Ganymede: An Extensible and Customizable Directory Management Framework

Jonathan Abbey and Michael Mulvaney – The University of Texas at Austin

ABSTRACT

In the fall of 1994, Applied Research Laboratories, The University of Texas at Austin (ARL:UT) presented a paper [1] at LISA VIII, describing work that we had performed designing and implementing a management framework for NIS and DNS, called GASH. In the years since that paper was presented, it has become clear that the design of GASH was insufficient to meet the complex, idiosyncratic, and rapidly changing needs of modern networking. GASH suffered from being too inflexible to be rapidly retooled for a changing network environment, from being limited to a single user at a time, and from being unable to provide management services to custom clients.

In the face of these issues, the Computer Science Division at ARL:UT went back to the drawing board and developed a Java-based directory management framework on the basis of the design principles presented in our GASH paper. Written in Java, Ganymede [1] is based on a distributed object design using the Java Remote Method Invocation [2] protocol and features a multi-threaded, multi-user server, and a graphical, explorer-style client. By supporting customization through a graphical schema editor, plug-in Java classes, and external build scripts, Ganymede is able to support a variety of directory services, including NIS, DNS, LDAP, and even NT user and group management.

Introduction

In early 1992, our laboratory had a problem. We had a need to pull computers from all over the lab together into a common NIS [3] and DNS [4] regime, but the lab was separated into several roughly autonomous groups. We needed a way to create a centralized NIS and DNS domain while preserving the ability of the groups to control their own user and group accounts. In addition, we needed to centralize email delivery and administer automounter volume definitions to support a useful and transparent network architecture. In response to all this, the Computer Science Division at ARL:UT developed the Group Administration Shell (GASH, for short), a text-based shell that allows designated users in our laboratory to issue commands that modify our NIS and DNS tables. We put this into operation in late 1993, and for the last four and a half years, we have run our laboratory on GASH, enjoying significant benefits in database consistency and simplicity of administration.

There have been problems with GASH, however. GASH was a very rigid program that directly manipulated NIS source files and a complex system database file that was transformed into DNS by a Perl script. Whenever we had a need to alter or elaborate any aspect of our network computing environment, we found that modifying GASH was extremely difficult and time-consuming. In addition, certain aspects of GASH's operation proved troublesome in practice. The permissions and ownership model used by GASH was very idiosyncratic, and made it difficult to transfer users between groups. More fundamentally, GASH had no clean way to interact with other tools. If a user wanted to change his password, he had to have his GASH administrator change it with GASH, or use yppasswd and take the chance that the yppasswd daemon might conflict with an administrator making changes in GASH. GASH was clearly an inadequate tool to take us into the future. We wanted to be able to tie our network and account management tools into our personnel databases, we wanted to be able to modify our network topology as needed without spending six months reworking the 50,000 lines of C code in GASH each time, we wanted to support LDAP and NT, and we wanted all of our end users to be able to take advantage of our management tools, which wasn't practical with a single-user tool.

By late 1995, we knew something had to be done. Looking around, we were not able to find a suitable and reasonably priced commercial tool that was focused on the issues we had developed GASH to address and that would give us the path to the future we wanted. The network management packages we were aware of at the time were all focused on managing distributed workstations rather than managing centralized directory services.

We had a lot of experience and insights into the problem domain we were dealing with. We knew we wanted a client-server system. We knew we wanted a generic system that could be easily customized, and

[1] Which stands for The "**GA**sh **N**etwork **M**anager, **D**eluxe **E**dition," of course.

we knew we wanted a GUI. So, our task was set. We would work to build a GUI GASH Construction Set.

A GUI GASH Construction Set

In our LISA VIII paper [1], we observed that we saw some potential in reworking GASH around an object database [5] to facilitate automatic consistency maintenance and the provisioning of a GUI. By late 1995, a certain highly caffeinated object oriented programming language was making a lot of noise for its sophistication, ease of use, and portability. After doing an extensive design review, we determined that Java looked as though it would enable us to meet our design goals on all fronts. The Java Virtual Machine provided us with the ability to have our code run on PC's and Mac's as well as on our UNIX workstations and X terminals, and the Java Remote Method Invocation (RMI) protocol allowed us to do a true distributed object design [6] without having to worry about obtaining a CORBA ORB [2] for all the machines in our laboratory.

After spending the first half of 1996 doing design work with pen and paper, we began the work of implementing Ganymede. Two years and 140,000 lines of Java later, in mid-1998, we are currently beta-testing the Ganymede system. We have produced a robust and customizable client/server system capable of doing everything GASH did, with plenty of room to grow. Our beta-testers have run the Ganymede server on Sparc Solaris, AIX, Linux, and FreeBSD. The client has been run successfully on Windows 95 and NT using Sun's Java browser plug-in and from the command line on the UNIX platforms mentioned above. At the lab, we are running Ganymede on a test basis with all the data from our installation of GASH loaded into the server and experiencing full functionality and essentially perfect up time on the server under the 1.2 beta 4 JDK. We expect to have fully transitioned to running the lab on Ganymede by the time of the LISA 1998 conference.

What Is Ganymede?

Ganymede is a system for managing data which is to be fed into a network through some standard distribution mechanism, such as NIS, DNS, Rdist, LDAP, or the NT Domain Controller system. Ganymede is designed with an emphasis on providing tight control over what types of changes can be made to the database it manages, and on allowing multiple users to make changes to that database simultaneously. The Ganymede server is not intended to serve as a high-volume directory server, but rather is designed to work with directory systems designed for high-volume use, such as NIS, DNS, and LDAP, whose servers may not themselves provide useful mechanisms for managing changes.

The Ganymede system is based on a client-server design. The server contains a built-in object database, and supports custom Java plug-ins which provide intelligent management of object types defined in the server. The server supports an admin console which can monitor the server and that includes a GUI schema editor that can alter the definition of the database held in the server as the server runs. Several clients can be talking to the Ganymede server simultaneously, each browsing the database, issuing queries, making changes, and committing transactions without interfering with each other. Whenever a client commits changes to the database, the server can schedule one or more custom builder tasks to write out source files for NIS, DNS, or whatever is being supported, and then run an external script to propagate the exported data into the network environment. See **Figure 1** for a diagram of the overall system.

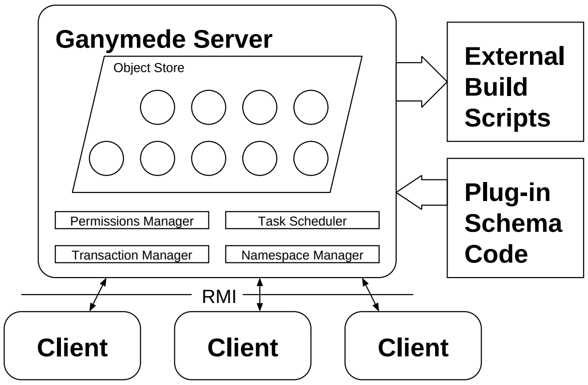


Figure 1: Ganymede Block Diagram.

The Ganymede Server

The Ganymede server, like the rest of the Ganymede system, is written entirely in Java. It contains over 100 classes, which provide for the storage and manipulation of objects, the management of object locking and transactions, the scheduling of database checks and external build processes, and a comprehensive ownership and permissions model, among other features.

Object Store

The Ganymede server has a built-in object database which is held in memory while the Ganymede server runs. The objects in the database are held in a set of thread-synchronized hashing data structures. This design gives the server good performance and multi-threaded safety at the cost of a potentially large RAM footprint. The in-memory database is backed by an on-disk **ganymede.db** file. During

[2]CORBA stands for the Common Object Request Broker Architecture. It is a standard for allowing object oriented code to communicate, object-to-object, across networks. An ORB is an Object Request Broker, a piece of software that handles the network communications on behalf of object oriented code on a given system. The CORBA specification is a product of the Object Management Group (OMG), and more information can be found at their website [7].

execution, the server dumps its database to ganymede.db every two hours. Between dumps, the server maintains a **journal** file, which is a record of transactions made since the last database dump.

The Ganymede database is broken down by object type. Everything held in the database is an instance of a defined object type. Such object types might include things like "User," "Group," "System," and so on. There are a number of object types pre-defined in the database that the server depends on for its operation. These are shown in **Table 1**, below. We will discuss all of these built-in object types in more detail later. For now, notice that each object type in the database is identified by an object type code. The server can handle Object Type ID's from 0 to 32k, but Type ID's of 256 or less are reserved for the server's internal use. Object Type ID's above 256 are available for Ganymede adopters to define for their own use.

Object	Object Type ID
Owner Group	0
Admin Persona	1
Role	2
User	3
System Event	4
Builder Task	5
Object Event	6

Table 1: Mandatory Object Types.

Each type of object in the Ganymede database has its own set of data fields defined. Each field has a name and ID, and can have certain options defined, depending on the type of field. The Ganymede server supports eight different types of fields, as shown in **Table 2**. Most types of fields just hold a single value, but String, IP Address, and Object Reference fields can be defined to be vectors, holding up to 32k values in a single field. String, Integer, and IP Address fields can be connected to a **namespace** defined in the server. The server manages such fields to make sure that the values in them are kept unique across the relevant objects and fields.

Field Type	Options
String	Vector, Length, Chars Allowed
Integer	Max/Min Value
Password	Crypted/Non-Crypted
Date	Max/Min Value
Boolean	Labeled/Non-Labeled
Permission Matrix	
IP address	Vector, IPv4 or IPv6
Object Reference	Vector, Target Type

Table 2: Field Types.

Most of these field types are self-explanatory, but a couple require some discussion. The **permission matrix** field type is used by the Ganymede permissions system and is not really useful in any other context. We'll talk about where the server uses the permission matrix field type when we discuss the Ganymede permissions system. We do need to talk about the **object reference** field type, but before we get into the details of this field type we need to talk about how objects in the server are identified.

Invids and Invid Fields

Objects in the database are identified by an object called an **invid**, which stands for **INV**ariant **ID**. An invid is a Ganymede object identifier, and is implemented as a pair of numbers. The first number is the object's type id, the second is a number between 0 and 2 billion unique to that object within its object type. Object numbers are never re-used. This makes it possible to unambiguously track the history of an object in the server's logs, but it does limit the server to handling 2 billion objects of a particular type over its lifetime.

The object reference field type is simply a field that holds invids. In fact, from now on, we'll refer to this field type as an **invid field**. Invid fields are used throughout Ganymede to link objects together. When one object's invid is placed in an invid field in another object, those objects are said to be linked. All object links in Ganymede are symmetrical, so that each object has references to all objects in the database that point at it, and vice versa. Because all objects in the database are symmetrically linked, the database can easily be kept up-to-date whenever objects are deleted. All that the server has to do in order to clean up after deleting an object is to modify all objects that were listed in the invid fields of the deleted object; it is not necessary to sweep through the entire database looking for linked objects. Another advantage of using invids to link objects is that objects in the database can be renamed or relabeled without disturbing the linkages established in the server.

Invid fields can be configured so that an invid field in one object is linked to an invid field in another object. This is shown in the **schema editor** screen shot in **Figure 2**. In this screen shot, we see the users field in the group object being edited. The users field in the group object is an invid field that points to the groups field in the user object. When the client edits a group object, the server will automatically provide a list of users that can be placed in this field. Adding a user to this group will automatically cause the group to be added to the user, and vice versa. The user can look at either object and see the relationship. This bi-directional linking is very important to the structure of the Ganymede server. It is responsible for a lot of the intelligence of the server. If a user were to try to delete a group, but didn't have permission to edit the users listed in the group, the server would detect this and might reject the operation, depending on how the schema was configured.

Some object reference fields are "edit-in-place," which means that the objects referenced by that field are handled as if they were contained within the referencing object. An object type must be designated in the schema editor as an **embedded** object in order to be linked to an edit-in-place field. An embedded object is for the most part very much like any other object in the database, but it does not have its ownership and permissions tracked independently of its parent, and the server handles its creation and deletion a little bit differently. The easiest way to see the differences between embedded and top-level objects is by looking at the special fields the server uses to keep track of these objects.

Mandatory Fields

Just as there are mandatory object types, so too are there a number of mandatory field definitions. The fields shown in **Table 3** are defined in all non-embedded object types in the server. All field ID's below 100 are reserved for global fields (fields defined in all top-level objects). Field ID's between 100 and 256 are devoted to fields in the mandatory object types that the server depends on for its operations. Field ID's above 256 are user-assignable fields and can be configured in the schema editor.

Embedded objects have a different set of mandatory fields, which are shown in **Table 4**. The **container field** is simply a pointer to the object that the embedded object is contained in. This field is linked to the field in the parent where the embedded object appears.

Field	Type	Field ID
Owner List	Invid Vector	0
Expiration Date	Date	1
Removal Date	Date	2
Notes	String	3
Creation Date	Date	4
Creator Identifier	String	5
Last Modification Date	Date	6
Last Modifier Identifier	String	7
Back Links	Invid Vector	8

Table 3: Mandatory Fields For Top-Level Objects.

Field	Type	Field ID
Container	Invid	0
Back Links	Invid Vector	8

Table 4: Mandatory Fields For Embedded Objects.

Most of the mandatory fields should need no explanation. We'll discuss the **owner list** field when we talk about the Ganymede permissions system a bit later on. For now, let's talk about the **back links** field.

The back links field is a bit special. Previously, we said that the server guarantees that all references made in an invid field are symmetrical, and we gave the example of user and group object types having their groups and users fields symmetrically linked. This kind of visible bi-directional linking sometimes doesn't make sense. In cases where it doesn't, an invid field can be configured so that it points to an object without specifying the target field. The server will use

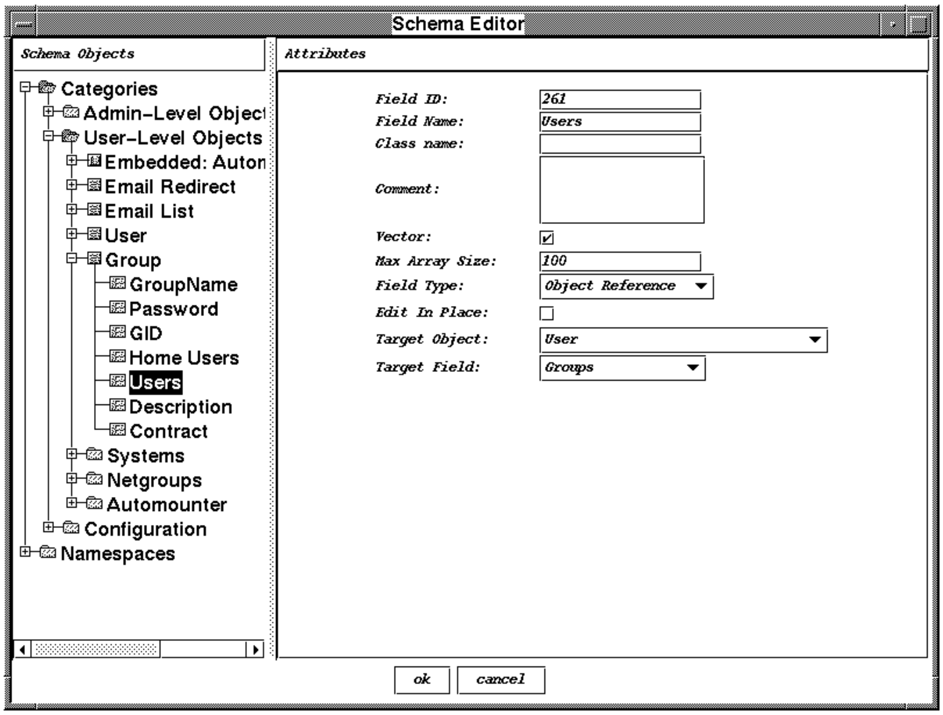


Figure 2: The Schema Editor Editing an Invid Field.

the back links field to handle the back-reference. The client won't see this back-link, but the server will, and will use it when the referenced object is deleted.

The DBEditObject Class and Server Customization

One of the keys to the Ganymede server's flexibility is that it takes advantage of Java's object oriented language features and dynamic linking to allow individual customizers to write classes to manage objects. The **DBEditObject** class in the server is consulted on every major decision having to do with how the server should handle operations on an object. The Ganymede schema editor allows adopters to bind custom DBEditObject subclasses with object types in the server. **Figure 3** shows the **arlut.csd.ganymede.custom.userCustom** class being bound to the user object.

DBEditObject provides over two dozen methods that can be overridden by custom logic to inject intelligence into the Ganymede server. While it is out of the scope of this paper to describe all of the ways in which DBEditObject can be customized, we can mention a few highlights, and we will provide a simple sample of customization through DBEditObject subclassing.

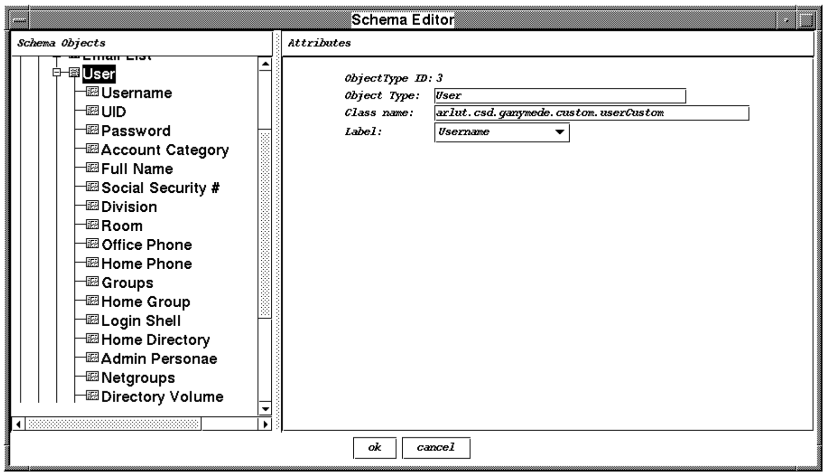


Figure 3: The Schema Editor Binding a Custom Class.

```
package arlut.csd.ganymede.custom;

import arlut.csd.ganymede.*;

public class userCustom extends DBEditObject {

// Boilerplate Constructors Omitted

public boolean fieldRequired(DBObject object,
                             short fieldid)
 {
  switch (fieldid)
    {
      case userSchema.USERNAME:
      case userSchema.UID:
      case userSchema.HOMEDIR:
      case userSchema.LOGINSHELL:
        return true;
      case userSchema.PASSWORD:
        return !object.isInactivated();
    }

  return false;
 }
}
```

Figure 4: A Simple Custom DBEditObject Subclass.

The DBEditObject class provides the ability for custom code to extend or override the default permissions system, to approve or deny any change to fields within an object based on the contents of the object, its relations with other objects, or the identity of the admin seeking to make the changes. It can provide a list of valid choices for **string** and invid fields. It can return custom dialogs in response to attempted operations, or even involve the client in a step-by-step wizard interaction sequence. It can get involved when a transaction is committed, to take actions outside of the database, such as creating home directories when users are created, or connecting changes to an object in the Ganymede database to an external database.

For a simple example of what is involved in a DBEditObject subclass, see **Figure 4**. This example shows the code necessary to specify what fields must be present in a user object when a transaction is committed.

Permissions and Ownership

One of the critical elements of Ganymede's design is the permissions model. Ganymede provides a universal permissions model that allows complete flexibility in apportioning privileges to classes of users/administrators, without becoming so unwieldy as to be impractical. The model is designed to support group administration, with ownership over objects shared by groups of administrators. Different classes of administrators can be defined, each with different privileges over different kinds of objects, and different fields within those objects. The Ganymede server has three object types defined to support this model, the **admin persona**, **owner group**, and **role** objects, as shown in **Figure 5**.

Each user in the Ganymede system can have one or more admin personae associated with it. The admin personae represent administrative privilege sets that the user can select, through an **su**-like mechanism in

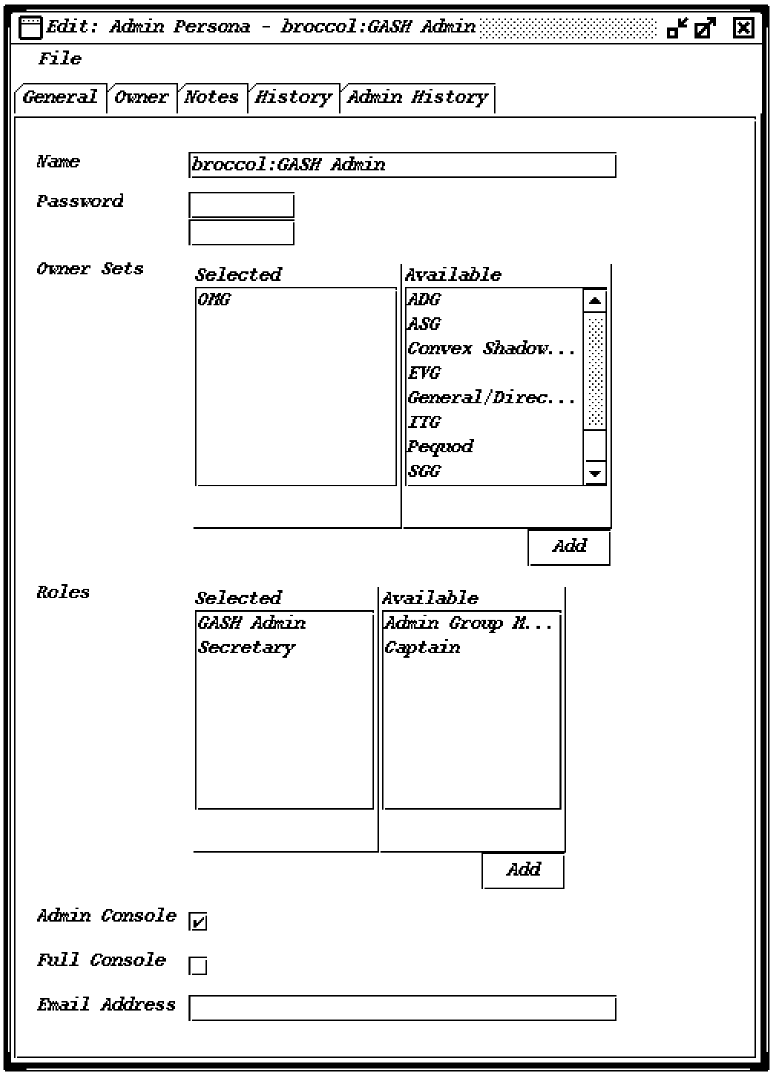


Figure 6: Ganymede Admin Persona.

the client. An admin persona object marries a list of owner group objects and a list of role objects to define the objects and operations that can be performed. The owner group objects define what objects are considered to be under the ownership of that admin persona, while the role objects define what operations the admin persona is permitted to perform.

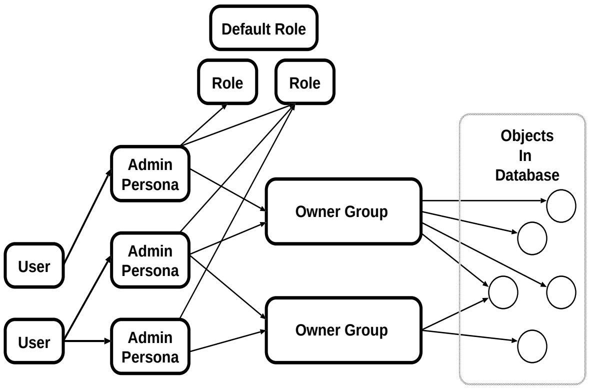


Figure 5: Ganymede Permissions Objects.

Admin Personae

All users registered with Ganymede may have some minimal permissions granted them by the **default role** object. This will typically include the ability to edit their own passwords, shells, and finger information, and to view some information about other users registered in the Ganymede database. If an individual is to have more privileges than that, an admin persona must be created for the user, as shown in **Figure 6**.

An admin persona includes a password, which must be entered by the user in order to access their extended privileges, a list of owner groups and roles, check boxes indicating whether the admin is privileged to run the server-monitoring console, and an optional email address for Ganymede to use to send mail to in response to the admin's actions.

Owner Groups

As mentioned above, all objects in the database are owned by owner groups, rather than by individual administrators. This design decision came out of our experience with GASH. By having all ownership vested in owner groups rather than in individual

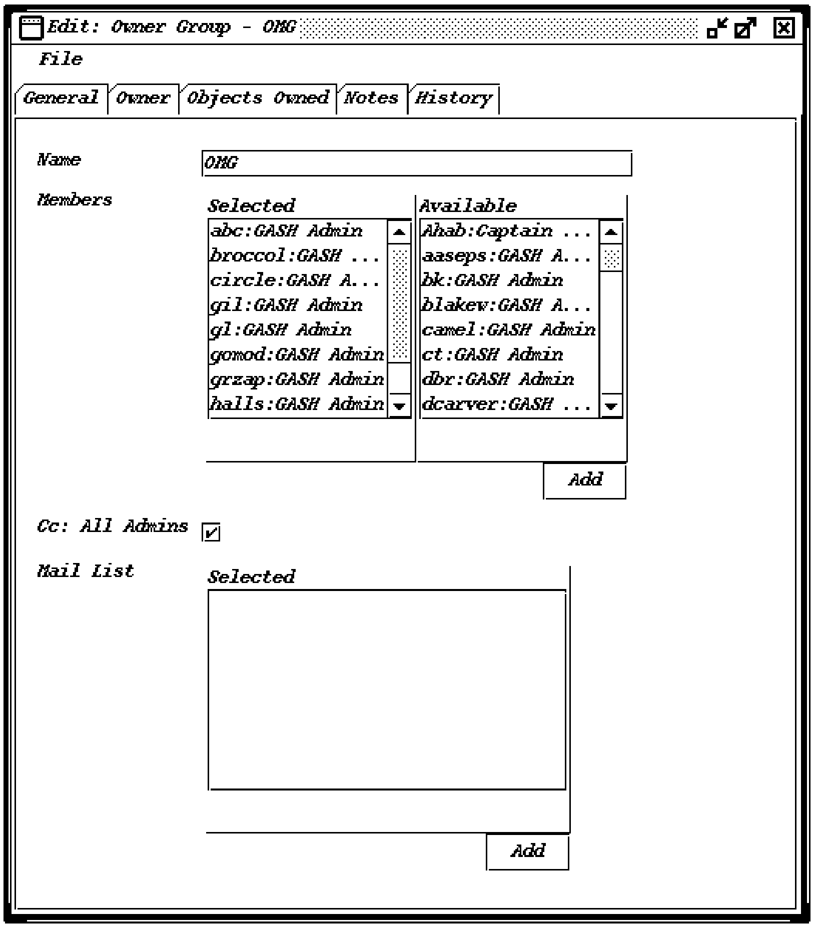


Figure 7: Ganymede Owner Group.

administrators, it is possible to bring a new administrator into a group without having to manually add that administrator to the ownership list of hundreds or thousands of objects. Other fields defined in the owner group object (see **Figure 7**) are designed to let administrators in a group share email notification for actions taken on objects owned by that owner group.

The Ganymede server supports a hierarchy of owner groups. One owner group can own another. Not only do the admins in the first owner group have ownership rights over the second, but also over all objects owned by that owner group. **Figure 8** illustrates this. Admins belonging to the engineering owner group in Figure 8 own not only the hardware and software owner groups, but also all of the objects owned by those groups. In addition, owner groups are considered to own **themselves**, so an admin belonging to the engineering owner group could, if permitted by the roles granted to him, add or delete admins from the engineering group as well as the hardware and software groups.

The **supergash** owner group is special; all objects in the database, including all owner groups, are implicitly owned by the supergash owner group. Any admins belonging to the supergash owner group have ‘‘root’’ privileges over the Ganymede database.

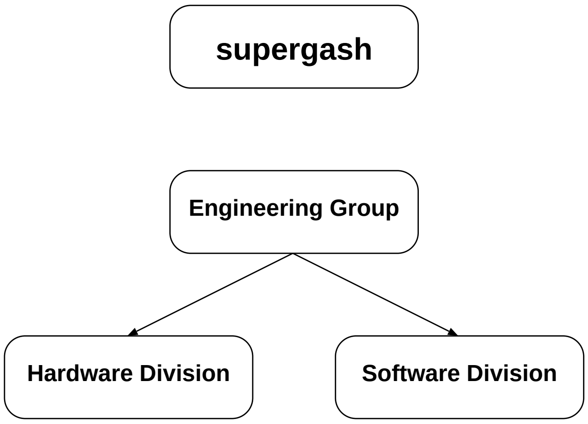


Figure 8: A Hierarchy of Ganymede Owner Groups.

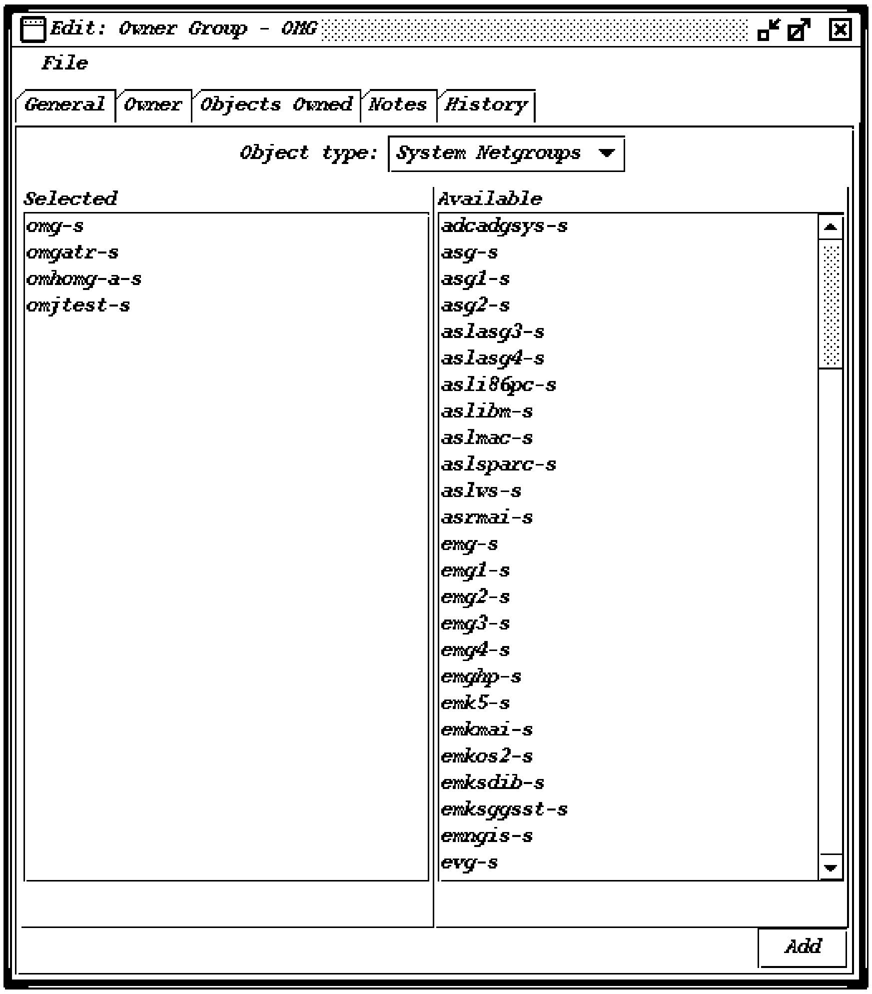


Figure 9: System Netgroup Objects Owned by an Owner Group.

An admin with editing privileges for an owner group can add or remove objects from that owner group by simply editing the owner group. This is shown in **Figure 9**.

Roles

The role object defines what an admin persona can do, both for objects owned by the admin through his owner group membership, and for objects "at large" in the database. Each role object contains two **permissions matrices**: one for objects owned, and one for default permissions. These permission matrices contain an array of booleans which allow access to the database by object type and field, with create, edit, view, and delete permissions categories. An admin can only view, edit, create or delete objects and fields that are specifically granted him by his set of roles. See **Figure 10** for a list of fields defined in the role object.

Of special interest in Figure 10 is the "Delegatable Role?" check box. In order to support a true hierarchy of administrative control, admins can be granted the power to create new roles, and to create new admins. An admin who creates a new role or admin may not grant that role or admin privileges that he himself does not have from a delegatable role. That is, if an admin has the **GASH admin** and **secretary** roles, and only the **secretary** role is delegatable, the admin will only be able to grant the **secretary** role to admins that he creates, and if he creates a new Role, will only be able to set bits in that Role that he got from either the **secretary** or **default** roles. **Figure 11** demonstrates this. The check boxes that are visible correspond to bits that the admin editing the role has himself had granted to him through a delegatable role. The boxes X'ed out represent privileges that this admin may not grant to other roles.

The combination of the owner groups, which determine which objects are accessible, and the roles, which determine what can be done to those objects, provides complete flexibility while maintaining the ability to make wide-ranging changes in the authorization schema by simply editing one or two objects in the Ganymede database.

As implied above, all of the objects in the Ganymede server, including the owner group, role, and admin persona objects, are administered by this permissions system. The same permissions system that controls access to the Ganymede database also controls access to the controls themselves.

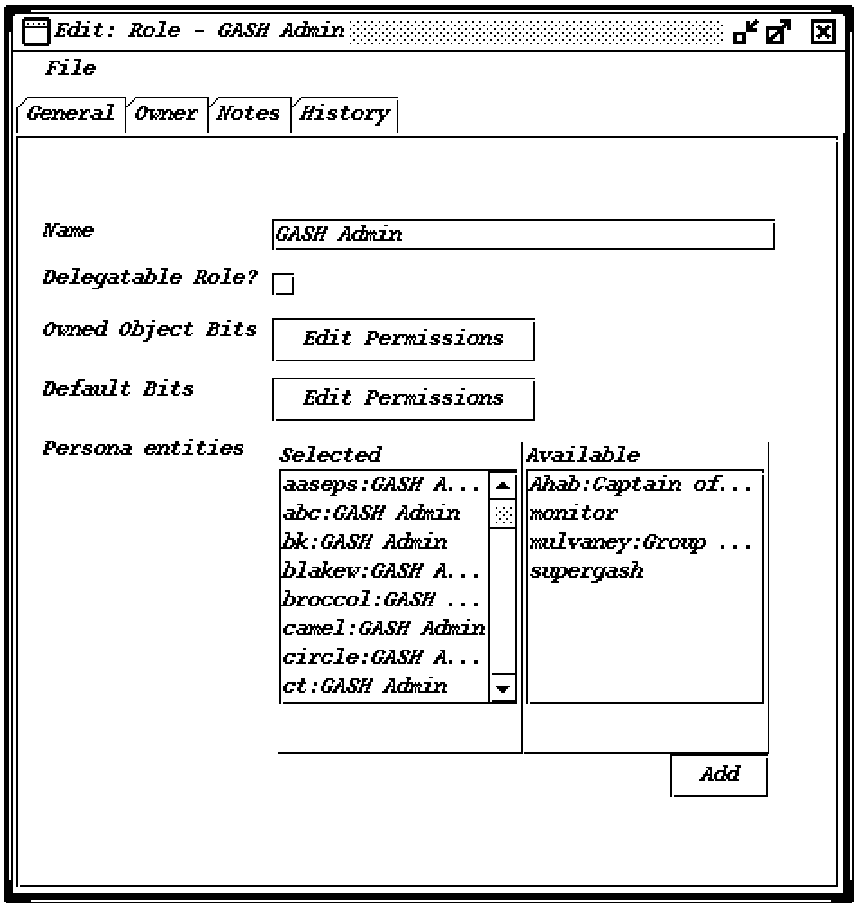


Figure 10: A Ganymede Role.

As powerful as this system is, it is not complete. There will be cases where a more specialized permissions model is required. Take for instance the case of maintaining a public mailing list where users should be able to add and remove themselves, but not touch any other user in the list. All that would be required to support this sort of model would be to bind a custom DBEditObject subclass to the Mail List object type in the Ganymede server and redefine the **anonymousLinkOK()** and **anonymousUnlinkOK()** methods in DBEditObject. Other methods in the DBEditObject class can be overridden to implement custom ownership determination logic, or even to entirely override the normal persona/role/owner group permissions system.

Transactions

The Ganymede server is built around a transactional model wherein clients connected to the server check out objects for editing. There may be many clients connected to the server simultaneously, but changes made to objects in one transaction will not be visible to other users until the transaction is committed. Queries issued by clients are guaranteed to be atomic with respect to transactions across the duration of their processing.

The server supports checkpointing and rollback within the course of a transaction. This allows for complex sequences of operations to be attempted and undone if the sequence could not be carried to completion successfully.

The transaction commit process in the Ganymede server is based on two-phase commit logic, so that custom code can be written to connect transactions issued in Ganymede to transactions in external databases.

In the Ganymede server, whenever transactions are committed, a record of the transaction is written to a journal file. This journal file allows the server to recover any transactions that were committed between the time that the server last performed a full database dump and an abnormal shutdown. The worst case for the Ganymede server on power failure or server crash is the loss of the most recently issued transaction.

When transactions are committed, the Ganymede scheduler schedules external build processes for execution. If multiple transactions are committed while the Ganymede scheduler is still executing the previous external build, the Ganymede scheduler will simply initiate another external build when the first build completes. Thus, multiple transactions made by users may be propagated to the external environment in bulk, depending on the rate that transactions are committed and the time necessary to complete an external build.

Logging and Email Notification

Ganymede, like GASH before it, has a very thorough logging and email notification system. All significant events on the server are logged to disk, and can also be emailed to interested parties. There are a wide variety of **system events** built into the server, and a supergash-level administrator can customize the server's email notification behavior. **Figure 12** shows the notification options for a system event.

While the system event categories are more-or-less hard-coded into the server, it is important that custom object types can have their significant events

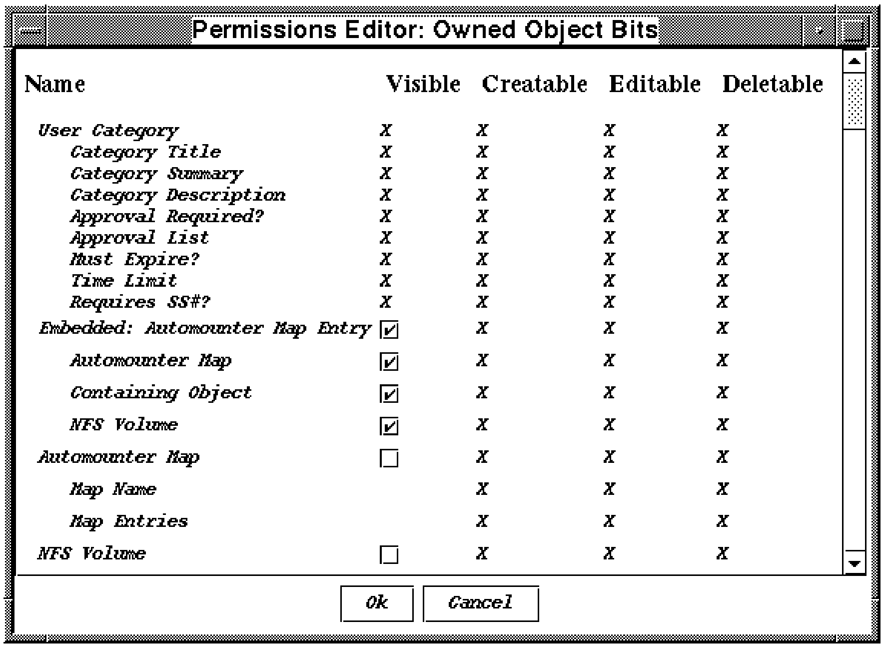


Figure 11: Permission Bits in a Role.

handled specially as needed. To support this, the Ganymede server also supports **object events**, which meld one of a set of common happenings to objects along with the type of object of interest. **Figure 13** tells the story.

One of the lessons we learned from GASH is that it is often important for admins to be able to tell what has happened to objects under their control. In response to this, the Ganymede log file has been designed to be easily parsed. The server itself can scan its own log file to report on changes made to an object, as shown in **Figure 14**.

Background Tasks

The Ganymede server includes a built-in task scheduling facility, similar to **cron**. A background thread queues various tasks for execution, including those shown in **Figure 15**.

The ‘‘garbage collection’’ task runs every night at midnight to do a bit of preemptive house cleaning in the server. The ‘‘database dumper’’ task runs every two hours, consolidating the database and cleaning the journal file. The ‘‘expiration’’ and ‘‘warning tasks’’ each run once a day, to handle objects that have had their expiration or removal times set. The warning task is responsible for looking at objects that will expire or be removed in the near future, and sending out warnings to the administrators responsible for those objects through email. And then, there are the builder tasks.

Builder Tasks

Just as it is possible to define custom subclasses of DBEditObject to provide custom management of object types in the server, it is also possible to define custom builder tasks to be loaded into the server at runtime, as in **Figure 16**. Whenever a transaction is committed, any builder tasks registered with the server are scheduled for execution. The builder tasks will scan the Ganymede database, write out source files for NIS, DNS, etc., and call an external shell script to take those source files and propagate the data into NIS, DNS, or whatever else is being managed.

Schema Kits

The combination of a Ganymede schema definition, a set of custom DBEditObject subclasses, and any custom builder tasks together comprise a Ganymede **schema kit**. As currently available for

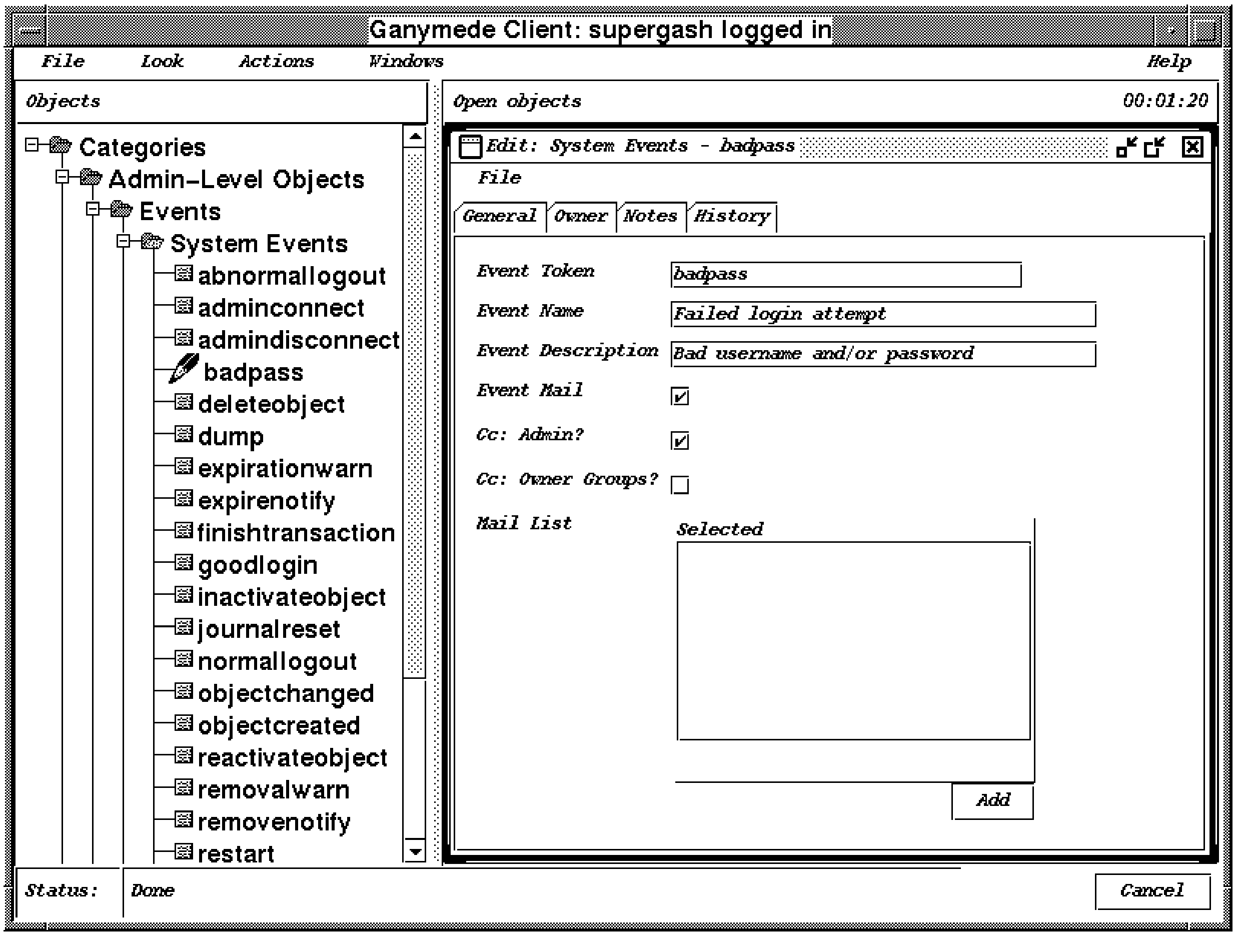


Figure 12: A System Event Record.

download, the Ganymede server includes three schema kits. One is an ''nisonly'' kit that handles Solaris-style passwd and group files, another is a ''bsd'' kit for managing BSD 4.4 master.passwd and group files, and the third is a full-scale ''gash'' kit that was designed as a way for adopters of GASH to have a minimal-work ''drop-in'' replacement for their existing GASH installations. Each of these kits includes a loader program to scan the original files and create a ganymede.db file which Ganymede can then manage.

The nisonly and bsd kits are provided so that UNIX admins can download Ganymede, load their passwd and group files, and start playing with Ganymede without a large commitment of time. The GASH kit is a much richer environment, and implements the complex network management logic described in our LISA VIII paper [1]. The GASH kit, like GASH itself, is designed to manage a single DNS domain and a single NIS domain, with support for NIS-specific features like netgroups and automounter configuration maps.

We are working on developing a next-generation schema based on the GASH model. This schema is to have richer support for DNS, including support for generic subnetwork allocation. We also want to separate out personal identification from the user account. Having separate person objects would facilitate proper generation of a canonical LDAP directory for our laboratory, and would allow us to track responsibility for user accounts more conveniently.

Another design feature of our next-generation schema is support for NT and UNIX integration. We are currently shadowing our UNIX accounts into our NT domain controller using Rsh and some Perl scripts on the NT side. In our next-generation schema we plan to have a check-box on user objects to control whether or not the account should be replicated on NT, and to support NT group accounts as well as UNIX group accounts within Ganymede.

There are quite a number of other interesting possibilities for schema development. One obvious possibility is a DNS-only schema, with support for managing a large number of DNS domains and IP address ranges. We have demonstrated in our work implementing the GASH kit in Ganymede that the DBEditObject customization hooks are adequate to handle multiple-interface systems and network allocation. Creating a DNS-only schema kit would be some work, but we do not believe that the task would be overly difficult, and such a schema kit could be of considerable utility for ISP companies.

The Ganymede Client

The Ganymede server supports a generic Java interface for clients, allowing various custom clients and automated processes to talk to the server. After

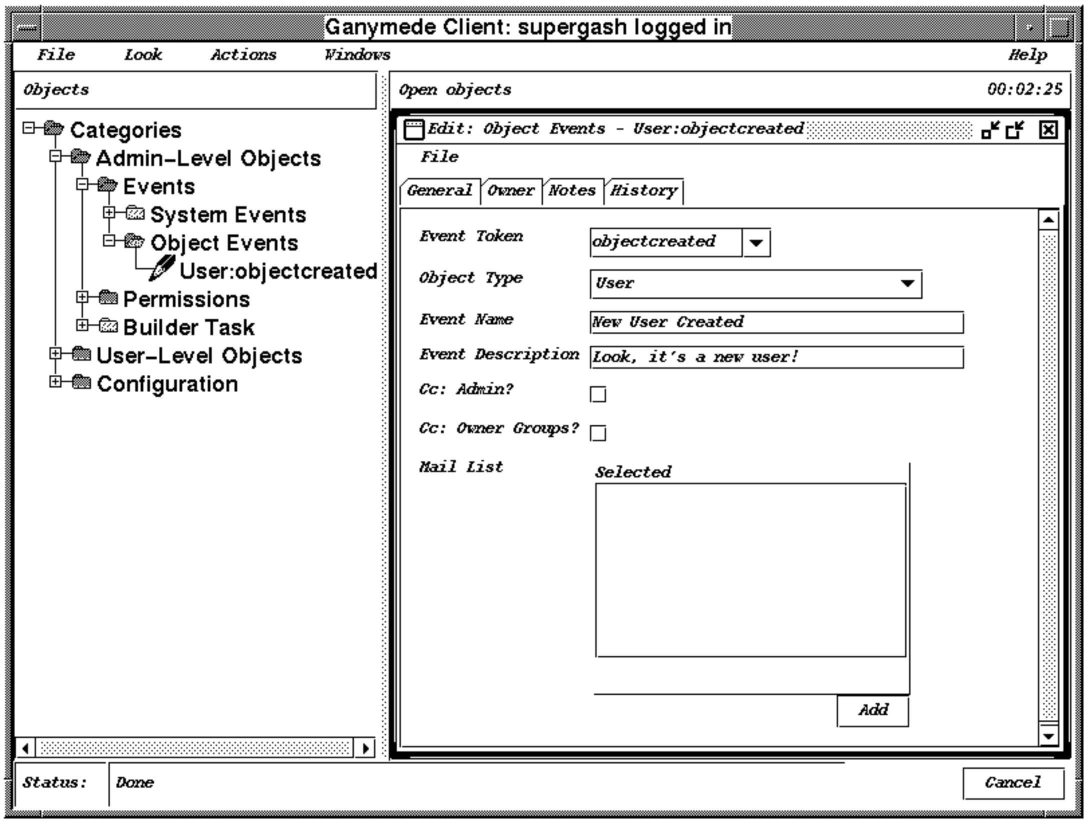


Figure 13: An Object Event Record.

connecting, clients can manipulate objects through RMI references to the objects in the database.

The generic client API permits us to write a variety of custom clients to handle specific tasks. For example, we have developed a command-line-based application that changes a user's password and looks just like Unix's passwd command. After prompting for the user's current and new passwords, this client logs on to the server, checks out the appropriate user object, changes the password, and logs out. Clients could easily be written to do bulk edits to the database, or to tie automated processes of various sorts to the Ganymede server.

Most of our effort in client development has gone into developing a generic GUI client. This client has very little customization built in to it; it queries the server at run-time to get information about the schema being used. With a few exceptions to provide special handling for the mandatory object types that the Ganymede server depends on, it is completely generic, and can be used with any database definition in the server.

This primary Ganymede client displays a large window with two main panels. The left panel displays a tree that lists all the objects in the database, sorted by object type, while the right panel holds windows used for viewing and editing objects. The tree is built when a user first logs in, and can be used to browse the database. Each node of the tree has a context-sensitive menu associated with it, which is accessible by right-clicking on the node.

When the client first connects to the server, it opens a new transaction. The user can commit or cancel changes to the database made during the open transaction by using the "Commit" or "Cancel" button in the lower right-hand corner of the client. No changes are made to the database until the "Commit" button is clicked, so the user has the ability to cancel any actions he performed during the transaction.

Logging In

The client can be run either as an applet in a Web browser or as an application. When the Ganymede client is run in a Web browser, it appears as an applet in the browser's window, as shown in **Figure 17**. After entering a correct username and password, the

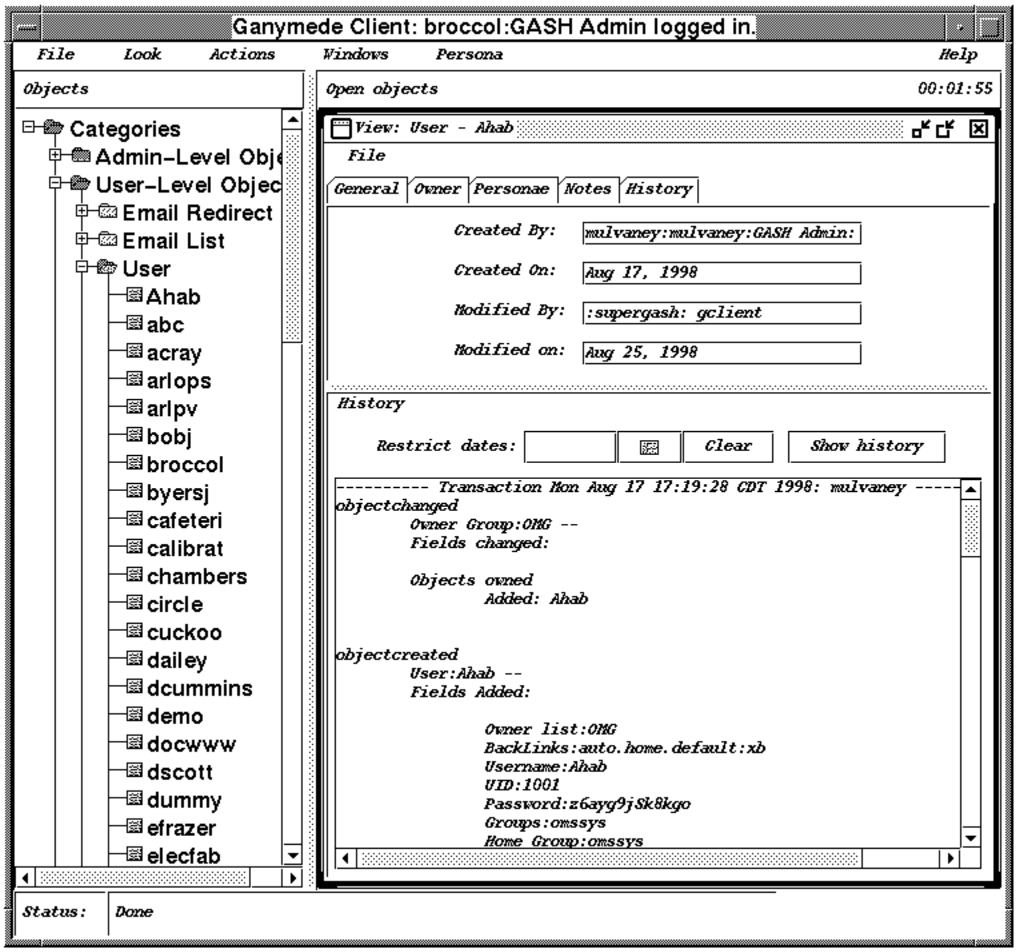


Figure 14: A User Object's Sordid History.

main client window appears. At this time, the client has queried the server to determine what sort of objects have been defined and to retrieve a definition for each type of object that the user will be able to edit. The client caches this information to speed up later operations.

When logging in as an end user, the only visible object in the tree is the user's own user object. In order to gain more privileges, a user can access his admin persona by choosing a persona on the **Persona** menu. The user will be presented with the dialog shown in **Figure 18**, where he will enter the password for his admin persona. Successfully changing to a new admin persona causes the client to rebuild the tree in order to reflect the expanded permissions, as well as to close any windows that might be open.

By default, the tree only shows those objects which are editable by the current admin persona, but the user can choose to have the tree display all the objects he has permission to view by selecting the **Show All Objects** menu item from the menu shown in **Figure 19**. Typically, the default role will grant users permission to view only a small number of objects, such as other users in the same group. Most administrators will be have roles assigned them which grant privilege to browse more of the database.

Editing Objects

After opening an object node, a list of the editable objects of that type is shown. When the "Edit Object" menu item is chosen from the tree's pop-up menu, an editable window is placed in the right-hand side panel. This window allows for the editing of this object.

When an object is first opened for editing or viewing, the server generates and sends the client a remote reference corresponding to that object. The client then talks directly to the object to determine what fields are defined within it, and builds up the fields displayed in the editing window.

Most types of fields in an object are displayed with simple GUI widgets: check boxes for boolean fields, date fields for dates, and text fields for single strings, IP addresses, and numbers. Object reference fields are either pull-down lists for scalar fields, or a composite selector for vector lists. The selector shows a list of available references on one side, and a list of

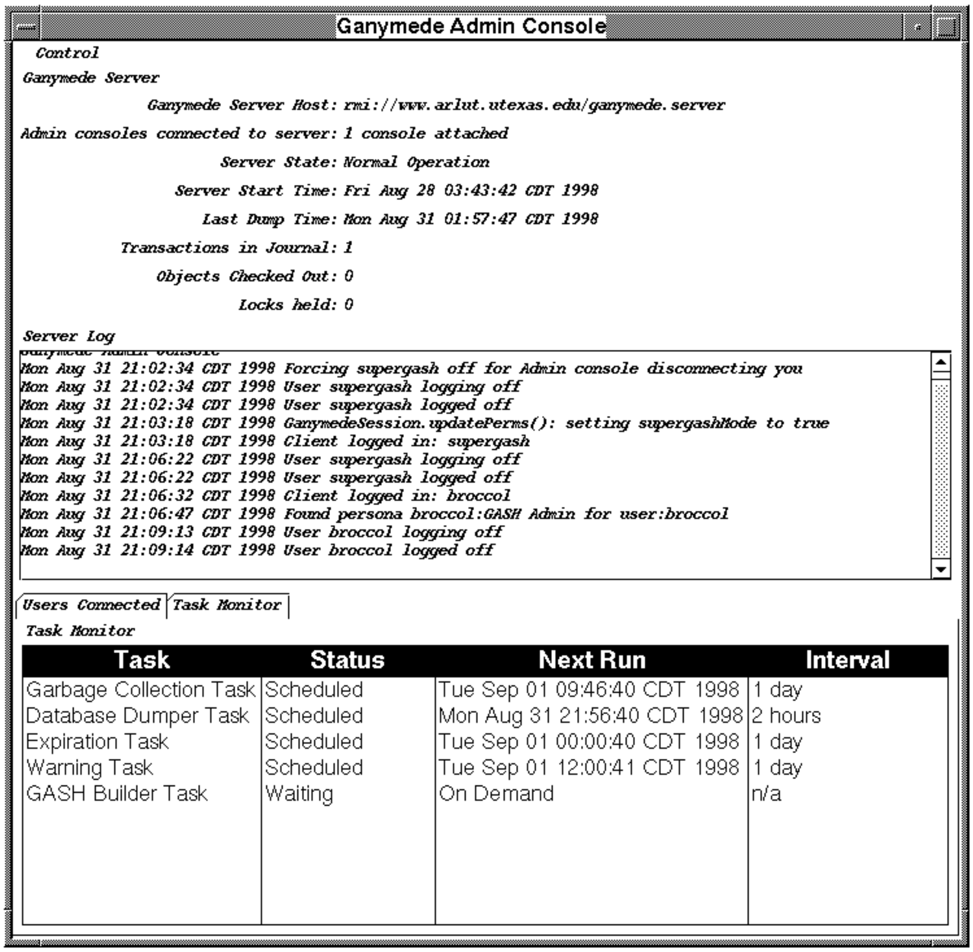


Figure 15: The Admin Console, Showing Registered Tasks.

selected references on the other. By moving the labels between the two lists, references are added or removed from the field.

Embedded objects are handled by embedding an edit panel within the main edit panel, as shown in **Figure 20**. The embedded object's fields can be hidden or revealed by clicking on a icon in the panel's container. Because embedded objects are always employed in a vector context, there are also controls to add or delete objects of the appropriate type from the containing object. Embedded objects may themselves

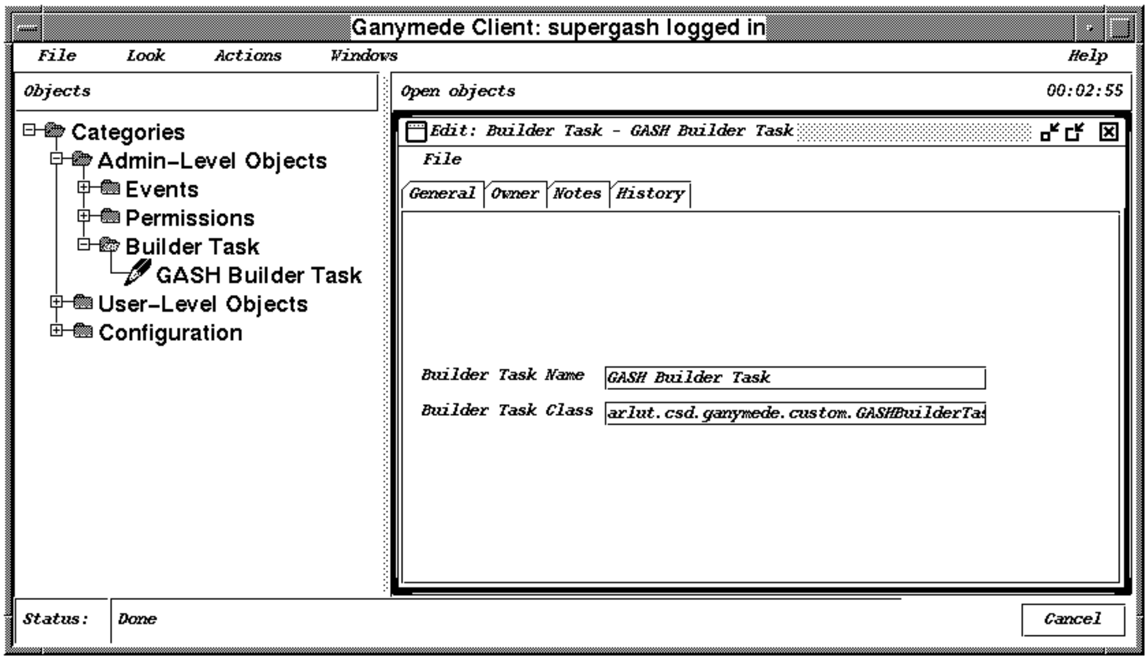


Figure 16: Registering a Builder Task.

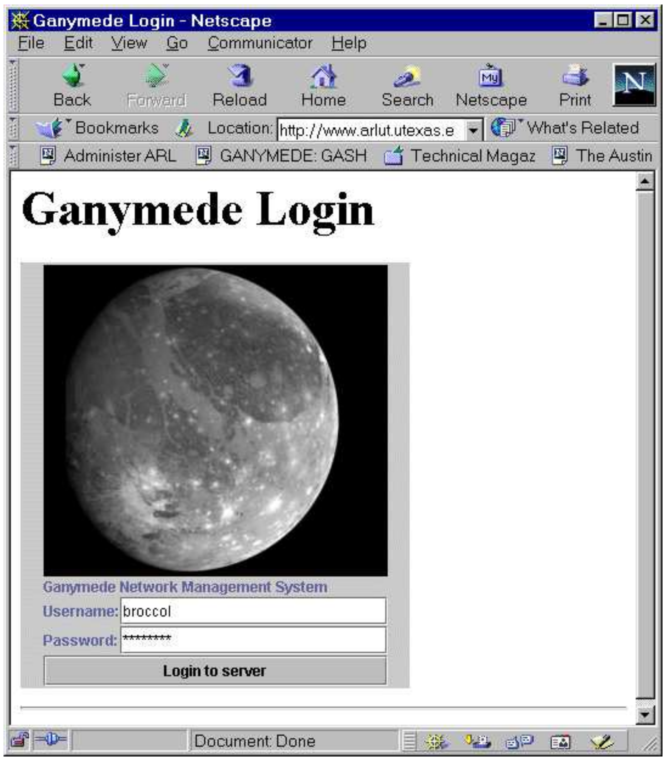


Figure 17: Ganymede Login Screen.

contain more embedded objects, and so a complex hierarchy of containment can be managed, if necessary.

Return Values

When a field is changed, the server reports to the client by sending a special object called a **return value**, which tells the client what happened as a result of the change. In a simple case, the return value contains information about the success or failure of the change. The return value is capable of much more, however; it can instruct the client to display an informational dialog, and it can tell the client which other fields need to be rescanned because of the change to the current field, either in the current object or other open objects. This causes the client to query the server and update the specified fields immediately. Also, the server can use the dialog to initiate a

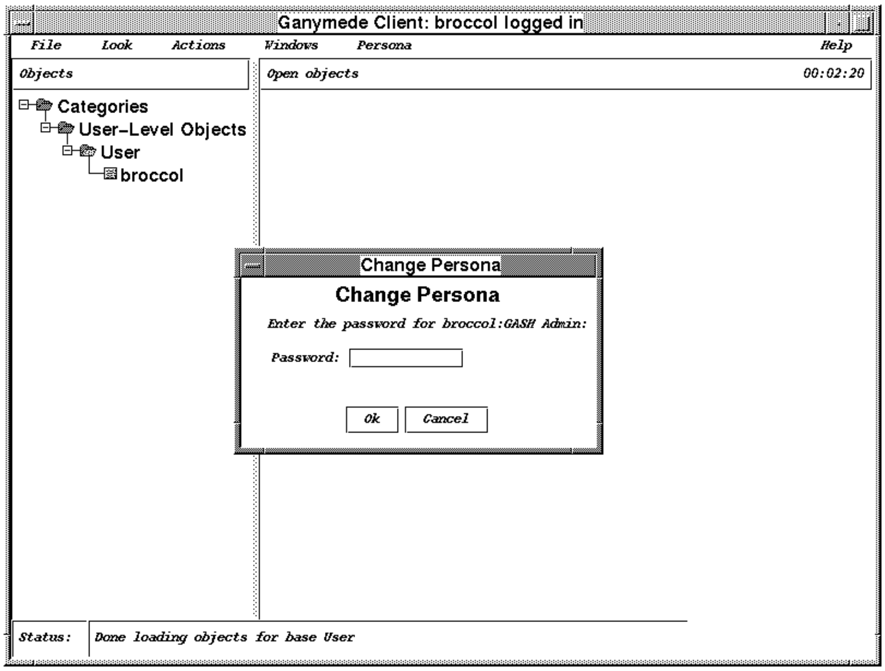


Figure 18: Admin Password Dialog.

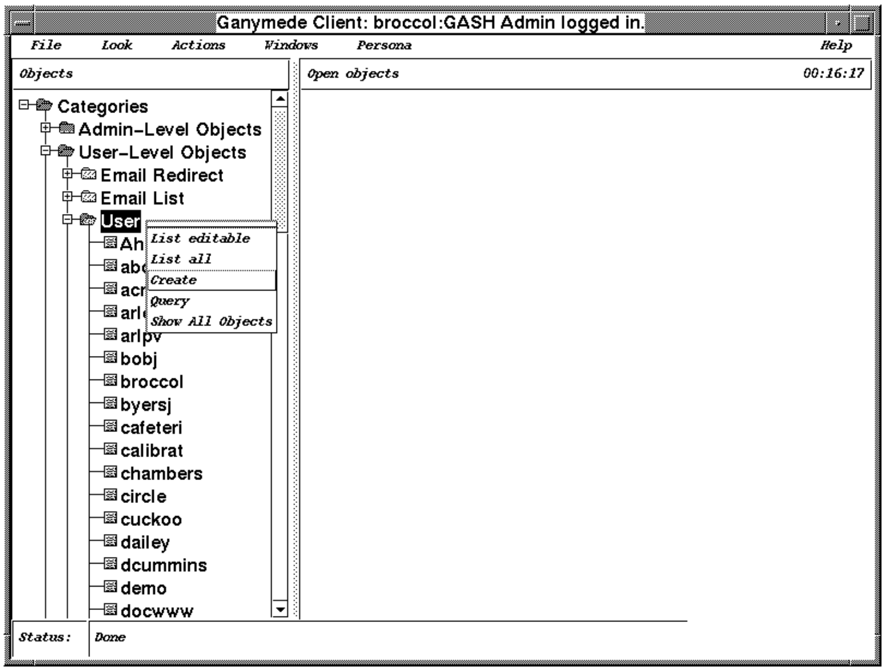


Figure 19: Object Menu.

complex series of wizards to gather more information about a complicated action.

Sometimes changes to the database require more information than can be provided by simply editing a single field. In such cases, the server can send a definition for a custom dialog to the client. The dialog may include graphics, text, and a range of GUI fields. The information from this dialog is sent back to the

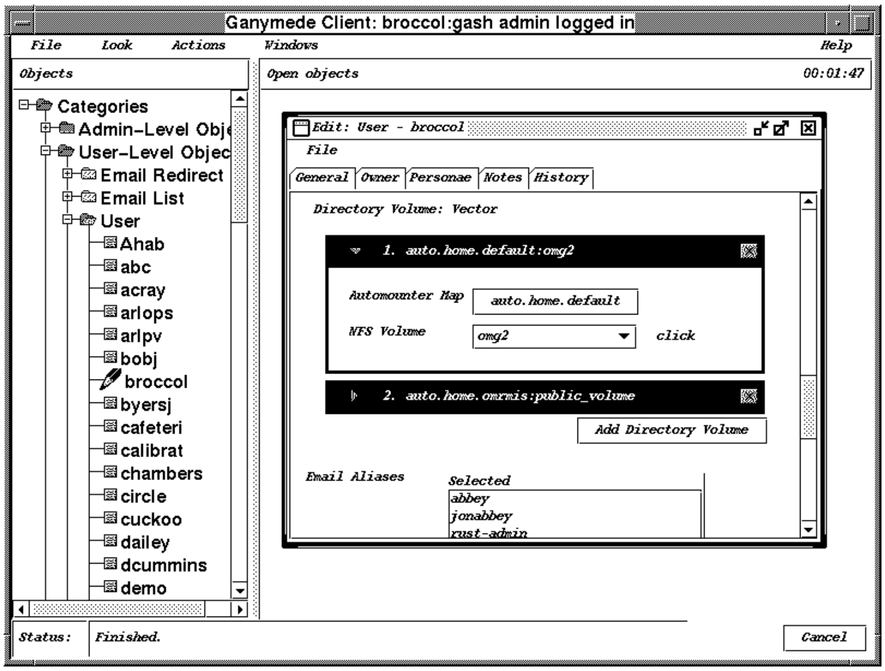


Figure 20: Editing an Embedded Object.

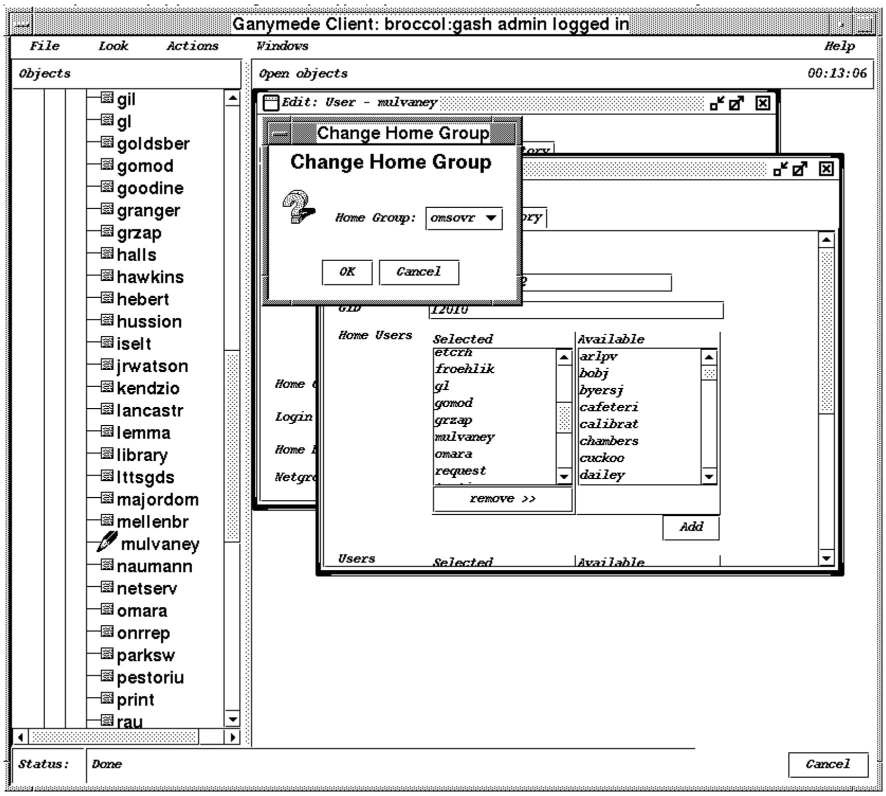


Figure 21: Picking a New Home Group.

server, which may in turn send down a new dialog to continue the discussion. In this way the server can walk the client through a series of steps using the traditional GUI wizard.

For example, when editing a group, removing a user reference from the **home users** field de-selects that group as the user's **home group**. This is demonstrated in **Figure 21**. In the GASH Schema, a user must have a home group, so a new home group must be chosen. In order to find out which group the user object should now have as the home group, the server sends a dialog with a list of the user's current groups to the client.[3] After choosing a group from this list the server places the user in the new home group, and completes the operation initiated when the user tried to modify the home users field.

Queries

The Ganymede server provides a powerful and flexible query mechanism. The client uses this system behind the scenes in many places. The client uses the server's generic query mechanism when it loads objects into the tree, for example. Users can create their own queries as well, using the **query box** shown in **Figure 22**.

The query box allows the user to build rather complex queries. The query box is interactive, and uses the client's cache of schema information to guide the user as to what fields and choices are available. Multiple query terms can be put together to specify as narrow a search as is desired. The user can further customize the query by indicating what fields should be returned in the server's report. When the query is submitted, the results are displayed in a table, like the one in **Figure 23**. The table contains one row for each object matching the query, and displays the values of the fields in the columns of the table. The table allows for sorting on each column, rearranging column

[3]Actually, the server only does this if the user belongs to two or more other groups. If the user only belongs to one group, that group is designated as the home group. If the user does not belong to any other groups, the operation will not be permitted. All of this logic is coded into the GASH schema, which we are using for demonstration purposes in this paper.

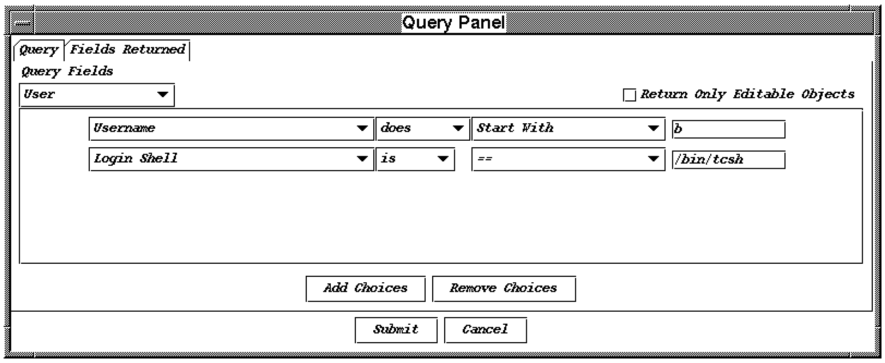
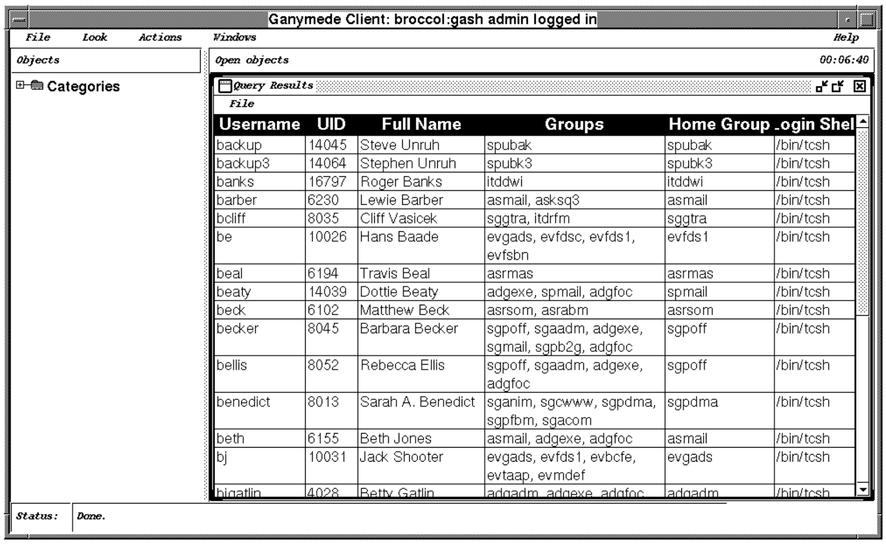


Figure 22: The Query Box.



Username	UID	Full Name	Groups	Home Group	Login Shel
backup	14045	Steve Unruh	spubak	spubak	/bin/tcsh
backup3	14064	Stephen Unruh	spubk3	spubk3	/bin/tcsh
banks	16797	Roger Banks	itddwi	itddwi	/bin/tcsh
barber	6230	Lewie Barber	asmail, asksq3	asmail	/bin/tcsh
bcliff	8035	Cliff Vasicek	sggtra, itdrfm	sggtra	/bin/tcsh
be	10026	Hans Baade	evgads, evfdsc, evfds1, evfsbn	evfds1	/bin/tcsh
beal	6194	Travis Beal	asrmas	asrmas	/bin/tcsh
beaty	14039	Dottie Beaty	adgexe, spmail, adgfoc	spmail	/bin/tcsh
beck	6102	Matthew Beck	asrsom, asrabm	asrsom	/bin/tcsh
becker	8045	Barbara Becker	sgpoff, sgaadm, adgexe, sgmail, sgpb2g, adgfoc	sgpoff	/bin/tcsh
bellis	8052	Rebecca Ellis	sgpoff, sgaadm, adgexe, adgfoc	sgpoff	/bin/tcsh
benedict	8013	Sarah A. Benedict	sganim, sgcwww, sgpdma, sgpfbm, sgacom	sgpdma	/bin/tcsh
beth	6155	Beth Jones	asmail, adgexe, adgfoc	asmail	/bin/tcsh
bj	10031	Jack Shooter	evgads, evfds1, evbcfe, evtaap, evmdef	evgads	/bin/tcsh
bigatlin	4028	Betty Gatlin	adgadm, adgexe, adgfoc	adgadm	/bin/tcsh

Figure 23: The Query Result.

widths, and exporting the contents of the table either to disk or as an email message. Reports can be generated, either as tab- or comma-separated ASCII, or encoded as an HTML table.

Adoption Considerations

We hope that Ganymede will prove attractive to a large community of administrators. We feel that its sophistication, flexibility, and portability will allow it to be widely adopted in a way that GASH could not be. Ganymede is not appropriate for all environments, however. There are issues regarding security and performance that should be taken into account when considering Ganymede's suitability.

Security

While Ganymede has an effective permissions and ownership model, it does not support encrypted communications between the client and the server. At the current time, Ganymede is based on RMI over a non-encrypted socket layer, and as such is not appropriate for use on the open Internet for those who are concerned about packet-sniffing.

Sun is moving to support RMI over custom socket classes in the Java 1.2 JDK, which will make it possible to produce a version of Ganymede that uses Secure Socket Layer (SSL) encryption. Sun has not yet committed to producing a freely available implementation of SSL sockets for Java, but third-party companies have demonstrated RMI over SSL using their SSL implementations [8]. These implementations tend to be very pricey, however, and there are the inevitable U.S. export and licensing issues to consider.

When and if an affordable SSL solution for RMI appears, Ganymede could be retrofitted to provide a higher level of security with very little work.

Performance and Scalability

When we originally designed Ganymede, we set as a goal the ability for Ganymede to scale up to handle 50,000 network object records, a figure some six times higher than we were handling with GASH. The main factors determining how well Ganymede will scale are memory, Java Virtual Machine (JVM) efficiency, and thread and database contention in the Ganymede server.

Memory

Ganymede has limitations on its scalability due to its RAM-resident database. Our current database is loaded with DNS information for 2,087 systems with 2176 total interfaces, 48 administrators, 753 user accounts, 238 account groups, 1200 email alias records, 234 NIS netgroups, 520 NFS volume definitions, 753 automounter entries for home directories, and network connectivity information for 517 rooms. This comprises a total of about a megabyte of GASH data files in text form, some 8,526 objects or so in the Ganymede server.

With this data loaded, the Ganymede server takes up just under 40 megabytes of RAM. Since the machine we are running the Ganymede server on has a gigabyte of memory, scaling up by a factor of six or so would not be a problem for us, at least as far as absolute RAM consumption is concerned.

JVM Efficiency

The biggest problem with such scaling is not the actual consumption of memory, but the time required for the server to handle garbage collection. In an earlier stage of development, the Ganymede server was taking up nearly 100 megabytes of RAM with our dataset loaded, and we would experience frequent and significant lags in the server's responsiveness. We were, however, running the Ganymede server on a relatively slow, relatively loaded machine, a 60-MHz multiprocessor Sparc system, which was also responsible for the laboratory's Web, news, mail and file services. In addition, we were running the JVM in interpreted mode with debug logging enabled, rather than using the JVM's Just-In-Time compiler (JIT) to convert the Java code to Sparc machine code. The latest 1.2 beta seems to be much more efficient, however. We are expecting that a good generational garbage collector, such as is promised with Sun's next generation HotSpot JVM, will do away with concerns over garbage collection overhead, even with a substantially larger database.

The biggest delays that users are likely to encounter will occur when the builder tasks are running. The builder task base class is designed so that the builder task can lock the database while it assembles the data needed for the build, and then release it while it is running external scripts to process the data. While the first phase of the builder task is running, no transactions may proceed to commit. A common source of delay is for a user to make some changes, commit the transaction, then immediately try to make some more changes and commit that transaction. The second transaction cannot proceed to completion while the server is busy writing out data files in response to the first.

Thread and Database Contention

Which brings us to the issue of thread and database contention. The Ganymede server has been designed to try to keep thread contention low. Queries on the Ganymede database are a very frequent occurrence, and the query system has been specially optimized. Any number of clients can issue queries on the database without experiencing thread contention for access to the objects in the database, as long as no transactions are committing. While a transaction is being committed, no queries can be issued on object types involved in the transaction until the transaction finishes committing. Individual object accesses proceed normally, as the server does do table-level synchronization at all times. Queries are guaranteed to be transaction-consistent, and are not processed while a transaction is being committed.

One important thing to note with regards to scalability is that the Ganymede server does not support multiple users simultaneously making changes to an individual object. This is most important with owner groups. If two administrators in the same owner group try to create an object and place it in that owner group, one of them will be unable to do so, and must wait until the other either commits or aborts his transaction so that the owner group is released. Whenever an object is added to or removed from an owner group, such contention can arise. Simply editing an object owned by a particular owner group, however, will not cause such contention.

The answer to this contention problem is to take advantage of the ability of the Ganymede server to support owner group hierarchies. If a group of admins gets large enough that the admins are often getting in each other's way trying to create or delete objects, it is a simple thing to create sub-owner groups that the objects can be placed in without contention.

Current Performance

With the current 1.2 beta 4 JDK we have had five users and an admin console connected to the Ganymede server simultaneously, with each user making changes and browsing the database without noticeable pauses. We do look forward to moving Ganymede to a modern UltraSparc server at some point, but even at the current level of performance, Ganymede provides very acceptable performance for our needs. Right now, we have less than 50 administrators registered in GASH, so it is rather unlikely that more than five admins will ever be using Ganymede at the same time. Once we have fully replaced GASH with Ganymede in the lab, we will be looking to see how well Ganymede scales. Ultimately, we hope to allow our end-users to have free access to Ganymede to handle their own passwords, shell information, and the like. At this time we don't know how heavily Ganymede will wind up being loaded in that environment, but it seems unlikely that too great a number of users would ever try to change their passwords at the same time.

Reflections on Java

When we first started investigating the possibility of developing a next-generation GASH, Java was far less developed than it is today. Version 1.0 was the exciting new thing, and critical features like RMI had not yet been released. But the promise of a sophisticated, operating-system agnostic development platform with widespread industry support was compelling. When Sun released RMI Alpha2 in mid-1996, it became clear that a distributed object design for Ganymede would be possible with Java. At that point, the basic outline of the Ganymede design started to take shape.

Since then, we have developed Ganymede as Java has itself been under development. In some cases we have wound up developing pieces that were not yet available from Sun, such as the tree and table components used in the client and the admin console. In many other cases, such as with RMI, we have found Sun providing just the right thing at the right time to make our work possible. We've had to wrestle with frustrating bugs as Java has matured, but the bugs got fixed. The momentum behind Java, both from Sun and from other companies has continually reinforced the appropriateness of our initial decision to go with Java.

Indeed, we do not believe that we could have done Ganymede with any other technology, given the resource constraints we were under. Java gave us a portable GUI, a distributed object API, a large set of thread-safe class libraries, and both memory and type safety. Most of these things can be had for C or C++, but only for serious money and/or limited portability. We certainly couldn't think of making a freely distributable tool using commercial C++ class libraries. With Java, we were able to write a distributed, multithreaded, portable GUI application of 140,000 lines of code using nothing but X-Emacs and the Java Development Kit.

Concluding Thoughts

Ganymede represents an attempt to provide a comprehensive and flexible management system that can be placed on top of the existing directory infrastructure already in place in typical UNIX networks. It is not designed to answer all the questions about how to organize directory services, but rather to allow adopters to bring their own experience and environment into the design of their directory management tool box. In this respect it differs greatly from GASH, which had a particular network management philosophy hard-wired into its implementation.

Ganymede seems most similar to Novell's Novell Directory Services [9] and Microsoft's forthcoming Active Directory [10] in as much as it provides both a customizable directory database and a set of GUI tools to manage the database. It differs from these in that it does less; the Ganymede server is not designed to act as a high volume directory server. Ganymede does not support database replication or distributed management with multiple Ganymede servers. Instead it is designed to leverage existing directory mechanisms such as NIS, DNS, and commercial LDAP servers which have their own mechanisms for providing reliability and scalability through backup servers. Ganymede depends on the flexibility of scripting mechanisms on UNIX to provide support for getting the directory data where it needs to go. Finally, Ganymede does not provide native support for LDAP or, indeed, any other standard directory API. Ganymede can feed data to servers which support such API's, but administrative programs written to manage directory services using such API's will find Ganymede, on the whole, an incompatible partner.

Ganymede does, however, provide a good solution for the medium-to-large-sized UNIX or mixed network. It has particularly good support for group administration, with the permissions system and the mail and logging system designed to facilitate administration teams. In addition, the Ganymede server is customizable in a rather deep way. Not only can the definition of the database schema be customized, but also a considerable amount of intelligence can be placed in the server using plug-in Java classes. This notion of an intelligent server is in keeping with our original design goal for GASH to produce a tool that would make it possible for a broad audience to safely manipulate our centralized directory information. Ganymede can be taught about a particular environment, and will work to keep it in good order.

Anticipated Future Developments

Barring some continuing polishing work, the Ganymede system is complete and ready to use as it is today. Our main goals at this point are to get Ganymede fully implemented within our laboratory, and to get Ganymede documented well enough that people can begin to work on developing their own custom schema kits. Essentially, most things that we can see needing to be done with Ganymede revolve around the development and elaboration of schema kits.

One important development would be to implement secure authentication and encryption for client-server communications. As we mentioned in our security discussion above, this will depend on an affordable implementation of the SSL protocol for Java.

One possible downside of Ganymede compared to GASH is that the Ganymede client requires a GUI display, whereas GASH could be run from a TTY console. It would be nice to have a completely functional and generic text client for Ganymede for those cases when a GUI display is not available.

Availability

Ganymede is currently available in pre-release at ftp://ftp.arlut.utexas.edu/pub/ganymede/, with documentation, screen shots, and information on joining the Ganymede developer's mailing list at http://www.arlut.utexas.edu/gash2/. While we don't yet have the licensing finalized, we intend to place Ganymede under the GNU Public License. Once we have Ganymede fully implemented in the laboratory and have licensing approved, we will formally release Ganymede 1.0.

Credits

Many people have contributed to the development of Ganymede. In addition to Jon and Mike, Navin Manohar and Erik Grostic made important contributions to the client-side code development. Gil Kloepfer provided design guidance for handling network issues and code for the GASH kit's back-end DNS support. Dan Scott supported the project administratively, and contributed greatly to the higher level design issues in Ganymede, as well as providing invaluable user interface design feedback and bug reporting.

A lot of Ganymede is based on the experience and design work that went into GASH. In addition to the aforementioned names, Dean Kennedy and Pug Bainter should be credited for their design work on GASH. Pug Bainter authored the original GASH makefiles that Ganymede's GASH kit uses to propagate information from Ganymede into NIS and DNS.

Finally, thanks to all of the administrators at ARL:UT and elsewhere who provided feedback on GASH and let it be known that things could perhaps be just a little bit better.

Author Information

Jonathan Abbey initiated the Ganymede project at ARL:UT in late 1995 and has been working on it pretty much nonstop ever since. He graduated with a B.S. in Computer Science in 1993 from The University of Texas at Austin, and has worked at Applied Research Laboratories since September 1989. He is reachable at jonabbey@arlut.utexas.edu .

Michael Mulvaney has worked on the Ganymede project since joining ARL:UT in early 1997, developing the Ganymede client. He graduated with a B.A. in Economics in 1996 from The University of Texas at Austin. He is reachable at mikem@mail.utexas.edu .

References

[1] Abbey, J. "The Group Administration Shell and the GASH Network Computing Environment," *Proc. LISA VIII*, September, 1994.

[2] Sun Microsystems, http://java.sun.com/products/jdk/rmi/ .

[3] Stern, H. *Managing NFS and NIS*, O'Reilly & Associates, Inc., 1991.

[4] Albitz, P., Liu, C. *DNS and BIND*, O'Reilly & Associates, Inc., 1993.

[5] Loomis, Mary E. S., *Object Databases: The Essentials*, Addison Wesley, 1995.

[6] Hogan, C., Cox, A., Hunter, T. "Decentralising Distributed Systems Administration," *Proc. LISA IX,* September, 1995.

[7] Object Management Group, http://www.omg.org/ .

[8] http://www.java.sun.com/products/jdk/1.2/docs/guide/rmi/SSLInfo.html .

[9] Novell Corp., http://www.novell.com/products/nds/index.html .

[10] Microsoft Corp., http://www.microsoft.com/ntserver/basics/future/activedirectory/ .