Graffiti Server — Design and Implementation

Technical Report UCSC-SSRC-07-02

Mark W. Storer
mstorer@soe.ucsc.edu

Storage Systems Research Center
Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

January 23, 2007

Graffiti Server — Design and Implementation

Mark W. Storer
mstorer@soe.ucsc.edu

Abstract

While data on file systems and data on the Web have traditionally been organized in a hierarchical structure, tagging has emerged as a viable technology for dealing with large collections of data. Tagging involves attaching descriptive keywords to data objects such as files and URLs. Most current implementations of tagging restrict the scope of tags to the website or application in which they were created. We have designed and implemented the Graffiti system to explore the collaborative use of tags across applications, computers and users. The Graffiti system is made up of two key components. The first is a client application which the user utilizes to manage the tags on their local file system. The second is a server application that enables collaborative metadata management and sharing.

The Graffiti server constitutes a back-end database and a server application which provides Graffiti clients with access to shared metadata. This document describes the design and implementation of the current version of the Graffiti server. It includes a complete description of the current installation of the server as well as detailed instructions for extending the capabilities of the server.

1 Project Introduction

Locating data that resides on file systems has traditionally been very different than finding web-pages. Locating data on the local file system is closely tied to the act of "filing" data. In contrast, locating data on the Web is more closely related to "finding". The disparity between locating data on local systems and the Web can be tied to fundamental differences in the way the user interacts with the system. Recently however, tagging has emerged as a new model for locating data on the Web and aspects of it may be applicable to local and shared file systems. To determine how tagging techniques might be applied to file systems we have create a tool called Graffiti which adds tagging capabilities to existing systems and allows us to collect usage data about how users utilize tagging in file systems.

On file systems, users usually store files in a hierarchical structure with the hope that this careful placement will make latter retrieval easier. For example a user may place a new document they are working in a directory named, `\home\myhome\documents\project1`. When it comes time to retrieve this document the user can rely on the structure of their folder hierarchy to know that within their home folder they probably stored their documents in the folder `documents` and if the document pertains to `project1` they have a good idea as to where they placed their file. In this "filing" scenario the user spends time to carefully place their data in a location that they can easily deduce later.

In contrast to the "filing" model used on file systems, locating data on the Web can best be described as "finding". In this model, since the user is not responsible for placing the data, they must deduce the data's full location based on what they know about its content. In a simple usage model users utilize search engines to find websites and lists of favorite URLs to easily return to the site at a later time. Traditional search engines on the Web, as well as in recent file systems, concentrate on automatic indexing. The main challenge of this approach is to rank matching results to a query. On the web, Google's PageRank [7]is addressing this by taking the link structure of the web into account. Based on anecdotal evidence, this search technique worked so well that sets of a few keywords became shorthand for URLs and greatly diminished the value of maintaining personal bookmarks.

Recently a new model, tagging, has emerged for finding data on the Web. Tagging, in the general case, consists of attaching descriptive text to objects. Many applications of tags have been focused on helping users locate data. Websites such as Flickr [2] and del.ico.us [1] have demonstrated that tagging can effectively replace hierarchal organization schemes and can efficiently organize large collections of data. Using similar, methods applications such as Apple Computer's iPhoto [3] have included tagging capabilities to help manage collections of data on the local computer. One common drawback that these schemes have is that they do not extend beyond the scope of a single application or website.

Tags can be applied to many problems besides data location. For example tags can be used to identify files that are to be included in an action. In this scenario

a user might attached a tag, for example `backup`, to a file. A backup program written to utilize tags would then search the system for that tag and backup the resulting files. Another example of inclusive tags might be a program designed to automatically copy all files tagged `synchronize` to all the machines that a user has an account on. In contrast to inclusive uses, exclusive uses might specifically tells a program to omit a file. An example of this exclusive model is a user that wishes to have a file left out of an index. In this case the indexer could be configured to ignore all files tagged with `private`. Tags might also be used to manage dependencies. For example, a library file might be tagged with the name and version of each application that utilizes it. In this manner, if all of the applications represented by the tags are no longer on the system a user would know that it is safe to remove the library.

2 Related Work

Our work has been inspired by the continued success of collaborative tagging services on the Web such as del.icio.us [1] and flickr [2]. Additionally, on the local file system tagging has been succesfully applied to assit in managing collections of similar data. For example, Apple's iPhoto [3] utilizes tags to manage digitial photographs and indev's MailTags [8] applies tagging to the problem of managing email. The key difference betweens these solutions and Graffiti is that Graffiti is a general use tagging tool that manages disparate data and spans the boundaries of file systems, hosts and users. We are currently not aware of work that implements collaborative management of file metadata that works across file systems, hosts and applications. There has however, been work done on invesitgating the use and implications of collaborative metadata [6, 4].

The Scientific Annotation Middleware (SAM) [9] system is using a combination of WebDAV and content management to administer a large variety of scientific data and metadaata. Its design assumes that data and metadata is stored in the databases of content management systems within a data grid framework. Graffiti, on the other hand, focusses on the metadata of files stored in file systems.

The Presto document management system extends traditional file systems with arbitrary attributes [5] that allow files to be grouped and searched by these attributes. The system presents itself as a file system and can mount other file systems via NFS and extend them with Presto functionality. Thus, Presto's approach to providing metadata across multiple file systems is accomplished by a layered architecture that duplicates and mimics traditional file system functionality in addition to extended Presto functionality. In contrast, Graffiti maps directly to file content independent of any particular file system structure and strictly compliments traditional file system functionality. In addition we focus on collaborative management of metadata across many users.

3 System Overview

Graffiti provides file systems with tagging capabilities. Using the system, clients use a local application to attach descriptive text strings that they create to files. These tags are stored in a persistent database on the local computer. The Graffiti client application allows these tags to be accessible by applications through a published API. These local tags can also be synchronized with Graffiti servers. The Graffiti servers aggregate tags from multiple client in a centralized database and allow tags to be shared across hosts and with different users.

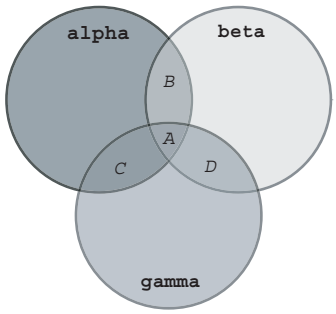
3.1 Files

In the Graffiti system tags are attached to files. The system uses a file's SHA-256 hash as a unique global identifier. This model has three distinct advantages. The first is that users may have multiple copies of the same files on the same computer. The second, and probably more common, is that users may have multiple copies of the same file on different computers. For example, a user may have a copy of an important file on both their laptop and desktop or on their home computer as well as their office computer. The third reason that checksum are a useful global identifier is that Graffiti works across computers and across different users. Users may have files in common and the checksum as a global identifier is a useful way of facilitating metadata sharing and tag suggestions.

There is, however, one important issue related to using the SHA-256 hash as a file's global identifier. A file's hash is dependent on its content and thus if a file changes, its global identifier changes. Thus, it is the responsibility of the Graffiti client components to insure that, as files changes, older tags migrate to the new file checksum. This disadvantage does have some useful semantics. Users may have shared files but they may diverge and the content-based file identifier would reflect this. For example, two users may download a document from the same source but those two users might make separate changes to their copy.

3.2 Tags

Tags are descriptive strings that user attach to files. In the Graffiti system they are user defined strings with no white spaces. This provides a high degree of flexibility in what can be expressed. For example, text strings can be used



file *A* : {alpha, beta, gamma}

file *B* : {alpha, beta}

file *C* : {alpha, gamma}

file *D* : {beta, gamma}

Figure 1: *Relationship implied with tags that is difficult to express using a hierarchical structure.*

to express URLS, key/value pairs, email addresses, or file paths. The Graffiti system does reserve certain tags as system tags which dictate the system's behavior. Examples of these system tags include those that inform the server of a files location and tags that control the server that the metadata is synchronized with.

Tags can express an implied relationship between files when multiple files have a tag in common. For example, it is implied that two files that both contain the tag `ProjectAlpha` are part of the same project. The same relationship would be implied if they were in the same folder. The relationship tags express differ from that implied by folders in that tags allow relationships that are non-hierarchical. Figure 1 illustrates a relationship between a set of files that could not be expressed using a hierarchical structure such as folders and subfolders.

3.3 Client Application

Users of Graffiti interact with the system through a client application. The client runs on each host computer that supports tagging and provides the system with a local metadata store and an interface for the client to interact with. The design of the Graffiti client provides a number of advantages. The client exists at the application layer. This design choice allows Graffiti to work on a wide variety of client platforms and files systems. In addition to the time it takes to implement tagging inside a file system, it also presents a risk to the data within that file system. Additionally, as one of the purposes of Graffiti is to test new metadata primitives, Graffiti allows the users to work with their current data in its current state and location.

The Graffiti client exports an API interface that allows any application to take advantage of the tags maintained by the local database. Many of the currently existing applications that utilizing tagging restrict the tags to the application they were made in. Graffiti attempts to remedy this and to encourage users to utilize tagging by increases a tags utility. If the tags exists in multiple applications they are more valuable than tags restricted to a single application.

The Graffiti client includes both GUI and command line tools for attaching tags to files. The tags users attach to files are instantly available locally as they are maintained in a local database. If clients would like these tags to be available to other users or on their other computer they can synchronize their tags to a Graffiti server. Clients can choose which servers they share their tags with using a special system tag which identifies a Graffiti server.

Clients communicate to the Graffiti server through secure HTTP. All communication between the client and the server is stateless. Each call that the client makes to the server is self-contained with the request containing all required information needed to authenticate the user to the system and fulfill the request. Currently, users are required to have a password-protected account on every server with which they communicate.

3.4 Server Application

The Graffiti server maintains a database of metadata aggregated over a number of clients and enables the sharing of metadata across computers. The metadata stores on the server record a username along with the file metadata. Utilizing this username along with a time-stamp of synchronizations, the server is able to provide a user with multiple computers the ability to synchronize metadata changes across multiple hosts. An example of this functionality would be a user with a laptop and a desktop and a set of files that exists on both. Solutions such as CVS can effectively manage the synchronization between the two computers but largely ignore the file's metadata. A user can tag a file using the Graffiti client on their laptop and synchronize this change to a Graffiti server. Later, that user can synchronize their desktop to that Graffiti server and have that tag automatically added to their desktop's local Graffiti metadata database.

4 Implementation Overview

The Graffiti system consists of a Graffiti client that runs on the users local machine and Graffiti servers that serve as centralized points for metadata collaboration.

4.1 Client Overview

The Graffiti client was built in Java using the SWT toolkit for the GUI. Data on the client is managed via an embedded Apache Derby database that is accessed through SQL

queries. The GUI and command line both access the data store through an API which allows common tagging operations, and is available to other applications. For now, filesystem event handling such as updating the database when files are renamed is only implemented for Mac OS X.

4.2 Server Overview

The Graffiti server is based on a database back-end and an API that clients access through secure HTTPS calls. The server has two primary roles. The first role of the server is to enable the collaboration of metadata across multiple machines. This is accomplished through the database and the API. The second role of the server is to collect usage data about collaborative metadata. This is done through aggressive event logging at the server and database levels.

The back-end database provides a persistent data-store for collaborative metadata. It is implemented using PostgreSQL version 7.4.11. Currently, each user that accesses the Graffiti server has an account managed by a special Graffiti users database table. In the future we may look into providing centralized authentication capabilities. The database has four sets of data to manage. The first is the set of user accounts for that server. The second is the set of files, identified by checksum, that are owned by users. Third, the database tracks the tags that have been placed by users on files. Finally the server is able to manage metadata that users choose to share.

The Graffiti server was implemented using the Twisted server framework version 2.1.0 and Python version 2.3.4. The clients makes calls to the server as HTTP requests. The server uses basic HTTP authentication along with a table of users in the database to authenticate Graffiti users.

One of the important tasks of the Graffiti server is to collect usage data based on interaction with Graffiti clients. This is accomplished at two levels. The first is logging at the database level. The second is logging at the server level. The logging at this level is accomplished using the Python `logging` module. This module allows the content of log messages to be separated from the messages presentation.

Clients interact with the server through a published API. The API receives requests from the client over HTTP. The API currently consists of five calls. The first call, `putTagChanges`, allows a user to update the server database with the tag changes that have been performed at the client. The second call, `getTagChanges`, allows a client to retrieve the tag updates that they have placed on the server. The server retrieves the tag changes based on a date passed as an argument and the username attached to the request. The server returns a list of all the tag changes that the user has made on any machine after the given date. The third call, `clearAllTags`, allows the user to reset all their tags in the server database. The server accomplishes this by turning off all the tags for the user and setting the modification time to the earliest possible system time. The third and fourth calls, `putSharedTags` and `getSharedTags`, allow a user to share the metadata they have attached to a file with another user. When a user shares their metadata they contact the server and identify the checksum that has the tags the user wishes to share. The server responds with an identification number that another user can use to collect the shared metadata.

4.3 Database

The current implementation of the Graffiti server utilizes PostgreSQL version 7.4.11. The name of the database that the Graffiti server attempts to connect to is `graffiti`. In PostgreSQL the database can be created by issuing the command from the system's command line.

```
createdb graffiti
```

The database can be administered from the command line using the `psql` command. To access the database you would issue the following command from the system command line:

```
psql graffiti [username]
```

If no user name is given then the default username is used. Within the psql interactive command terminal commands are terminated using the ';' character. Commands can span several lines. To exit the psql command terminal use Ctrl-D or `\q`. To access a list of useful PostgreSQL commands use `\?`.

4.3.1 Database Users

There are two database users that the Graffiti server uses. The first, `graffitiserver`, is used for general queries to the database and the other, `graffitiserveradmin`, is used to manage users. In PostgreSQL users are created using the `createuser` command at the system command line. The Graffiti server uses two different user accounts so that security can be more fine-grained.

The `graffitiserveradmin` account is used to manage user accounts. It has permissions that allow it to modify the tables related to user accounts. In the current Graffiti server implementation it is used in the `adduser.py` script which walks through the process of creating a Graffiti account. This is the only account

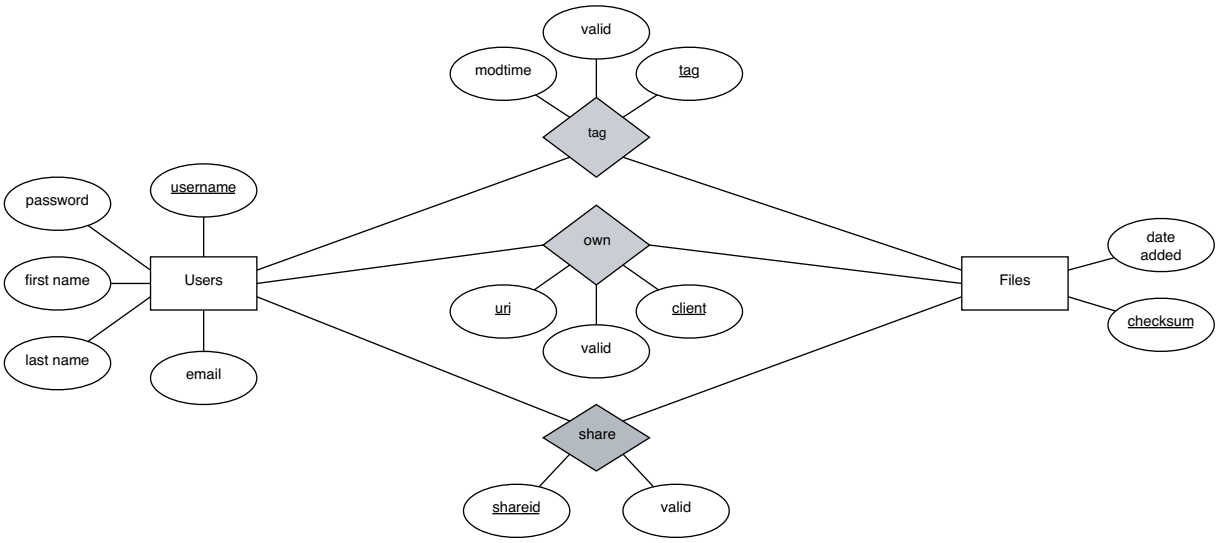


Figure 2: *ER model diagram of the Graffiti server database schema*

User	Permissions	Table
graffitiserveradmin	SELECT,INSERT,UPDATE	users
graffitiserver	SELECT	users
graffitiserver	SELECT,INSERT,UPDATE	user_files
graffitiserver	SELECT,INSERT,UPDATE	tags
graffitiserver	SELECT,INSERT,UPDATE	file_ownership
graffitiserver	SELECT,INSERT,UPDATE	tag_sharing

Table 1: *Table permissions needed by the Graffiti server database accounts. Note that no database user account needs* `DELETE` *permissions for any of the tables.*

that needs permissions to modify the `users` table as described in section 4.3.2. It does not, however, have any access to any other table as it is only used for managing user accounts.

The other account, `graffitiserver`, is used to modify the contents of every table in the database with the exception of the `users` table. Since it is used by the authentication module it does, as table 1 illustrates, have select access to the `users` table and select, insert and update access to all of the other tables. Table permissions can be granted to an account using the GRANT command from within the database shell.

4.3.2 Users Table

Each user that authenticates to the Graffiti server has an entry in the `users` table. Authentication involves two fields in the table: username and password. The password is stored in the table as an MD5 hash of the actual password text chosen by the user. The remainder of the fields in the table store information about the user such as their actual name and contact information. The SQL command for creating the table is as follows:

```
CREATE TABLE users (
  username   VARCHAR(32),
  password   VARCHAR(32),
  first_name VARCHAR(32),
  last_name  VARCHAR(32),
  email      VARCHAR(32) NOT NULL,
  PRIMARY KEY (username))
```

4.3.3 User Files Table

The second entity in the server schema is a user file. These are stored in the `user_files` table. User files are uniquely represented in the system by their checksums. The table consists simply of the checksum and the date that they were added to the system. The SQL command for creating the table is as follows:

```
CREATE TABLE user_files (
  checksum   VARCHAR(64),
```

```
  date_added TIMESTAMP DEFAULT now(),
  PRIMARY KEY (checksum))
```

4.3.4 Tags Table

Tags are central to the Graffiti system and in the server schema are described in the ER-model as, "Users tag Files". The tag itself has attributes such as the string that makes up the tag, its last modification time and whether the tag is valid or not. Tags are never deleted from the system for three reasons. The first is that users would need to have access rights to delete data and this was considered a needless security risk. The second reason is that one of Graffiti's primary roles is to provide usage data and as such operations such as deletion should be recorded in a way that leaves a clear record. The third reason is related to synchronization. Deleting a tag by removing its entry from the table results in the need to log actions taken by the user that can be consulted to get synchronization data. In contrast, in the current model deletion is simulated by setting the validity of the tag to *false*. Using this technique along with the modification time it easy to determine the changes that have occurred since a given time.

The constraints on the `tags` table insure that the username that owns the tag as well as the checksum that the tag is placed on are present in the `users` and `user_files` table respectively. The SQL command for creating the table is as follows:

```
CREATE TABLE tags (
  modtime  TIMESTAMP DEFAULT now(),
  tag      VARCHAR(256),
  valid    BOOLEAN DEFAULT TRUE,
  username VARCHAR(32),
  checksum VARCHAR(64),
  PRIMARY KEY (username,checksum,tag),
  FOREIGN KEY (username)
    REFERENCES users,
  FOREIGN KEY (checksum)
    REFERENCES files)
```

4.3.5 File Ownership Table

The relationship between users and files is described in the ER-model as, "Users own Files". This relationship is contained within the `file_ownership` table. This relationship describes not only which file a user owns but also where the owner has placed the file. This location information is related to two attributes on the "owns" relationship. The first attribute is the `uri` field which consists of the fully qualified path to the file. The second attribute is the `client` field which is a descriptive string identifying a host. This client name only needs to be understandable to the owner of the file and does not relate directly to a hostname (although a hostname would be a logical choice for the client field). This description is assigned by the file owner.

As with tags, the constraints on the `file_ownership` table insure that the username that owns the tag as well as the checksum are present in the `users` and `user_files` table respectively. The SQL command for creating the table is as follows:

```
CREATE TABLE file_ownership (
  uri      TEXT,
  client   VARCHAR(64),
  username VARCHAR(32),
  checksum VARCHAR(64),
  valid    BOOLEAN DEFAULT TRUE,
  PRIMARY KEY
    (username,checksum,uri,client),
  FOREIGN KEY (username)
    REFERENCES users,
  FOREIGN KEY (checksum)
    REFERENCES files)
```

4.3.6 Tag Sharing Table

The third relationship found in the server's database schema relates to the collaborative aspects of Graffiti's metadata. It is described in the ER-model as "Users share Files". The relationship is actually a bit misleading as the relationship describes not the sharing of file data but rather of file metadata (in this case tags). In the current usage model the user chooses to share the metadata attached to a checksum and receives a token. The user that the metadata owner wishes to share the data with is given the token which the receiver redeems at the server. In the current implementation the token is an identification number. The `tag_sharing` table relates a checksum to the identification number.

As with the other relations, the constraints on the `tag_sharing` table insure that the username that owns the tag is in the `users` , the file being shared is in the `user_files` table and the user sharing the file owns the checksum according to the `file_ownership` table. The SQL command for creating the table is shown as follows:

```
CREATE TABLE tag_sharing (
  shareid  INTEGER,
  valid    BOOLEAN DEFAULT TRUE,
  username VARCHAR(64),
  checksum VARCHAR(64),
  PRIMARY KEY (checksum, shareid),
  FOREIGN KEY (username)
    REFERENCES users,
```

```
FOREIGN KEY (checksum)
  REFERENCES user_files)
```

4.4 HTTP Server

The Graffiti server users HTTP to communicate with the clients. The web server portion of the server is implemented using the Twisted 2.1.0 package and Python version 2.3.4. Starting the server involves running the `graffiti.py` file through the Python interpreter as follows:

```
python graffiti.py
```

The server's main loop is implemented in the `graffiti.py` file. This is the file that directly implements the Twisted framework's factory classes. The primary class that handles a secure HTTP request is the `FunctionHandledRequest`. This class' `process` method is automatically called when the server receives an incoming request.

The first thing that the server does when it receives a request is to authenticate the user. The details of user authentication are explained in section 4.4.1. If authentication succeeds, server control is passed to the `graffitiHandler` method of the `graffitiServerAPI.py` file.

The `graffitiServerAPI.py` is responsible for taking the request from the user and, utilizing the URL, determining the API call requested and the handler for that call. This module exports one method which uses a lookup table to match the user's request to the API call and handler. Extending the number of API calls is a three step procedure as follows and should not require any changes to the `graffitiHandler` method.:

1. Implement a Python module which performs the new call.
2. Add an import for the module's entry point to the `graffitiServerAPI.py` file.
3. Edit `graffitiServerAPI.py` by adding a lookup table entry for the new API call

4.4.1 Authentication

Authentication in the Graffiti server is handled using basic HTTP authentication. In the server implementation, all of the code for handling the authentication is located in the file `graffitiauth.py`. This file implements a single method, `authenticateUser`, which takes the username and password associated with a request and confirms that these values agree with values in the Graffiti server database. If the values are correct than the method returns the username back to the caller, otherwise it raises an authentication exception.

Currently, the user must have a local account on each server that they access. This could easily be extended by updating the `authenticateUser` method to access a centralized authentication server.

4.4.2 HTTPS and Certificates

In an effort to provide transport level security, the Graffiti server performs all communication over HTTPS. As such, the server requires a private key and a certificate. In the current implementation the keys and certificates required by HTTPS are generated using the `openssl` command. In the current implementation, the server's certificate is not signed by a valid certificate authority and thus would not be automatically trusted by a generic HTTPS client such as a web browser. For the purposes of this project, that is not an issue.

The private key required by the Graffiti server is generated using the `genrsa` command and the OpenSSL command line tool. The final argument specifies the key length as a bit length. The current implementation uses a key of length 1024 but this is somewhat arbitrary. The command used to generate the key is as follows. It outputs the key in a file named `graffitiServerKey.pem`.

```
openssl genrsa -out
    graffitiServerKey.pem 1024
```

The certificate for the Graffiti server is produced using the `req` command and the OpenSSL command line tool. The `req` command is an interactive command line tool that walks the user through the process of creating an x509 certificate. The command used to generate the certificate is as follows:

```
openssl req -new -x509
   -key graffitiServerKey.pem
   -out graffitiServerCertt.pem
   -days 1095
```

4.5 Logging

One of the goals of the Graffiti project is to collect usage information for tags. To this end, the Graffiti server utilizes Python's `logging` package. This package separates the content and presentation of the log messages and provides a single point of configuration for log messages.

In Python's `logging` package, `Logger` objects are used to generate log messages. These `Logger` objects are not passed as variables but rather accessed by name. Configuration of logging within the Graffiti server is centralized to the `graffitiLogger.py` file. This module

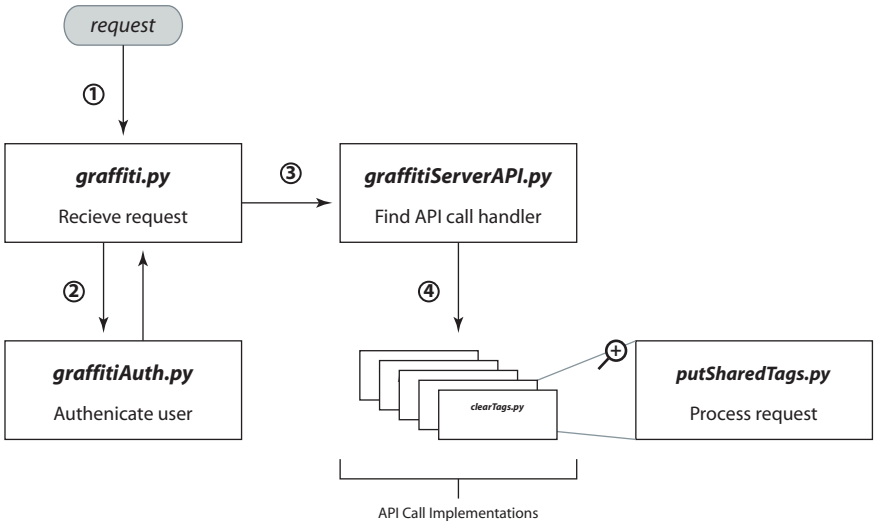


Figure 3: *Request processing*

implements one method, `setupLogging`, that is called from the `graffiti.py` just before the server starts. It configures one `Logger` object named `graffiti`. Each user module that wishes to use this logger must import the `logging` module and retrieve the logger by name using the following command:

```
log = logging.getLogger("graffiti")
```

While `Logger` objects are used to generate log data, presentation of that data is left to one or more `Handler` objects. These `Handler` objects represent output streams such as files and standard output. The advantage of the Python logging scheme is that each `Handler` can be configured independently but in one central location. For example, in the current implementation there are two handlers. The first is the standard error stream which outputs server errors. The second is a file stream which produces a detailed log of all the requests made to the Graffiti server.

4.6 API Calls

All API calls to the server are issued as URL's in which the call is specified as the requested resource. Arguments are passed in the CGI format for URL arguments. The general form for API calls is as follows:

```
https://<server>:<port>/<call>?<args>
```

4.6.1 getTagChanges

This API call gets all the tag changes associated with the client that have occurred after the given time. The `getTagChanges` call is implemented in the `getTagChanges.py` file. The tag changes are interpreted and not log based. For example, if a tag has been added, and then deleted, and then added again since the timestamp given, the server will respond with one ADD operation. Thus, the tag changes returned by the server represent the current state of the tags and not the sequence of events that brought it to its current state. This call takes the following arguments:

Arg	Status	Description
year	req.	timestamp's year field
month	req.	timestamp's month field [1,12]
day	req.	timestamp's day field [1,31]
hour	opt.	timestamp's hour field [0-24]
min	opt.	timestamp's minute field
sec	opt.	timestamp's seconds field

The server responds to the client request with an HTML encoded page. It is the client's responsibility to parse the contents of the page. An example page that returns a log of two operations would have the following format:

```
getTagChanges
User: <username>
Timestamp: <timestamp given by user>
<ADD|DELETE>, <tag>, <checksum>
<ADD|DELETE>, <tag>, <checksum>
```

4.6.2 putTagChanges

This API call is used to put the client's metadata information on the server. The `putTagChanges` call is implemented in the `putTagChanges.py` file. It is used to

place both tag and file ownership information on the Graffiti server. The tags are connected to a checksum and the server does not attach the old tags if a checksum changes. It is up to the client to keep the server up to date and keep old tags moving forward as the checksum changes.

Each tag placement request is passed to the server in the url string as a three tuple labeled `req` where the tuple is of the following form:

```
<ADD|DELETE>,<tag>,<checksum>
```

File ownership is expressed through the use of a special tag. If the tag is a file url (i.e.: `file://`) then an entry is made in the server's `file_ownership` table associating that user with that checksum at the client and path given. For example, if the user has the file with a checksum of `mummlemummle` the special request might look as follows:

```
ADD,file://home/docs,mummlemummle
```

When the server receives a file, certain sanity checks are made. First, the server will insure that the checksum is located in the file table. If it is not then the server will add it to the file table. Second, if the request is to add a tag that the user has already associated with the file, the server will insure that the valid field is set to true. This may occur if a user adds a tag, then deletes it and then chooses to add it again.

4.6.3 getSharedTags

This API call fetches the tags associated with a shared checksum and then sets the share to invalid so that it cannot be fetched a second time. The `getSharedTags` call is implemented in the `getSharedTags.py` file. This call takes one argument, `shareid`. The server responds with the following output if the shareid is valid and the validity of the shared checksum is set to true:

```
getSharedTags
User: <username>
<checksum> : <tag>
```

If more than one shareid is given, all the tags for all the valid shareids will be fetched and output.

4.6.4 putSharedTags

This API call adds the checksum to be shared to the server's tag_sharing table and returns a nonce which is used to get the shared tags. The `putSharedTags` call is implemented in the `putSharedTags.py` file. Since the user shares a checksum and not specific tags, all of a users tags attached to the checksum are shared. This method takes one argument, `checksum`. The server replies with the following output:

```
putTagChanges
User: <username>
<checksum> : <shareid>
```

If there is an error during the database interaction or if the user does not own the file according to the database the shareid returned will be -1.

4.6.5 clearAllTags

This API sets the validity of all of a user's tags to false and then resets their modification date. The `clearAllTags` call is implemented in the `clearAllTags.py` file. It is important to note that since the server is implemented in Python the new modification date is not the start of epochal time but rather the start of AD time, "0001-01-01 00:00:00"

It is also important to note that there is no confirmation when the client makes the call. The user that is authenticated will have their tags reset on the server so the client must be careful making this call.

5 Server History

The first version of the Graffiti server was implemented using Java 5 and, instead of HTTP, relied on Java's Remote Method Invocation (RMI) for exporting its API to clients. This approach proved to be inflexible as the authentication and secure communications capabilities were insufficient for our application. In spite of RMI's usage over CGI we were unable to produce a suitably secure version in an acceptable amount of time. One advantage that this method did have was the ability to pass arguments and return results as Java collections. While this reduced the amount of parsing required on the client's end, this alone was considered a poor tradeoff for the communications shortcomings.

The Java version of the Graffiti server was first implemented using the MySQL version 3.23 database. Two issues prevented the further use of this database. The first was this version's inability to use arbitrary text fields as table keys. As figure 2 shows, the database schema for the Graffiti server utilizes a number of tables that use text fields as keys. The second issue that caused the move to PostgreSQL was MySQL version 3.23's poor support for foreign keys. This particular feature was considered important for maintaining the consistency of a number of tables. For example, it was important to insure that users were tagging and sharing files that existed in the database.

6 Conclusion

Tagging has proven to be a useful method for organizing collections of data. Its use has been proven on websites

as well as in local applications. The Graffiti system has been developed to experiment with tags that cross application, host and user boundaries. A key component that enables this capability is the Graffiti server which makes tags available across host boundaries and promotes tag sharing and collaboration.

References

[1] del.icio.us. http://del.icio.us, Nov 2005.

[2] Flickr - photo sharing. http://www.flickr.com, Nov 2005.

[3] Apple Computer Inc. iphoto.

[4] T. Butler, S. Fisher, S. Hockey, G. Coulombe, P. Clements, S. Brown, I. Grundy, K. Carter, K. Harvey, and J. Wood. Can a team tag consistently? experiences on the orlando project. *Markup Languages: Theory and Practice*, 2(2):111–25, 2000.

[5] P. Dourish, W. K. Edwards, A. LaMarca, and M. Salisbury. Presto: an experimental architecture for fluid interactive document spaces. *ACM Trans. Comput.-Hum. Interact.*, 6(2):133–161, 1999.

[6] S. A. Golder and B. A. Huberman. The structure of collaborative tagging systems. Technical report, Information Dynamic Lab, HP Labs, 2005.

[7] M. Henzinger. The past, present and future of web search engines. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 2004.

[8] S. Morrison. Mailtags, 2005.

[9] J. D. Myers, A. R. Chappell, M. Elder, A. Geist, and J. Schwidder. Re-integrating the research record. *Computing in Science and Engineering*, May/June 2003:44–50, 2003.