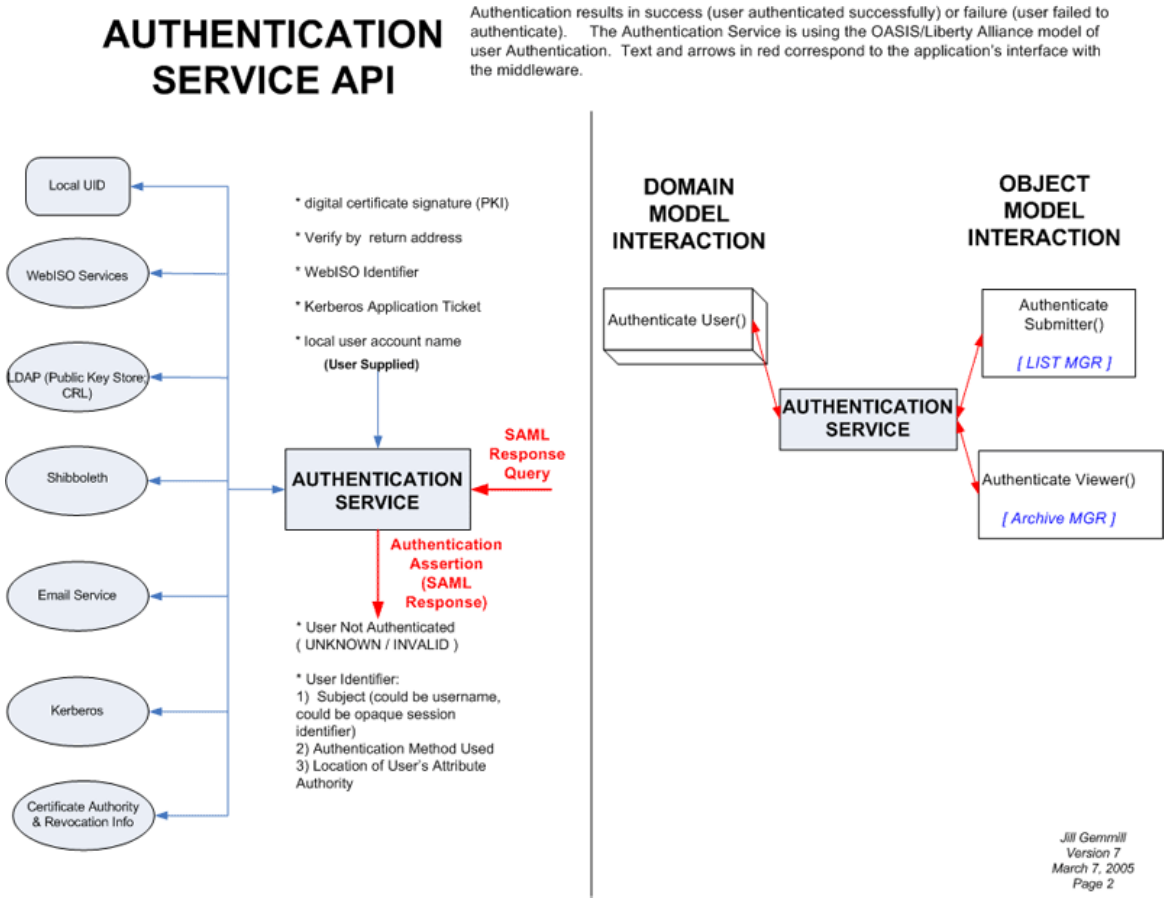
with one or more filter services, the interaction might be modeled by this Filter Service API diagram
http://middleware.internet2.edu/mlist/docs/draft-internet2-mace-mlist-middleware-interaction-focus-04.pdf

5.2 Federated Identity Providers allow "Generic" Authentication

The Liberty/Shibboleth approach of **federated administration** means that authentication can be distributed among many identity providers and that the authentication method does not need to be the same at each provider. One may use Kerberos tickets, another may use username/password. Typically, an authoritative directory of some type exists at the home institution in which all persons affiliated with that institution in some capacity are listed along with their type of affiliation. Each Identity Provider authenticates the user at their "home".

This architecture means that the application does not need to be able to handle each type of possible authentication protocol. Instead, through a previously established trust agreement, the application trusts that the home institutions correctly identify their members. The application just needs to know that this user has been authenticated; this information is provided by an Authentication Service. In the figure below we show the use of digitally signed Security Assertion Markup Language (SAML) assertions to conduct this exchange.

It is **VERY IMPORTANT** to note that the authentication service does not return any identifying information; only that a successful authentication occurred. "Useful Information" will be provided via Shibboleth, the attribute transport service. (see the Session State Diagram at the end of section 5).
A working implementation of this authentication service is available to applications through the WebISO and Shibboleth projects. WebISO takes care of each institution's home authentication; Shibboleth provides a **Target** which is able to receive the necessary information. The first information the target receives is indicated in the "Authentication Service API" figure.



[*Link to larger version of diagram*]

5.3 Resources

Identity Management
About Identity Management - Standardized Directories
see http://middleware.internet2.edu/core/directories.html

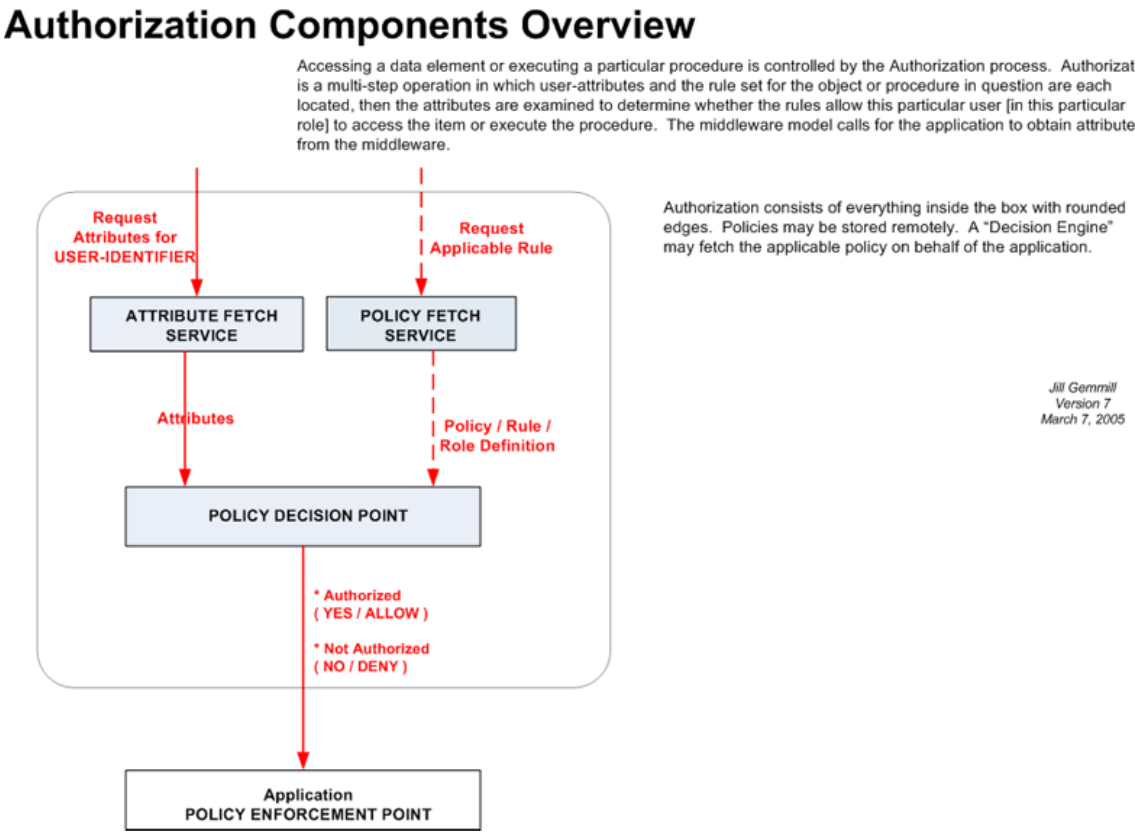
Authentication
About Authentication using Web Initial Sign On (WebISO)
See http://middleware.internet2.edu/core/authentication.html

SAML (Security Assertion Markup Language)
About Open Source Security Assertion Markup Language
See http://www.opensaml.org/

LDAP (Lightweight Directory Access Protocol)
Introduction to LDAP http://metric.it.uab.edu/vnet/cookbook/v2.0/node28.html
LDAP Resource List http://metric.it.uab.edu/vnet/cookbook/v2.0/node129.html

5.4 Authorization in Federations (Shibboleth)

Authorization is a complex process in a distributed system. In addition to user identity, user attributes (or roles, such as "list moderator") are defined and maintained inside the application, along with the rules (or policy) and also the process by which the rules are applied. The "Big Picture" of Authorization is that the appropriate rules are found, the current user's (or process's) attributes are examined, and the application decides whether this user (or process) has permission to execute the requested action. In our model, this is called the Policy Decision Point. It is entirely possible that the application could send the appropriate attributes to some external application and wait for a decision. An NMI-EDIT (NSF Middleware Initiative - Enterprise and Desktop Integration Technologies Consortium) component named PERMIS is one example of an external access control decision engine that works in this manner. See http://www.nmi-edit.org/releases/index.cfm#Permis



[*Link to larger version of diagram*]

An overview of the complete authorization process is illustrated in the "Authorization Components Overview" figure. The text following this figure explains each subcomponent in further detail.

Shibboleth Resources

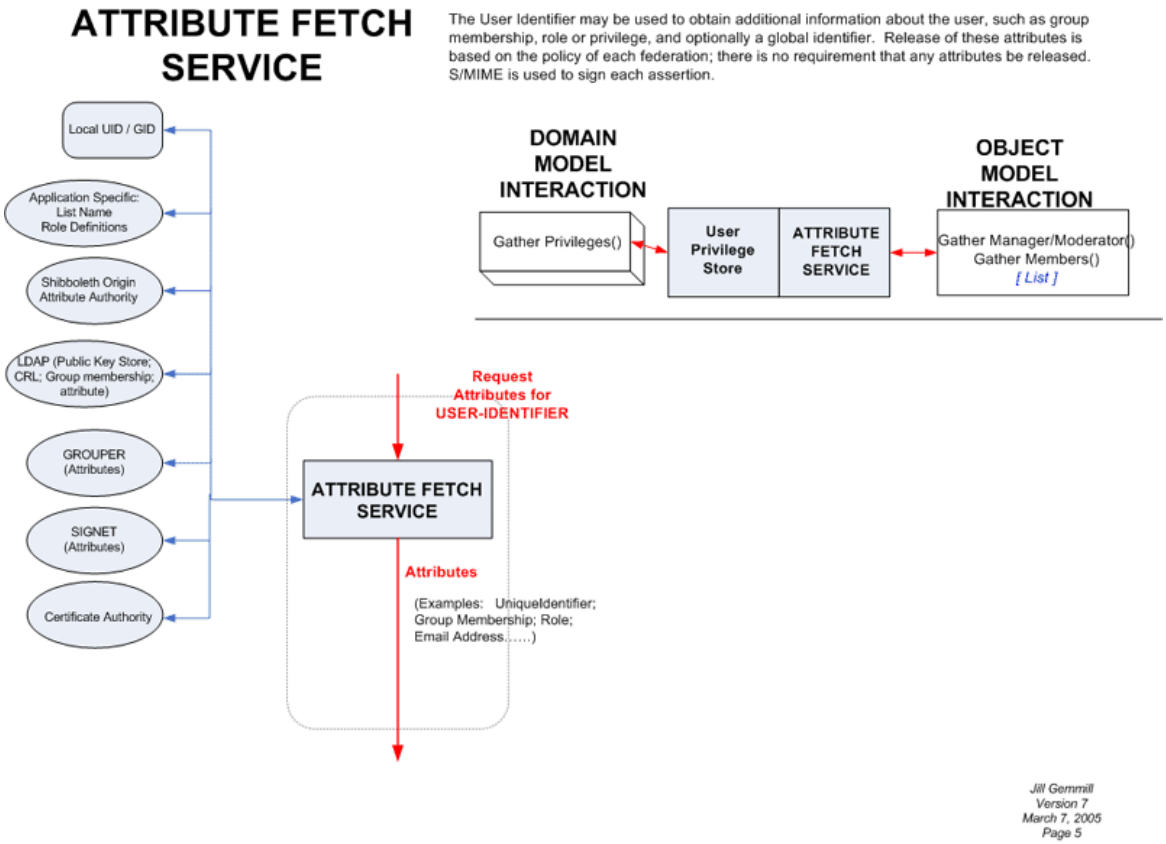
About Shibboleth (Federated Attribute Transport)
http://shibboleth.internet2.edu/

5.5 Attribute Fetch Service

The user identifier that is available may be a global user identifier or some anonymous "handle" that a home institution can use to send further information without necessarily revealing identity. Policies about what information is provided when is developed by agreement among the participating parties, who are sometimes referred to as a "Shib Club". The user identifier might be as transient as a session ID, or as persistent as an LDAP distinguished name (a globally identifiable and searchable name). **Attributes** describing the user may be requested by using the identifier. (See "Attribute Fetch Service" diagram, below)

Some things to keep in mind about attributes:

- The user's name (even a globally unique name) is an attribute.
- Email address could be an attribute that is released by the enterprise; email address might look identical to the identifier; but email address should never be used as an identifier for members of federations. In an application like the mailing list it is probably a good idea to use the value of the user's email address (if provided) as a suggestion for the user's preferred email address; the authenticated user could have the option of modifying this suggestion. This approach allows users to have a single identifier and still choose separate email addresses for each list if they like.



[*Link to larger version of diagram*]

Benefits to the application developer from supporting this approach are: (a) the identifier can be used to collect all lists subscribed to by a single user, even if that person uses a separate address for each list; (b) other applications that have received a user identifier in the same way will have the identical user identifier available; this is useful to system integrators.

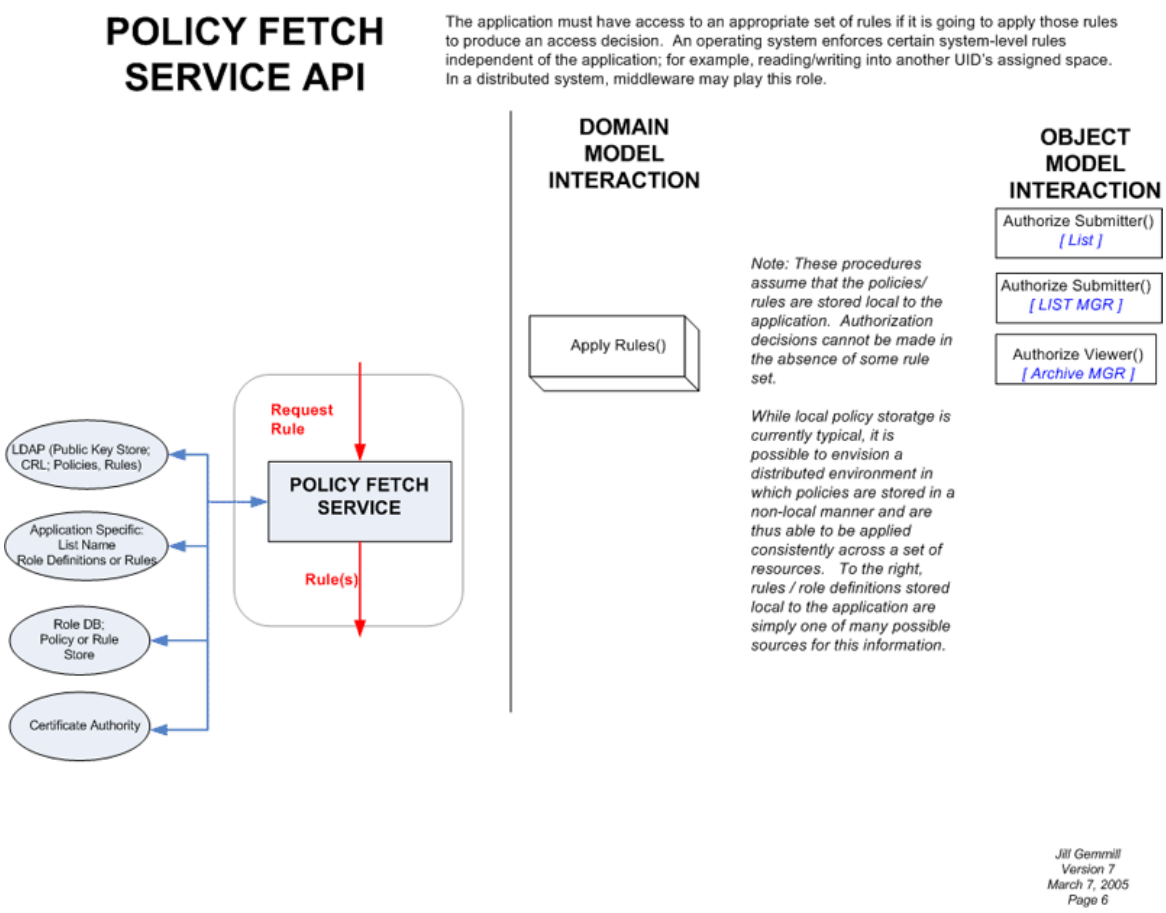
5.6 Attribute Authority

The Attribute Authority resides with the Identity Provider. Once the user authenticates at their home institution, the attributes associated with that person and stored in the authoritative home directory can be made available. Whether the information is released or not should depend on a policy established by prior agreement. For example, the authenticated user at a university may have attributes "staff, student", representing an employee enrolled in a course or perhaps a student with an on-campus job. For some applications, these attributes may be more useful than the person's name; for example, to allow access to on-line library materials that are licensed to a university for use by faculty only.

One attribute that could be released is the user's global identifier. Current Shibboleth code passes this information as an environment variable that is available to Apache. The next version of Shibboleth will support non-web-based transfer of this information.

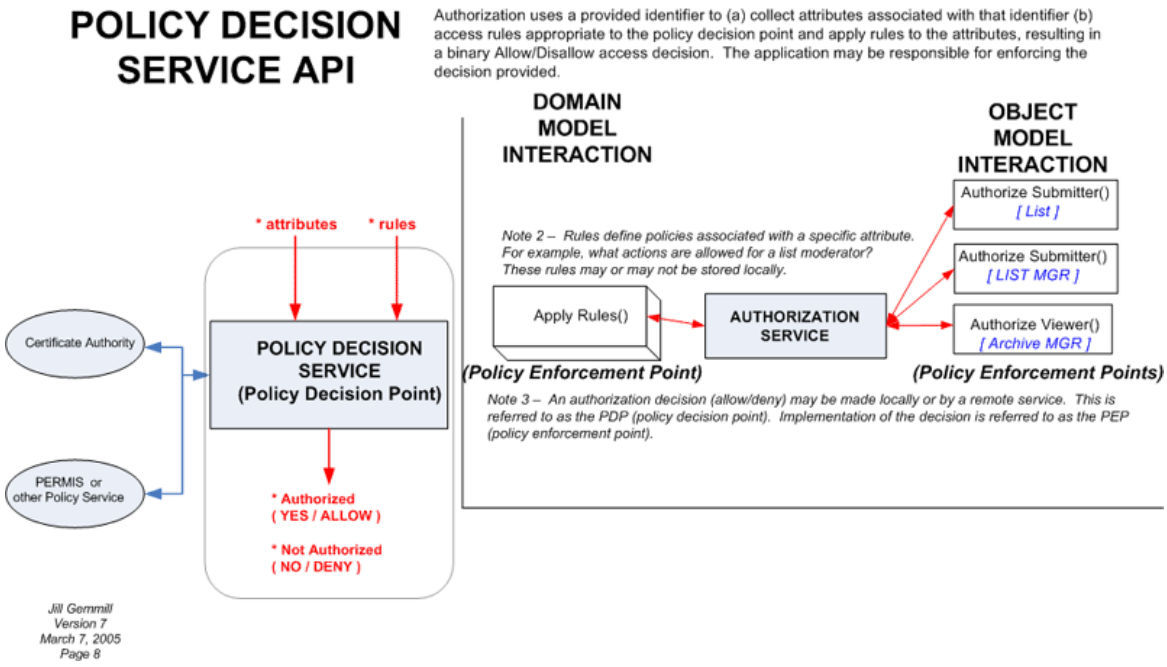
5.7 Policy Fetch Service

User attributes (group memberships, roles, identifier) are used in authorization by examining the set of attributes to determine if the requirements of the appropriate rule of policy are satisfied. The important point to remember here is that policies (roles, role definitions) may be stored in a remote database. (See Policy Fetch Service API figure)



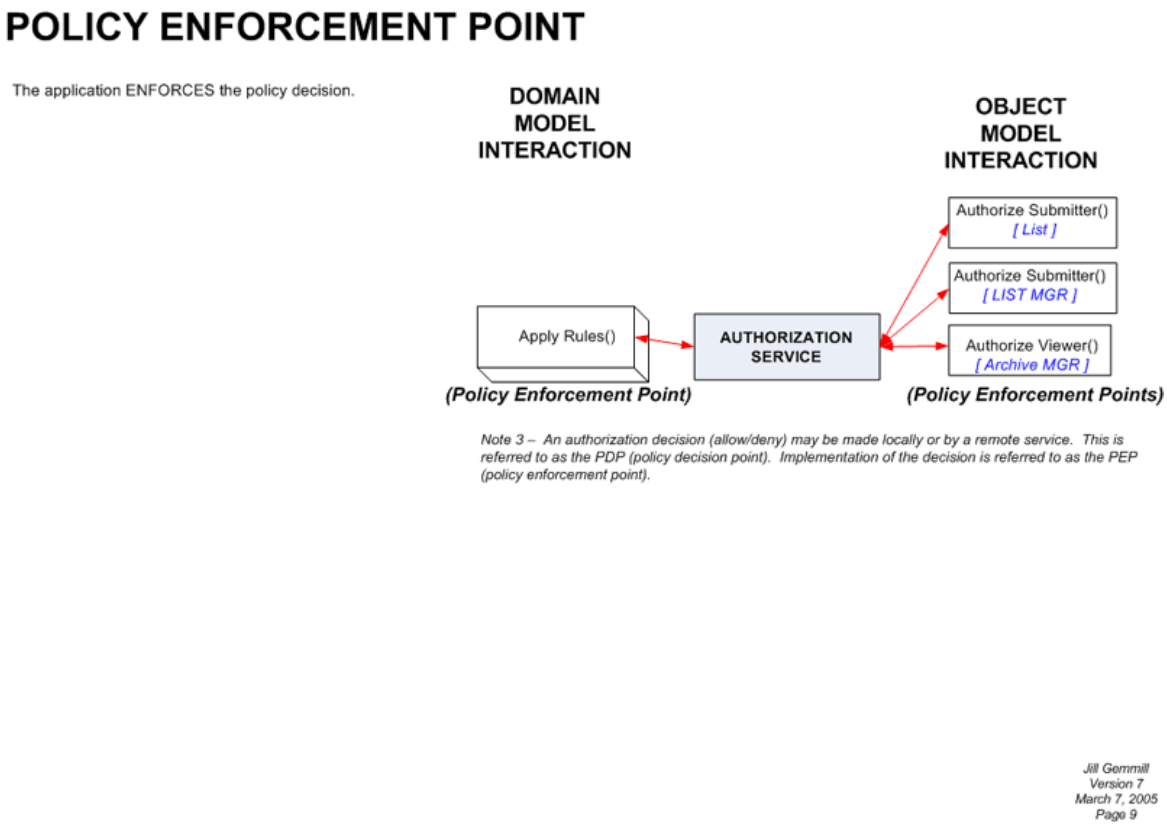
[*Link to larger version of diagram*]

5.8 Policy Decision Point
In our model, the Policy Decision Point occurs at the point that an "ALLOW" or "DENY" decision is made based on examining the rules and available attributes. Keep in mind that this decision point could exist outside your application.



[Link to larger version of diagram]

5.9 Policy Enforcement Point
The application enforces the ALLOW / DENY decision.

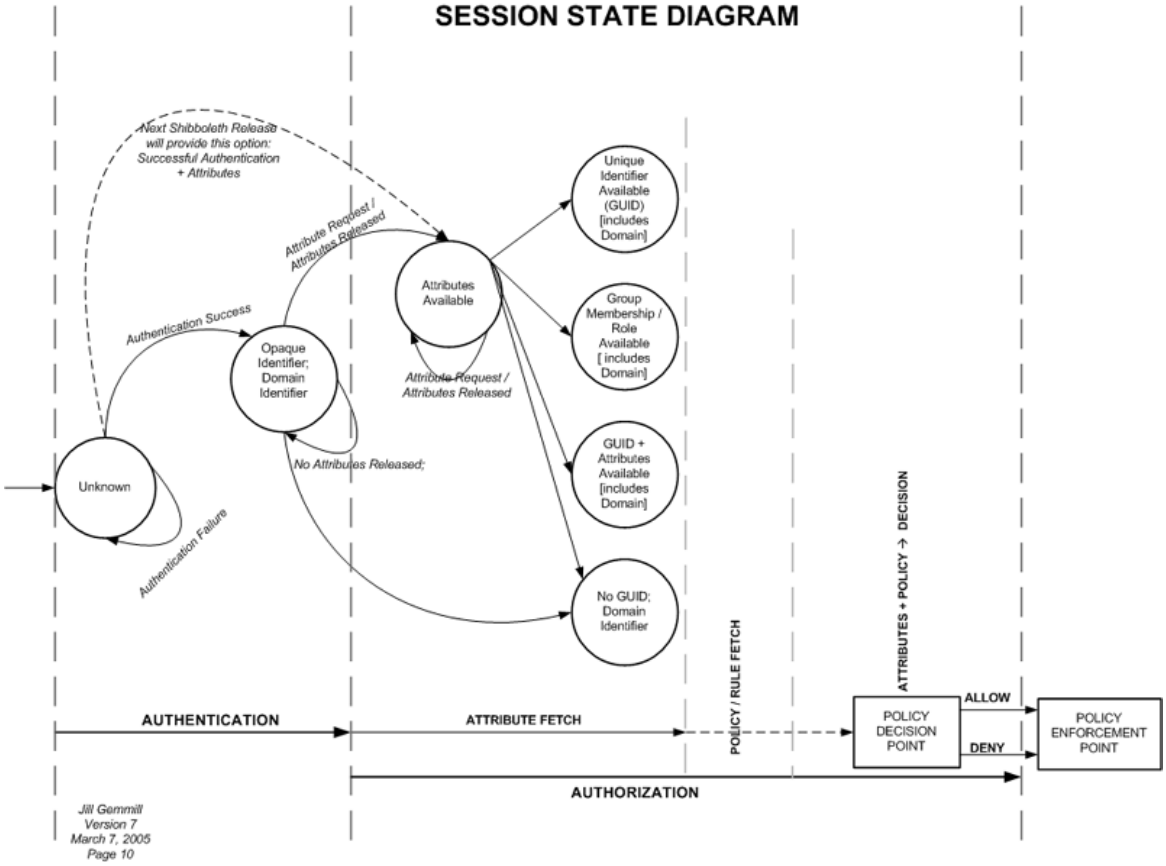


[Link to larger version of diagram]

5.10 A Summary of the Entire Authorization Process

It is common to directly implement specific authentication and authorization protocols in an application. However, experience suggests that greatest flexibility comes from relying on the modules from the authentication/authorization developers to implement the protocols with full attention to secure handling of credentials and other security issues, and have the application simply obtain this information from those services. In the current implementation of Shibboleth, this is simply inheriting the information via the CGI environment using the standard trust relationship with the web server. Most authentication/authorization technologies provide code to build modules for your web server and documentation on how to hook them into the namespace served by your application.

The figure labeled "Session State Diagram" provides a summary of session state from the first time of arrival at the target site (the application) through policy enforcement .



[*Link to larger version of diagram*]

6. Application Design Consideration and System Requirements

6.1 User Identifiers

Building a mailing list system that spans application and administrative boundaries requires that we define a consistent identity for users. This identifier can take many forms.

- The identifier can be a simple, unstructured string, eg. username
- The identifier can be a structure string, eg. user@domain.
- The identifier can be inherited from another system environment, eg. mailbox@maildomain.

Mailing list software is primarily concerned with distributing copies of a one message to multiple mailboxes. Considering this function, it is natural for the mailing list software to treat a mailbox as an identity within the system.

This perspective leads mailing list software to focus on per-destination mailbox subscription settings. If each mailing list is treated as a distinct group, then each destination mailbox may have distinct subscription settings. If one end-user with a single mailbox is subscribed to many lists, this can lead to configuration complexity for the subscriber. Some mailing list software introduces the concept of a parent account under which all these subscriptions can be managed. This offers some configuration enhancements to the end user.

Determining how to integrate these various approaches with middleware can be complex and there is no one right answer. The requirements for a specific mailing list implementation will often depend on which system environments the developers intend to support and how they perceive the integration points within that system.

6.2 Inheriting User Identifiers from Other Systems

In many mailing list applications and their associated web interfaces, the user identifier is synonymous with an email address. These tools are primarily concerned with distinguishing between individual users and communicating with those users. The email address conveniently does both. It is both an identity for the user and a communication interface with the user. This is not a bad practice, however, it's important to recognize the overloaded functionality of this identifier.

The email address, though, is an identifier inherited from an external email system (ie, the system hosting the user's mailbox). Inheriting the user identity from the email address can be problematic, however, unless the mailing list system environment can make assurances about the validity of an email address as a communication interface. It is not possible to immediately assume that because an identifier looks like an email address that it is.
Verifying the validity of an email address is a familiar process. The application sends a confirmation message to the address and if the user responds within for the request times out, then the application can be assured it has been given a working communicating interface to the user. The uniqueness of the mailbox@maildomain syntax also gives the application an identifier it can use to distinguish between communicants.

Note, the email address may not represent an individual person, but it can still be considered a unique identifier for the purpose of controlling the subscription preferences.

6.3 Inheriting User Identity from Middleware

In a middleware enabled system environment, the identity of a user is defined and verified by applications external to the traditional mailing list software. In this environment, the mailing list software needs to establish trust relationships with the external systems that supply the user identifiers.
It's important to recognize that the mailing list software can become constrained by system policies defined by these external applications and that this can cause conflict between the assumptions of users accustomed to the traditional, stand-alone tool sets and the requirements of the mailing list software to respect the policies of the system environment defined by the trust relationships with its data suppliers.

The degree to which these external systems influence the behavior of the mailing list system should be left in the hands of the system designer. The ideal mailing list software would enable a system designer to dictate the degree of flexibility the mailing list software should offer the users of the system. Let's consider two extremes to illustrate this point.

In a **tightly integrated** middleware enabled system environment all data generation and validation decisions are made by a central authority. The mailing list software is no longer authoritative for any of these tasks; this scenario is likely to be of greatest use for official intra-institutional communications.

In a **loosely integrated** middleware enabled system environment, such as is commonly found in federated environments, multiple external systems are trusted to supply user identity and other attributes, such as the email address. While the federation will usually supply an authoritative statement of identity for individuals, there may be little additional data which has consistent definitions across members of the federation.

In general, applications need to be willing to inherit user identity for the containing system environment. When you make an application capable of integrating with a larger system environment, it simplifies system integration if you allow your application to take on the user identification standard of the newly defined system environment rather than forcing the system environment to adopt or accommodate the user identification conventions of your application. This is an area of conflict for application developers, and, as usual, there doesn't seem to be one easy answer. Adaptability to integration, however, does improve the ease with which a system builder can integrate your application. Let's consider a few relevant examples.

- If you have a web application that insists on a case sensitive and forced mix of upper & lower case characters in the user id (as in a wiki application) it may be difficult for that system to seamlessly inherit an external user identifier, eg. via REMOTE_USER.
- If you have a well established system environment, it may be difficult to force that system to adopt the conventions of the newly defined system environment. For example, a mailing list application is often built around an identifier of the form user@domain and a *nix system environment is built around an identifier of the form username. In both of these cases it would be difficult to adapt these tools to a new naming convention so some flexibilty and creativity is needed. A *nix system could adopt the user@domain convention by considering the unspoken addition of the unix host's domain name. However, this works only if the domain component is consistent for all users, ie. the unix system is dedicated to a single domain. Translation of the user id might be necessary, eg. user@domain ids from the mlist could map to user-domain ids in *nix. This translation convention would then need to be remembered when accounting.
- It seems to come down to determining which system environment you consider primary. It's easy to want to consider our own tools as the primary environment, but in a world where integrated systems are built by leveraging middleware and transcend traditional system boundaries, your application may play either role. It may be the primary environment which defines user id conventions or it may be a secondary component which inherits the conventions of its containing system environment.

This makes it necessary to offer more configuration options for the system integrator or end user and to enable some of the traditional data verification procedures of the stand-alone environment. For example, it may be desirable for the end user to supply alternate email addresses directly to the mailing software which must first confirm their validity as with traditional environments. In this case, the data needed for the mailing list software to operate essentially comes from two places: the user identity is supplied by an identity provider but the additional data (like email address) is supplied directly by the end user.

The degree to which this flexibility is supported is up to the mailing list software developer to determine, however, the decisions made may limit the deployment of the tool to certain situations or may require that additional external system management tools be put in place. It is up to the system integrator to judge what level of integration is needed and possible.

6.4 Subscriber lists

The subscriber list can simply represent input data to the mailing list software or it can be integral in determining the user identities of mailing list

users, depending on the implementation of the software and role of the mailing list. Subscriber lists can also be used during message acceptance processing to determine if a user is authorized to post to a particular list. We'll consider that use in the next subsection. In this subsection we'll consider the subscriber list in the context of message distribution.

In an announce list, where many people are subscribed to a list but few are allowed to send messages to the list, the subscriber list is data consumed by the mailing list software in order to perform message distribution. In a discussion list, where the list serves as a communication form between subscribers, the subscriber list plays a dual role as data consumed during message distribution and as a component of user identities recognized by the list. There are many variations on this, so even announce lists may treat the subscribers as light-weight accounts to support preferences. The point isn't to show all the possible combinations, but to highlight two uses of this data within the mailing list system for this data.

Mailing list software should be capable of dynamically gather the list definitions and destinations addresses for subscribers from a multitude of sources. The middleware technology most relevant here is LDAP, but using SQL databases and other custom data providers adds flexibility.

The list software should recognize that it may not be authoritative for controlling this data and should separate the gathering of subscription addresses from the management of those addresses. Considering that a data source which supplies group names and member email addresses may not be geared toward serving all the needs of a mailing list, there may be some data that the mailing list software is authoritative for, like individual subscriber preferences. The software should not demand that all data be stored in the same place. It may be necessary to merge middleware data with local data in order to resolve the final delivery preference.

6.5 Authorization: attributes, rules, and decisions

The next major area where middleware can play a vital role is with the authorization processing for message acceptance. This is an emerging area of integration. Today's solutions are likely to be specific to each mailing list software package, since standard definitions for authorization decisions don't yet exist. The ability to control message processing externally, however, is a powerful extension of the middleware technologies in the user identity and subscriber list definitions. A few examples may help illustrate the point.

- The subscriber list determines who can post to a list
- Determining the identity of the message author may rely on a simple =From: header address matching, confirmation message approval, or require S/MIME signatures. (Please see the Autentication and Authorization description for Sympa as an example.)
- Message approval processing may be controlled via a web interface with middleware authenticated moderators or direct web-interface composition of the message
- Message approval may also include content filtering mechanisms.
    - SPAM analysis is normally provided externally but it would be desirable to let regular list moderation processes to be triggered to review potential SPAM.
    - Technologies similar to SPAM filtering could be integrated during message analysis to help ensure that messages posted to the list adhere to established content preferences.

In all of these scenarios existing middleware technologies for user identification, including attribute collection, could be leveraged to control web-based user interfaces to these processes. For example, members of an institution that are faculty and associated with specific course may automatically be granted moderator access to the mailing list for that course. Their identity and role could be readily determined by the web interface to the moderator queue. These examples are intended to highlight potential uses for middleware data in order to stimulate software designs that accomidate this level of integration.

6.6 Other Integration Concerns

Creating an integrated system touches on all aspects of the environment. While outside the scope of this document, application data and user interfaces integration is important. The following comments only serve to introduce the issues.

6.7 Application Data: Messages

Mailing list systems are somewhat lucky in that they are based on a common data exchange format and mechanism as defined by RFC2821 (SMTP) and RFC2822 (message format). This makes data integration across systems fairly simple if one accepts replicating the data into application specific data stores.

This allows our focus to be on sharing identity and authorization decisions across system boundaries.

6.8 Visual Elements: User Interfaces

The focus of this document is the middleware that enables sharing identity and authorization decisions across system boundaries. Users will often consider integration to be incomplete unless the visual elements of an application are consistent across the toolset. In the web world, visual elements can be controlled with consistent graphics and stylesheets, given customization options in the web interfaces. This integration is outside the scope of the middleware technologies considered in this how-to.

6.9 Logout in Middleware context

6.10 Common Problems

- Storing an account in the same place as the authentication credentials

Frequently, developers assume that if identity can be determined via an LDAP password verification, then the account should be stored in the same place. Keep your authentication verification and account definition separate.

7. System Authorization Models

7.1 Account-action Model

These systems define a (sometimes long) list of actions that can be performed in the application, like "post to list" "moderate" "subscribe" etc. They then go about associating accounts with the actions they can perform. When per-account assignments become cumbersome, the tools typically move to a role based assignment of authorizations that allows you to assign roles to actions and accounts to roles. At the very least, this keeps you from having to modify a lot of accounts if you decide to change authorizations. You just modify the role.

7.2 Object-owner Model

The other angle from which to approach authorization is to define object-owner authorizations. This requires the system to define the internal objects that can have owners. You can then give owners (accounts) permission to control the object they own, eg. a mailing list or subscriber list. You can obviously introduce roles (or groups) into this approach in an effort to help manage the complexity of object ownership assignments. You then assign accounts to groups and groups as object owners.

7.3 Hybrid Model

You can also combine the account-action and object-owner models and get various degrees of flexibility and/or complexity. For example, you could define the accounts "joeuser" and "moeadmin", the action roles "read" and "write", the owner group "admin" (with moeadmin as a member), and the object "subscriber list". You could then describe permissions like "joe user is allowed to read the subscriber list and moeadmin is allow to read and write the subscriber list".

7.4 Middleware Feeds the Model

The authorization model in your application can then become fairly flexible if you promote your actions and data abstractions to the same level and have your authorization system worry only about enforcing the owners and actions for each object and not worry about what the type of the object is (ie. what it contains).

For example, a fairly simple system could add the action role "execute" for objects to the definitions in the previous example. Execute gives the owner of that object the permission to "set the object in motion" (so to speak). In this system you could define the objects "moderate", "mailing list A", and "mailing list B". The admin group could be given execute permission on the "moderate" object. The account "joeuser" could be added to the admin group and given ownership of "mailing list A" object but not of the "mailing list B" object. With this, you could describe something like "joeuser can moderate any mailing list that he owns".

The point here is not to get involved in the design of the application, but simply to recognize some fairly common system authorization models. There is no single correct model to choose from. The important thing to realize is that the data supplied by middleware technologies feeds these authorization decision processes. Recognizing the model your system environment follows will help make it easier for you to find the correct points at which to integrate middleware in your application

8. Case Study: Sympa, a middleware enabled list management application

8.1 What is Sympa?

An excellent description of how to design mailing list software for middleware (authentication and authorization, specifically) has been written by the developers of the Sympa Mailing List open source software. The specific section of interest for triple-A is AA in Sympa.

Sympa is a rich open source mailing list software. It is well known in Europe and has recently become of greater interest to mailing list managers in the US because of its early adoption of middleware interfaces.

Sympa's feature set includes

- Bulk emailer
- Internationalization, translated to 15 different languages
- Service messages and web pages defined by templates
- Subscriber and list owner information stored in a RDBMS (Mysql, Pg, Oracle or Sybase)
- Web interface with user and admin features
- Different web authentication back ends including LDAP and Single Sign-on systems
- Possible definition of list members using external information system
- Web document repository for list members
- S/MIME support for both signature verification and mail encryption (Presentation at Linux Expo Paris - February 2001)
- Automatic bounce management. See http://www.sympa.org/documentation/article_smime/sympasmime.html
- External antiviral plugin
- Virtual robots management
- SOAP XML service

8.2 Sympa Mailing List Manager Middleware Integration Points

Authentication

Sympa supports multiple authentication methods including

- Internal password allocation by email (as in many web applications)
- LDAP authentication
- Generic SSO authentication : a way to configure Sympa to be used with various SSO system.
- Can be used withShibboleth
- Specific SSA method for CAS authentication
- https strong authentication based on user certificate

These authentication methods can be added to Sympa by editing a configuration file. Multiple methods can be used in parallel. This is important because list services often include users from different institutions having different local infrastructures.

A complete discussion of Sympa's approach to authentication/authorization can be found here:
AA in Sympa. http://www.sympa.org/documentation/AA-in-Sympa/

List member definition

Most mailing list managers require that lists are created using manual procedures, and that user-subscription or administrator-addition are the only methods for adding list members. Sympa allow s automatic definition of list members sets using attributes extracted from external information systems.

The methods currently supported are:

- Include other lists
- Include email from SQL query
- Include emails from LDAP search
- Include output of some remote web access (including any cgi that produce a list of email)

Both manual subscription and any of theses methods can be used for each list. List owner and list moderator can be defined in the same way.

List family

List *family* is a way to define a set of list using a model and an XML definition of variables that are used as list attributes. It is designed to manage a large number of dynamically created lists. For example : create a list for each course and subscribe all students enrolled in the course; automatically create or remove alist when a new student category is created or deleted; automatically change the list owner when the instructor is changed, etc.

Sympa authorization and middleware

Sympa includes a language for sophisticated definition of privilege that can be applied to all MLM services (subscription, archive, message sending, hide or show some private list, etc). This mechanism can use attributes inherited from the authentication mechanism used or from LDAP query filters.

Sympa SOAP interface

Sympa's soap interface has been designed to provide a service oriented API to MLM in order to provide a nice integration into the global services of a network infrastructure. This is the method used to provide a Uportal canal for Sympa, which is available from the esup portal project (consortium esup-portail) .

Sympa SOAP services are also a way to contribute to the middleware infrastructure of your network; the Sympa server can be used to provide user authentication (requesting password validation) and/or authorization (group properties) by querying the Sympa server. This has been implemented recently in apache::authsympa which is a module that provides authentication and/or authorization based on SOAP requests to Sympa. The goal of this apache module is to give authorization to any resource served by an Apache httpd server based on whether or not a user is subscribed to some particular list.

8.3 Applying the MLIST Domain Model to Sympa

The Sympa developers found our Domain Model to be a generally good fit to their application which is advanced in its degree of middleware integration. Their comments on the "fitting" process are contained here.

8.4 Some further references about Sympa

- Sympa's Home Page (www.sympa.org)
- "An Open Source Middleware-Enabled Mailing List Server", Presentation on Middleware Integration by Sympa developers at Fall 2004 Internet2 Member Meeting.
- A comparison of Sympa and Mailman (Is someone candidate to update this comparison?) http://www.sympa.org/documentation/mailmanvssympa.html

9. Additional Thoughts - Desirable Features for List Management Applications

There are many features in a mailing list system that are not required for the mailing system to operate, but that are integral to a smoothly operating system environment. Middleware integration can enable additional functionality for the mailing list application.

9.1 Multiple Interfaces

It is desirable to have multiple interfaces into your mailing list system. These interfaces can take the form of many different applications that may use any protocol that can transfer and receive information. The interfaces will need access to the same vital information such as the names of the users, the names of the lists, and the knowledge of which users are subscribed to which lists. Each of these applications needs to have access to the same information in order to provide consistent authorization decisions; it should not be necessary for all of the applications to be running on the server where the vital data is stored. In fact, each application could be on separate physical networks separated by firewalls and other barriers to collaboration. Middleware allows these applications to circumvent these barriers and provide consistent results to the users. The following are several examples of interfaces that may be desired in a mailing list system.

- To post to a mailing list most users would use a standard email client to send a message to the mailing list. Other users, however, may

prefer to use a forum, or bulletin board, to access the mailing list. It is possible that a mailing list could be viewed in a forum-type web interface where one could post to the forum which would, in turn, send a message to the mailing list on behalf of the user posting.
- Retrieving archived messages is a large part of what comprises a mailing list system. Archives for a list may be retrieved in many different ways. A user may wish to read archivies by sending a directive to the list through an email client, by accessing a forum that provides a view into the mailing list, by using a news server (nntp), by subscribing to an RSS feed, or by using IMAP. What is important is determining who is allowed to access the archives.
- There are several other examples where different views, or interfaces, can be opened into the mailing list system.

9.2 Transparent Provisioning

A mailing list application should provide transparent provisioning of accounts when a user subscribes to the system. In other words, the application should populate application specific data with reasonable default values to establish an "account" within the system. A user should not be required to proceed through a lengthy registration process. This provisioning also should not require (unless requested) additional steps by an administrator in order to complete the creation of the account. This is an extremely desirable feature in a middleware-enabled system because of the fact that the user is not authenticated at the application level.

9.3 Interface with Essential External Systems

When individual lists are created, it is a convenience if the mailing list system has the ability to interface with other systems (such as the mail transport system) in order to set up mail aliases and other such data needed to finalize the creation of the list. Many systems, at least in the past, required administrative intervention or, at the very least, some scripting to generate data. This was due, in part, to the variety of mail transport agents that could be used.

10. Acknowledgements

Additional authors of this document are Jason L. W. Lynn, University of Alabama at Birmingham, and also Serge Aumont, Comité Réseau des Universités. This document would not have been possible without the significant contributions of Jim Phelps of the University of Wisconsin and members of the MACE-MList Working Group (http://middleware.internet2.edu/mlist/). Thanks to UAB graduate students Prahalad Achutharao, YiYi Chen, Silbia Peechakara, and Song Zhou for contributions to authentication/authorization integration issues.

The authors wish to thank Internet2 and industrious flywheel Steve Olshansky for staff support of the MList project.

11. Glossary of Terms

Account:
An **account** represents a set of resources needed by an application to function on behalf of a user: for example, disk space called "mailbox".

Accounting
Accounting is the act of logging /summarizing what actions in the system were performed and by who, for purposes of reporting utilization.

Attribute
An **attribute** is a descriptor of a person who has authenticated successfully. The username is a commonly used attribute. UNIX group is another example of an attribute. A database role is also an example.

Attribute Authority
An Attribute Authority (**AA** is the name of a component in the Shibboleth architecture. An AA exists as part of an Identity Provider's domain. The AA will release attributes associated with each user; the release policy (what is released to who) is agreed upon by the federation.

Authentication
Authentication should be viewed exclusively as the act of identifying the current user of the system, ie. the owner of the request for action.

Authorization
Authorization is about who can do what. A "big picture" view of authorization is this: **Attributes + Rules --> Allow/Deny Decision**.

Federation
Federation is a trust relationship among a set of root authorities. The federation does not require a common root authority. Each domain / institution maintains its own authoritative list representing its own community. Federation members trust each other to identify their own communities properly, so they do not need direct access to each others' user lists.

Identity
Identity is a "name" that is unique within a system. It is useful for a person to have a single identity within a single domain- for example, a university employee enrolled in a course is one person with attributes 'employee' and 'student', not two people. Federations may choose to represent member identifies to other domains in a manner that is useful as a global identity. For example: susieq@uab.edu tells you that a person with identity susieq is a member of the UAB community, and is not the same as susieq@dartmouth.edu. Even though these identities 'look like' email addresses they should not be treated as such.

LDAP
LDAP stands for Lightweight Directory Access Protocol. It is "lightweight" compared to X.500. For details, see RFC 3377

Mailing List Software
Mailing List Software is intended to refer to an individual packaged message distribution tool. These are the types of systems we see most often today. It is the message distribution and configuration control software that is delivered as a stand alone application. (Compare to Mailing List System)

Mailing List System

Mailing List System is intended to refer to an integrated message distribution solution. This may be a stand alone software application that includes all these functions, but it can also include a system constructed for multiple stand alone applications. It refers to a collection of these tools leveraging middleware to act as a unified application.

Middleware
Middleware is software technology that delivers data across system boundaries. The complexity of the middleware is usually related to the complexity of the relationships between systems. In a simple system, a single application all programmed in the same language, the middleware may be indistinguishable from the system functions themselves. In this simplified case the middleware is an abstract concept that binds the parts of the system together. It might simply be data structures or objects that are referenced at multiple points within the system in order to control the operation of the system. In a complex system, the middleware becomes tangible software components that are often independent of the application components that make up the system environment. In this case, however, the middleware is still focus on making common data objects available throughout the system environment so that all parts of the system can operate in a uniform and coordinated fashion.

mlist
The word **mlist** is an abbreviation for mailing list. The term can be used in many contexts. It is the name of the project and the name of the general class of software which distributes messages to multiple message stream subscribers and can include features such as web management interfaces and message archive stores.

Policy Decision Point
A Policy Decision Point (**PDP**) refers to the **ALLOW / DENY decision** occurring as the last step in authorization.

Policy Enforcement Point
A Policy Enforcement Point (**PEP**) refers to a point in the application where the **PDP** decision is enforced. In a distributed system, the location of the decision and the enforcement of that decision may occur on different systems.

SAML
(OASIS Security Assertion Markup Language)
A software library named OpenSAML exists. It is a set of open source Java and C++ libraries that are fully consistent with the SAML 1.0 and 1.1 CR specifications. For specific information on which features are included, see the README.txt file included in each release.

Shibboleth
When you want to share secured online services or access restricted digital content, the **Shibboleth** system offers a powerful, scalable, and easy-to-use solution. It leverages campus identity and access management infrastructures to authenticate individuals and then sends information about them to the resource site, enabling the resource provider to make an informed authorization decision. More information about the Shibboleth Project.

Single Sign On (SSO)
Single Sign On (**SSO**) refers to a single identity that is shared across multiple systems.

WebISO: (Web Initial Sign On)
WebISO is a system designed to allow users, with standard web browsers, to authenticate to web-based services across many web servers, using a standard, (typically username/password-based) central authentication service. WebISO systems available today include *pubcookie, CAS*, and *WebAuth*