

# Distributed User Management Tools: Developing Kiva::User and McFeely

*Chris Dent*  
*Independent Professional*  
*Adrian Hosey*  
*Kiva Networking*

*Matt Liggett*  
*Independent Professional*  
*Jeremy Fischer*  
*Kiva Networking*

## Abstract

Kiva::User and McFeely are software tools developed at Kiva Networking, a medium-sized Internet Service Provider in Indiana. They were created to facilitate the automation of user management across a distributed network of servers. Kiva::User provides a Perl based object-oriented representation of users. McFeely provides a generic tool for performing related operations on a variety of machines. Both were developed using freely available tools and are themselves freely available.

This proposal outlines the problems Kiva::User and McFeely try to solve, the development history of the tools, the current state of development and ideas for future development. A full report, if written, would provide detailed explanation of the design decisions made in the coding process, show examples of how the tools are used, demonstrate the benefits of using freely available tools and an open development environment, explain more completely scenarios under which this solution has worked well for Kiva Networking and address how it may be generally applicable to users in other environments.

## 1. History

In late 1997 the Systems Department at Kiva Networking determined that in order to properly deal with growth and increase service reliability, Internet services provided to users--such as shell access, web space and mail delivery--would need to be distributed across multiple machines. Before that decision user management had been relatively simple: a home-grown Perl script called `addusers` was able to directly modify configuration files and create home directories. The introduction of the multi-server plan made user management a more complex problem.

### 1.1. The Problem

In a multi-server architecture user generation, modification and deletion requires relatively complex actions on a variety of machines. For example, when creating a user with shell access the following actions may be required: 1) home directory creation on the shell server, 2) mail destination creation on the POP server, 3) home directory creation on the web server and 4) setting quota on any machine on which the user will have filesystem access. When renaming a user the actions are similar but involve renaming existing structures instead of creating them from scratch.

Prior to implementing the multi-server architecture a decision was made: there must be tools to automate the user generation actions or the architecture will fail right

out of the gate. While the spirit of the architecture was well defined, the details of the architecture and of company-wide user management policy were not. Because of this it was further decided that the tools generated must be modular. In addition, where possible, the tools should be as general as possible to allow for easy adjustment as new requirements were discovered.

Early on the developers were aware that at least part of the work they were doing had potential value for other people in similar situations so at the start they sought, and received, permission from the company to release the software under the GNU General Public License. As the company's entire service delivery architecture was based on open software such as the Linux kernel, Apache, and many other packages it was only appropriate that some effort be made to give something back.

## 2. The Solution

After much thought, a few false starts, and much scratching of heads Kiva::User and McFeely were born. Kiva::User is a Perl based object-oriented representation of a user on a Unix-like system. McFeely is a generic distributed task execution architecture. The two, combined, are able to perform many, but not yet all, of the system based user management tasks at Kiva Networking.

Two separate tools were developed because the developers realized that a distributed task execution

architecture would have value throughout the organization. A primary design goal in the project was to find that which could be generalized and make it independently usable.

## 2.1. Kiva::User

Kiva::User provides a Perl based API for creating and modifying users. Users are represented in the system as Perl objects with attributes. Attributes are features or associated with a user. Examples include: the class of user (pop\_only, residential, business, virtual domain); the existence of a shell account; the amount of quota on the web server; and the username, uid, password, shell and GECOS information for the user.

An object may be created from scratch, or created from an existing (real on the computer) user. For existing users attribute data is filled from the system. For new users or users who are gaining attributes as the result of a change, attributes are filled by Implications.

Implications are at the center of Kiva::User. A (somewhat) human readable text file contains defaults for different classes of users. For example, the implications file indicates that a residential class user should get a GID of 100; should have shell, web and pop access and should have certain default values for quota on the shell, web and pop servers.

Implications are used to automatically set the values of some Attributes based on the values of other Attributes. For example, the POP architecture at Kiva Networking places users on different POP hosts based on the first letter of their username. Therefore, an implication exists to set the POP\_Host Attribute based on the value of the Username Attribute. The automatic actions allow the application programmer to populate a Kiva::User container object with a minimal data set and let Implications "do the right thing" by filling in the rest of the object. The application code can stay ignorant of a great deal of service architecture specifics and administrators may make major architecture changes without having to recode multiple applications simply by changing the implications file.

Once a Kiva::User user object is created by an application that uses the API, methods can be called to change the real world user: install(), disable(), enable() and morph(). These in turn call install(), uninstall() and morph() methods on the Attributes which are objects in their own right. Some of these methods add McFeely Tasks to the user object. These Tasks will be injected and performed as the application completes. Other Attributes methods perform actions themselves, such as manipulating the /etc/passwd file, that must be done immediately to avoid locking and race condition issues.

Implications and Attributes make Kiva::User very flexible. If user policy is changed the implications file can be adjusted to reflect new defaults. The changes will be available to programs that use Kiva::User from that point on. If additional Attributes are required (e.g. a flag indicating access to DSL services) the Attributes code can be subclassed to include a DSL attribute.

Many applications can use Kiva::User. At Kiva Networking there are tools for creating users, disabling users, upgrading and downgrading users, renaming users, and tools for presenting user information to customers and technical support staff.

## 2.2. McFeely

When Kiva::User was first put together a remote task execution system called jobq was used to distribute tasks to the required network servers. This tool proved unstable and did not have the features necessary to take full advantage of Kiva::User. When time was available McFeely was created.

McFeely is a client server architecture for the distribution and management of jobs amongst a set of configured machines. Jobs are made up of Tasks which may be ordered and may have dependencies. A Task may fail with a hard result causing the Job to fail or it may fail with a soft result, meaning that it should be deferred and tried again. Tasks can be distributed to machines over ssh tunnels meaning that sensitive information travels in an authenticated and encrypted fashion. A more complete description of McFeely can be found at <http://web.systhug.com/mcfeely>.

In essence, McFeely is a queuing mechanism for arbitrary tasks. Tasks consist of a host, a "comm" and optional arguments. Tasks are collected within Jobs. A protocol, the Task Transfer Protocol (TTP, defined at <http://web.systhug.com/mcfeely/ttp2.txt>), is used to transfer task information to remote machines where the "comm" is executed from a secured environment. A "comm" is a specially designed binary or script which either succeeds or exits with two different error codes: one indicating permanent failure, the other indicating that the task should be tried again later.

Results are returned to the McFeely server which determines whether the Job from which the Task originated has completed, has failed, or has deferred Tasks waiting to be run. Deferred tasks are tried again repeatedly with a growing time interval between each deferral. When a Job completes a success or failure notification can optionally be sent to an email address that is defined per job.

McFeely is both the client-server tools which manage Jobs and Tasks and a Perl API which allows for the creation of Jobs and Tasks. McFeely was designed to be generally applicable in any situation where related actions need to be performed on multiple machines. Examples of how to use McFeely can be found at <http://web.systhug.com/mcfeely/examples.html>.

### 2.3. Example

When used together Kiva::User and McFeely are a very powerful combination. A quick example shows this.

Kiva Networking uses a command called rename-user to change a username on the system. For a username change filesystem and configuration data which refer to the username must be updated. Kiva::User and McFeely take care of all these changes.

rename-user is a thirty-three line perl script which uses the Kiva::User API. The core of the program is shown in the following four lines of code:

```
$user_obj = Kiva::User->new_from_username($user);
$user_clone = $user_obj->clone;
$user_clone->username($nuser);
$user_obj->morph($user_clone);
```

The old username is in \$user and the new username is in \$nuser. A Kiva::User object is created and then cloned. The clone has its username changed to \$nuser and the original user object is morphed to the clone. This action causes morph() to be called on all the Attributes. Some of the Attributes cause McFeely Tasks to be added to the McFeely Job for the user object. In the case of a username change this causes: home directories to be renamed on the shell, web and mail servers; symlinks between the shell home directory and the web server NFS mount to be recreated; and a variety of access control lists to be updated.

This happens in just a few seconds and has proven reliable enough to be a casual act. This is a fabulous improvement: username changes, prior to a mature Kiva::User, were the source of much dread for the Systems department.

### 3. The Future

Kiva::User and McFeely are far from perfect. Through use, continued development and, in the case of McFeely, valuable input from other users around the world, both tools have improved, but there is more that can be done.

In the past year McFeely has become far more reliable because of an ongoing examination of the code. There are several further improvements that are desired: 1) McFeely Tasks can currently return ok() for success,

soft() for deferral, and hard() for failure. In certain circumstances a warn() return value would be helpful to alert an operator of a non-fatal condition. 2) There is currently only the Perl API to McFeely. Interfaces for other languages, including C and Python have been discussed. 3) The current McFeely is very process intensive. Many processes are chained together via pipes and forks. A CORBA based McFeely2 which will allow for remote method invocation is under consideration.

Kiva::User initially only supported user generation and had abysmal error handling. Today Kiva::User has support for disabling, enabling and modifying users. Soon it will have support for deleting user data from the system. Error handling has improved greatly by using eval blocks to trap and effectively pass error messages up to code that can handle the error or to the application where it will be reported. There is more to do in this area.

From a usage standpoint Kiva::User is somewhat slow. When a user object is created a substantial number of methods on a substantial number of objects must be called. This may be improved by autoloading methods as required.

Finally, and perhaps most importantly, Kiva::User needs to be made more generic. Whereas McFeely is readily available for download and use by whoever chooses to use it, Kiva::User is currently not available in any form that is easy to distribute. It has potential value for other Internet Service Providers and other organizations that need to manage the creation and modification of a large number of users.

### 4. Summary

Kiva::User and McFeely have proven immensely valuable to Kiva Networking. Today, adding additional machines to the server architecture or changing user policy is far less of a concern than it was three years ago. Management of those machines and the users on those machines can now be accomplished with a well-defined, stable set of tools and methods.

Accomplishing this took a willingness to see the value of a long term investment. Development of Kiva::User and McFeely was a long and time consuming process but, luckily, the investment paid off.

The developers hope that by sharing their experience, others will benefit; both from their successes and their failures. Kiva::User and McFeely can be valuable to others and will be improved if more people use it, dissect it and tune it up. A full report on the tools will help to bring this about.