Investigate available open source implementations for authentication and authorization of jabber based instant messaging solutions. Write a short report and presentation of your findings.

Giorgio Badaini

Jabber overview [1]

Jabber is an open system primarily built to provide instant messaging service and presence information. Jabber is based on open standards. Like e-mail, it is an open system where anyone who has a domain name and a suitable internet connection can run their own Jabber server and talk to users on other servers.
The standard server implementations and many clients are also free/open source software.

Features [2]

Decentralization
The architecture of the Jabber network is similar to email; anyone can run their own Jabber server and there is no central master server.

Open standards
The Internet Engineering Task Force has formalized Jabber's core XML streaming protocols as an approved instant messaging and presence technology under the name of XMPP, and the XMPP specifications have been published as RFC 3920 and RFC 3921. No royalties are required to implement support of these specifications and their development is not tied to a single vendor.

Security
Jabber servers may be isolated from the public Jabber network (e.g., on a company intranet), and robust security (via SASL and TLS) has been built into the core XMPP specifications. To encourage use of channel encryption, the XMPP Standards Foundation also runs an intermediate certification authority at xmpp.net offering free digital certificates to Jabber/XMPP server administrators under the auspices of the StartCom Certification Authority (which is the root CA for the intermediate CA).

Flexibility
Custom functionality can be built on top of Jabber's core protocols; to maintain interoperability, common extensions are managed by the XMPP Software Foundation. Jabber applications beyond IM include network management, content syndication, collaboration tools, file sharing, gaming, and remote systems monitoring.

Decentralisation and addressing

The Jabber network is server-based (i.e. clients do not talk directly to one another) but decentralized; by design there is no central authoritative server, as there is with services such as AOL Instant Messenger or MSN Messenger. Anyone may run their

own Jabber server on their own domain.
Every user on the network has a unique Jabber ID (usually abbreviated as JID). To avoid the need for a central server with a list of IDs, the JID is structured like an e-mail address with a username and a DNS address for the server where that user resides separated by an at sign (@), such as username@domain.com.
Since a user may wish to log in from multiple locations, the server allows the client to specify a further string known as a resource, which identifies which of the user's clients it is (for example home, work and mobile). This may then be included in the JID by adding a forward slash followed by the name of the resource. For example the full JID of a user's mobile account would be username@domain.com/mobile. Messages that are simply sent to username@domain.com will go to all the user's clients, but those sent to username@domain.com/mobile will only go to the mobile client.
JIDs without a username part are also valid and may be used (with or without a resource part) for system messages and control of special features on the server.

The Jabber Typical architecture consists of 3 components: [1]

- Jabber Client
- Jabber Server
- Transports

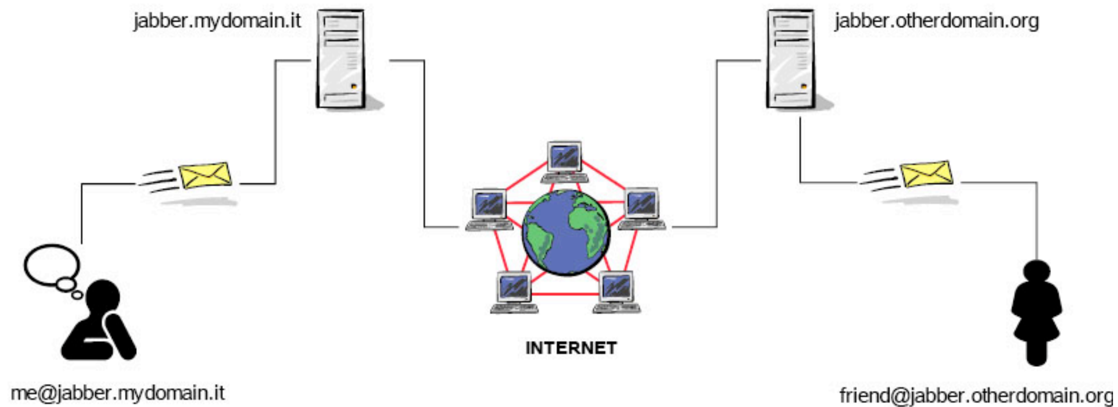
Jabber Client
A client Jabber can be every program that is able to communicate with the TCP/IP protocol. Clients communicate directly with servers using XML format messages.

Jabber Server
Jabber servers permit the users registering and authentication, keep offline messages and deliver them when the user come back online. And Jabber servers use DNS to find the others servers in the Web or use "transports" to comunicate with the others IM servers.

Jabber Transports
Jabber transports permit the comunication between jabber servers and other IM systems.

The Jabber architecture is similar to e-mail architecture

- User Me@jabber.mydomain.it sends a IM to friend@jabber.otherdomain.org
- Server jabber.mydomain.it handles the message
- jabber.mydomain.it opens a connession to server server jabber.otherdomain.org and deliver to it the message
- jabber.otherdomain.org delivers the message to the user friend@jabber.otherdomain.org



The protocol XMPP [3]

The Jabber protocol is XMPP (Extensible Messaging and Presence Protocol). It is an open, XML-based protocol for near-real-time, extensible instant messaging and presence information.
As specified in RFC 3920, the core "transport" layer for XMPP is an XML streaming protocol that makes it possible to exchange fragments of XML between any two network endpoints. Authentication and channel encryption happen at the XML streaming layer using the IETF-standard protocols for Simple Authentication and Security Layer (RFC 2222) and Transport Layer Security (RFC 2246). The normal architecture of XMPP is a pure client-server model, wherein clients connect to servers and (optionally) servers connect to each other for interdomain communications. XMPP addresses are fully internationalized, and are of the form <node@domain> for clients (similar to email). A wide variety of applications can be built on top of the core XML streaming layer. The first such application is instant messaging (IM) and presence. The basic IM and presence extensions specified in RFC 3921 address the requirements of RFC 2779, as well as the contact list functionality expected IM and presence systems. RFC 3921 also makes it possible to separate the messaging and presence functionality if desired (although most deployments offer both).

An example of how Jabber work in authentication and authorization: GoogleTalk [4]

Details of a Session Connection
A peer to peer session is actually made up of two streams of information:
The high-level session management connection, which negotiates the who, what, and how of making a connection. This is XMPP information, and is sometimes called the signaling channel.
The data channel, which is the actual bytes of file or voice data being exchanged.
The diagram above shows the pathways for each of these two types of data.
Here is a general overview of how the different pieces of a libjingle application work together to handle a peer to peer session.
Startup
In order to start running a libjingle application, you must first instantiate and set up a number of required objects. This includes a few objects from each component: the XMPP task manager, SessionManagerTask to listen for incoming messages, NetworkManager, PortAllocator, SessionManager, a SessionClient subclass and

perhaps another Session Logic and Management Component, plus any required Thread objects.

Signing in

Once your application is running, you must sign in to the XMPP server in order to connect to other computers. Send the user's XMPP server name and password to XmppClient::Connect directly, or else use the XmppPump::DoLogin helper task (recommended). XmppClient sends status signals indicating the status of the sign in request. See Signing In to a Server for more details.

If sign in has succeeded, the XmppClient sends the SignalStateChange (STATE_OPEN) signal. The application then sends user status information to the server and requests status information for the user's roster members. PresenceOutTask creates and sends the user status out to the server. PresencePushTask listens for roster member presence notifications from the server, and whenever it receives a presence stanza, it sends a SignalStatusUpdate signal with information, and the application can alert the user.

The following sections describe a connection attempt between two users, Romeo and Juliet.

Connecting

After the application receives roster member status, it can display the list of available members to the user, who can then decide to send a connection request as described here.

In order to connect the data stream, the two computers must negotiate the data connection details over XMPP. This negotiation involves the exchange of two important pieces of information:

Session request specifics [from the initiator to the target]. This describes who you are (by JID), what you want (to connect), and what you want to exchange (files, voice data). It includes a description element specific to the session type: for a voice chat, it would include a codec list; for a file exchange, it would include a file name, data direction, and other information. Both applications must be configured to understand a custom session description stanza. In reply, the target sends an acceptance or rejection stanza. An acceptance stanza might include a description element of its own.

A transport list [from each computer to the other computer]. This is a list of connection information such as local connection candidates (generated by the Port objects under the direction of the P2PTransport object), as well as other connection information such as priority and password information. For a list of what types of what types of candidates are generated, see Candidates and Transports .

In this latest version of libjingle, both of these are in the initial XMPP connection request stanza; in the previous version, the transport list was a candidate list, and it was contained in a separate stanza. The person receiving the connection request must reply with a selected transport in order for the connection to begin.

Romeo sends a connection request to Juliet

Romeo's application instantiates a new Session object. It does this by creating the top level Session Logic and Management wrappers, and having one of them call SessionManager::CreateSession, which creates the new Session object and returns it to SessionClient::OnSessionCreate. After creating Session, the application connects to all its signals to be alerted when the session changes status (accepted or rejected), as well as to receive notifications of incoming XMPP messages.

Next, the application creates a description of the session and wraps it in an appropriate SessionDescription subclass. Voice call descriptions include codecs; file share descriptions include file names.

Next, the Session Logic and Management wrapper calls Session::Initiate on the new session, passing in the description. This causes it to take the following steps:
Session creates a set of potential transports it can use. Each P2PTransport creates a Port object, each of which generates a candidate, and sends the compiled list back to Session. This list includes a default transport as its highest priority transport.
Session sorts the transport offer list with highest preference first, and generates an XMPP offer stanza with the description and transport list and sends it to Juliet.
Session tries immediately to connect with its default channel.
The Session object sends signals whenever it changes state as a result of incoming or outgoing stanzas (sent session invitation, received session invitation, sent accept, running, and so on). Session also sends a signal (SignalInfoMessage) whenever it receives an informational message. These messages are customized to the session type--so, for example, the file sharing application sends a "no more files" informational message when all files have been requested so that the receiver can shut down.
Juliet receives the connection request
The stanza is caught by SessionManagerTask, which forwards it to SessionManager, which recognizes a connection request. SessionManager creates a new Session object and copies the description and candidate list to the new Session object. When SessionManager creates the new session, it sends a SignalSessionCreate notification, which includes the direction of the request (here: incoming to indicate that it is an incoming request). Juliet's SessionClient subscribed to that notification when it was created, and it pops up a dialog box telling Juliet that her roster buddy Romeo is requesting a connection.
Without waiting for Juliet to accept, Session tells P2PTransport to start generating its own local list of transports. When these are ready, Session searches for the first matching transport from Romeo's list that is also in its own list.
If, after being notified about the incoming request, Juliet decides to accept Romeo's request, she calls Session::Accept on the Session object that her application created for the new connection request. Session then creates its own connection description, which includes her chosen transport as well as its own session-specific description object (if necessary) and sends this back to Romeo. It immediately tries connecting over the accepted transport.
If Juliet decides not to accept, she calls Session::Reject, which sends a rejection message and destroys the associated Session object. If Juliet cannot find a transport on her list that she can accept, she will send a rejection message, and SessionManager will destroy the session.
(The order of stanza exchanges differs for old clients that do not support the <transport> tag but libjingle is backward compatible with these clients.)
Romeo receives Juliet's acceptance stanza
The received stanza is sent down to the Session object, which calls OnTransportAcceptMessage to verify that one of the transports it sent out was included in the reply. It then sets the returned transport as the active P2PTransport for the Session, and tells the transport to create and connect channels with the candidate accepted. As soon as the socket can be written to, Romeo can begin sending data.
Session changes its state to STATE_RECEIVEDACCEPT, which will later be changed to STATE_INPROGRESS. Session sends a signal with the new state, so the application can start sending data when appropriate.
See Transports, Channels, and Connections for more information about connections.
The Peer to Peer session begins

The data session continues until one of the issuers sends a redirect or terminate request, or the data channel is broken.
Cleaning Up
Exact details about cleanup vary depending on the client type, but the cleanup of the common objects (Session and P2PTransport) are the same. When a client calls Session::Reject (for an incoming request) or Session::Terminate (for an established connection), the client will send a Terminate message to the other client, and tell SessionManager to delete the session, and call the derived version of SessionClient::OnSessionDestroy. This derived function must free up any allocated resources for that session, and also delete the associated P2PTransport object (by calling Session::DestroyChannel on the session sent to OnSessionDestroy).
To sign out of a session with the XMPP server, call XmppClient::Disconnect.
When a client receives a Terminate message from the other party, it will send the message to the proper Session, which handles it the same way as when a client calls Session::Terminate itself.
When a connection is broken during an exchange of data, P2PTransport tries to reconnect over all the connections for a specific period of time. If the connection cannot be reestablished within a timeout period, it terminates the session as just described. The timeout period is specified by the timeout_ member of SessionManager.
When a connection is broken during an exchange of XMPP information, the sending client will receive an error message, which will cause the session to terminate as described previously.
If one user signs out without explicitly terminating the session, no Terminate message is sent. If the user that signs out actually closes the network connection, the session will terminate because of connectivity as described earlier whether or not data is being sent at the time (libjingle sends periodic pings over all the ports to determine connectivity, and if they all fail and it cannot reestablish a connection, it will terminate). However, if the network connection is not broken, these pings will succeed, and the session will not end until one or the other client either closes or the network connection is broken. In order to avoid this scenario, your application should monitor changes to the roster (by using PresencePushTask).

Our Problem

Our problem is create a distribuited system for Norwegian Health System that will warrant privacy, data security but that permit to every health component access to the patient information from every health structure.
One solution could be integrate the jabber system with the XACML OASIS standard.

What is XACML [5]

XACML is an OASIS standard that describes both a policy language and an access control decision request/response language (both encoded in XML). The policy language is used to describe general access control requirements, and has standard extension points for defining new functions, data types, combining logic, etc. The request/response language lets you form a query to ask whether or not a given action should be allowed, and interpret the result. The response always includes an answer about whether the request should be allowed using one of four values: Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision

cannot be made) or Not Applicable (the request can't be answered by this service).
The typical setup is that someone wants to take some action on a resource. They will make a request to whatever actually protects that resource (like a filesystem or a web server), which is called a Policy Enforcement Point (PEP). The PEP will form a request based on the requester's attributes, the resource in question, the action, and other information pertaining to the request. The PEP will then send this request to a Policy Decision Point (PDP), which will look at the request, find some policy that applies to the request, and come up with an answer about whether access should be granted. That answer is returned to the PEP, which can then allow or deny access to the requester. Note that the PEP and PDP might both be contained within a single application, or might be distributed across several servers. In addition to providing request/response and policy languages, XACML also provides the other pieces of this relationship, namely finding a policy that applies to a given request and evaluating the request against that policy to come up with a yes or no answer.

There are many existing proprietary and application-specific languages for doing this kind of thing but XACML has several points in its favor:

- It's standard. By using a standard language, you're using something that has been reviewed by a large community of experts and users, you don't need to roll your own system each time, and you don't need to think about all the tricky issues involved in designing a new language. Plus, as XACML becomes more widely deployed, it will be easier to interoperate with other applications using the same standard language.
- It's generic. This means that rather than trying to provide access control for a particular environment or a specific kind of resource, it can be used in any environment. One policy can be written which can then be used by many different kinds of applications, and when one common language is used, policy management becomes much easier.
- It's distributed. This means that a policy can be written which in turn refers to other policies kept in arbitrary locations. The result is that rather than having to manage a single monolithic policy, different people or groups can manage separate sub-policies as appropriate, and XACML knows how to correctly combine the results from these different policies into one decision.
- It's powerful. While there are many ways the base language can be extended, many environments will not need to do so. The standard language already supports a wide variety of data types, functions, and rules about combining the results of different policies. In addition to this, there are already standards groups working on extensions and profiles that will hook XACML into other standards like SAML and LDAP, which will increase the number of ways that XACML can be used.

This standard is quite new and I found only two systems that integrate Jabber and XACML. Both this systems are Australian prototypes.

1 Vannotea [6]
The prototype called Vannotea has been developed which enables the collaborative indexing, annotation and discussion of audiovisual content over high bandwidth networks. It enables geographically distributed groups connected across broadband networks (GrangeNet) to perform real time collaborative sharing indexing, discussion

and annotation of high quality digital film/video and images (and shortly 3D objects). Vannotea's flexible design and metadata architecture allows it to be used within many other domains outside the original target domain:

Australian Institute of Sports

Vannotea has enormous potential as a tool to enable geographically distributed coaches to collaboratively analyse and improve athlete's performances. Discussions are underway with the Australian Institute of Sport (AIS) about the software's deployment and application within their Performance Analysis Unit.

The Performance Analysis Unit, which aims to help coaches make effective use of IT in order to enhance athletes' development, is exploring the use of DSTC's technology to assist coaches in disparate locations.

DSTC has been using archival video from the AIS to test Vannotea's usability and applicability for collaborative performance analysis of elite Olympic swimmers. Vannotea has a user friendly search, browse and retrieval interfaces to the video database of swimmers performances during training and competition. For example, the database can be searched on swimmer's name, coach's name, event, lap number etc. It enables coaches to automatically segment the swimming videos into laps or selected clips and analyse and annotate the content using swimming terms and concepts, drawn from a predefined ontology.

CancerGrid

This project is a component of a Department of Education Science and Technology (DEST) Innovation Access Programme research grant (CG050091) - "Integrating Australia into Global eScience". Vannotea will be deployed at UK eScience Centres in Cambridge, Oxford and Southampton. In particular, it will be trialed within the CancerGrid project where it will support collaboration between distributed clinicians coordinating clinical trials for cancer diagnosis and treatment. In particular it will provide distributed clinical teams with shared group access to data associated with the clinical trials. [http://www.escience.cam.ac.uk/projects/cancergrid/]

Museums - Indigenous Knowledge Management

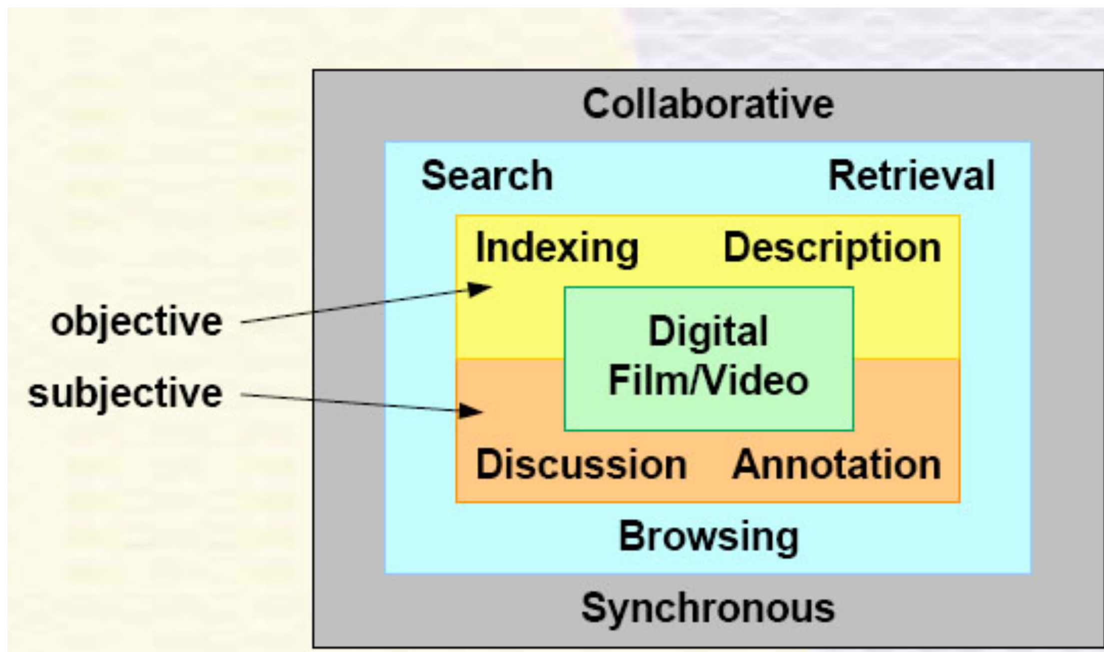
In conjunction with DSTC's Indigenous Knowledge Management project, Vannotea is being extended to enable museum staff to share and exchange knowledge and digital representations of artifacts with the Indigenous communities who are the traditional owners. The aim is to deploy the software within museums to enable distributed groups to collaboratively discuss, describe and contextualize museum content from a variety of different perspectives.

In particular, Vannotea has been extended to enable users within videoconferencing environments to collaboratively attach descriptive, rights and tribal care metadata and annotations to digital images, video or 3D objects.

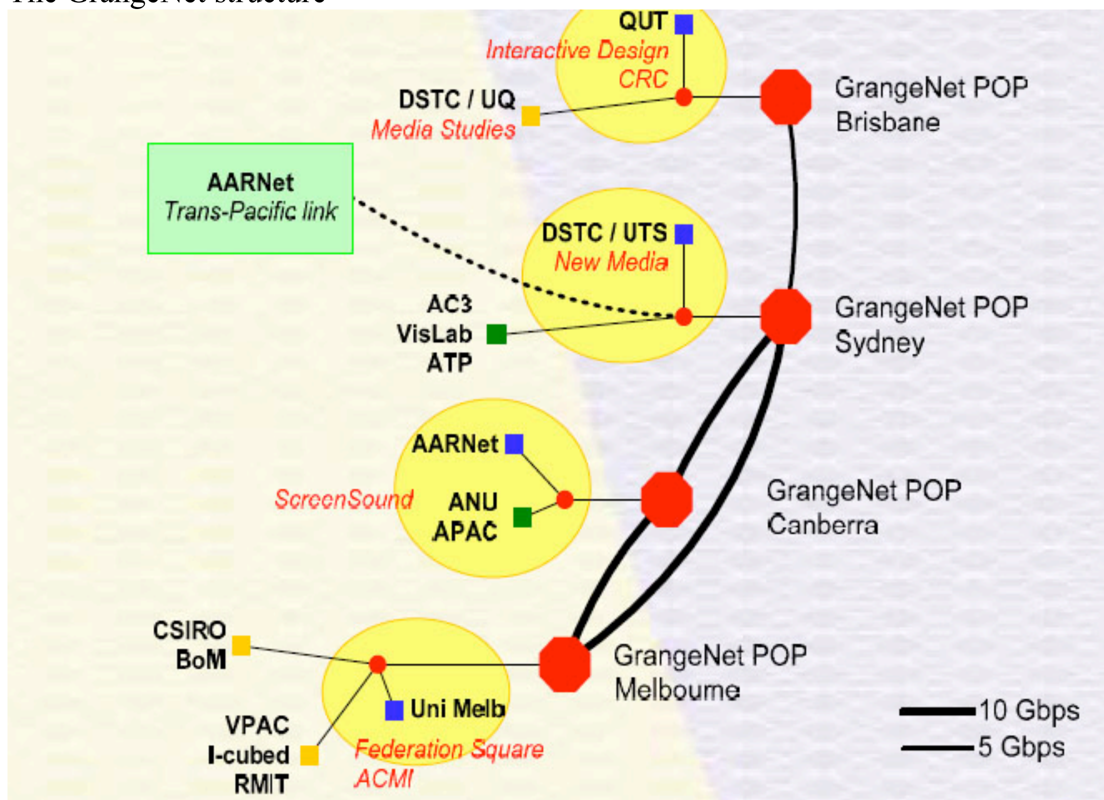
This sharing and exchange of knowledge will hopefully revitalize cultures eroded through colonization and globalization and repair and strengthen relationships

between museums and indigenous communities. [http://metadata.net/ICM/]

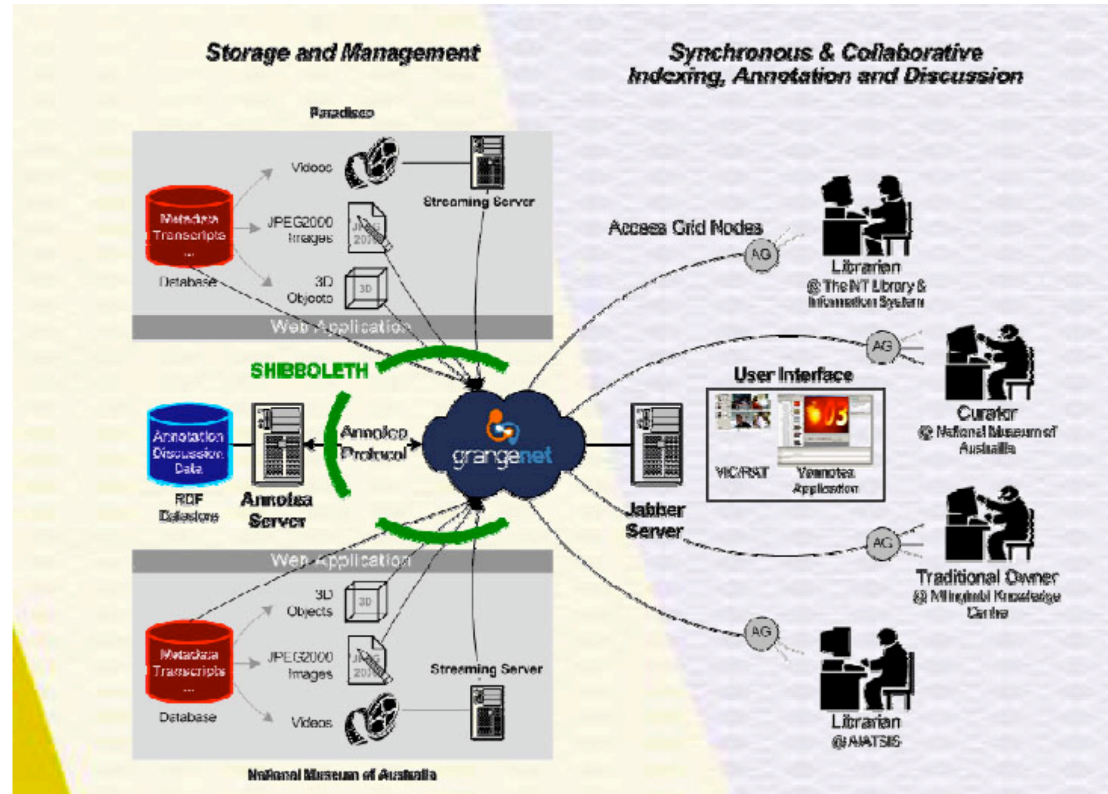
Vannotea scope:



The GrangeNet structure



Vannotea Architecture



2 Identity & Access Mgmt in Australia: MAMS Project [7]

MAMS is building a prototype federated Identity and Access Management (IAM) infrastructure for Australia's Higher Education (HE) sector. This infrastructure consists of Identity Providers (IdP) and Service Providers (SP) which trust each other, and the Federation to manage the trust between all parties. As a result, when a user wants to access a protected service at an external SP, instead of creating new guest accounts for external users, it allows an SP to leverage the user's account with his or her home institution to access it. In other words, the SP will receive all necessary user attributes from his or her IdP, which it trusts, and those attributes will determine the privileges a user gets at the SP. The project allows for the integration of multiple solutions to managing authentication, authorisation and identities, together with common services for digital rights, search services and metadata management. The project provides a 'middleware' component to increase the efficiency and effectiveness of Australia's Higher Eduction research infrastructure.

REFERENCES:

[1] http://en.wikipedia.org/wiki/Jabber
[2] www.jabber.org
[3] http://www.xmpp.org/
[4] http://code.google.com/apis/talk/libjingle/libjingle_applications.html
[5] http://sunxacml.sourceforge.net/guide.html#xacml and http://www.oasis-open.org/home/index.php
[6] University of Queensland: http://www.itee.uq.edu.au/~eresearch/projects/vannotea/index.html; http://www.metadata.net/filmed;
[7] http://mams.melcoe.mq.edu.au/zope/mams; http://www.melcoe.mq.edu.au/projects/MAMS/