Managing User Requests with the Grand Unified Task System (GUTS)

Andrew Stromme Danica J. Sutherland Alexander Burka Benjamin Lipton
Nicholas Felt Rebecca Roelofs Daniel-Elia Feist-Alexandrov Steve Dini Allen Welkie
Swarthmore College Computer Society
staff@sccs.swarthmore.edu

Abstract

As system administrators who are also full-time students, we aim to minimize the time we spend approving and carrying out standard tasks that comprise much of our day-to-day work. The less time required for these repetitive tasks, the more time we have available to provide new and exciting services to our community.

To facilitate the automation of this process, we have created the Grand Unified Task System (GUTS), which consists of a small core (a web interface and task executor) that unites task request processing for a range of modular services. This design allows for enhanced security and makes the system easily understandable and extensible, especially to new administrators. The Python backend provides deep integration with standard UNIX tools; the Django-based frontend provides a web interface friendly to both users and administrators. These design decisions have proven successful: deploying GUTS in production allowed us to dramatically reduce our response time for approving tasks, reach a much larger portion of our potential user base, and more easily support a diverse array of new services.

Keywords: infrastructure, project management, security, teaching system administration, web

1 Introduction

Since 1993, the Swarthmore College Computer Society (SCCS) has provided shell, mail, and web services to students, alumni, and other users affiliated with Swarthmore College. Because SCCS is run entirely by student volunteers on a small budget, both server resources and system administrator time are quite limited. To make efficient use of these resources, users receive access to most SCCS services only upon request. However, manually setting up user accounts, mailing lists, and databases on a case-by-case basis is both repetitive and excessively time-consuming. An ideal automated solution minimizes the amount of time that system administrators spend performing repetitive tasks while ensuring the security and supportability of the underlying systems. The Grand Unified Task System (GUTS) answers both of these questions through via its comprehensive web interface and deep integration with existing system tools. It replaces an older SCCS-designed system that performs a similar purpose, the Los [9] automated task request and approval system.

GUTS provides a full vertical solution and includes a submission and administration system for user requests, a website frontend for users, and a library of backend scripts that perform the actual tasks. An example of one such request is user creation, which involves a number of individual steps but is presented as a single task to the user. We have a policy of administrator approval for user creation tasks, but many other types of user requests are handled automatically and immediately by GUTS, without any interaction from the administrator (for example, password change requests). Having an administrator check over major requests helps both with security against malicious requests and with ensuring that various subjective policies that are hard to enforce automatically are followed. It is crucial that the system for handling requests makes review and approval of individual requests as easy as possible so that the administrator's time can be spent on reviewing the details of tasks rather than fighting with the system.

Users submit tasks to GUTS via a convenient web interface based on the Django [4] framework. Upon visiting the GUTS website, the user sees a listing of the available services in the form of tiles on the *SCCS Dashboard*, shown in Figure 1. After selecting the task, the user then encounters a form for entering data pertaining to the task. The data is type-checked and verified before further processing occurs. For tasks which do not require administrator approval, the user receives a visual report of the status of the task and often an accompanying email confirmation. If the task does require moderation, GUTS

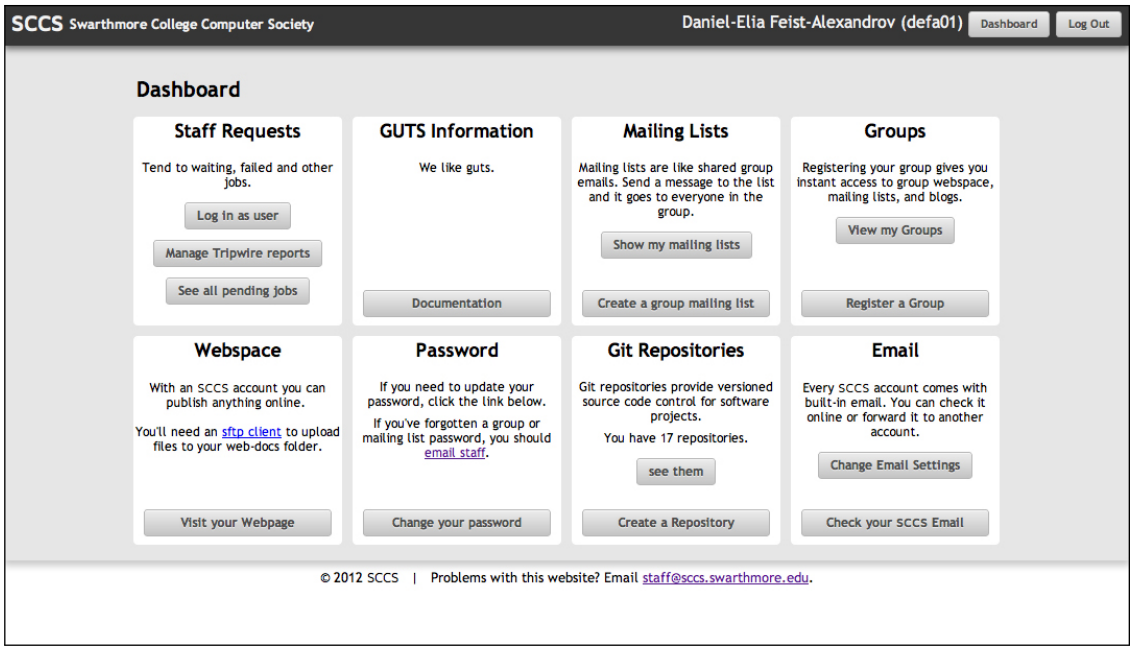

Figure 1: The SCCS Dashboard serves as the landing page for GUTS and provides users with an overview of the available tasks they may request. Each tile represents a self-contained pluggable GUTS service. Non-administrators do not see the *Staff Requests* or *GUTS Information* tiles.

sends an email to the administrators notifying them that there is a task awaiting approval. After the administrators review the task, the requesting user will receive an email detailing the status of their request.

At present, almost all of our system administration tasks follow this model. Through GUTS, users can request new accounts, change account passwords, set up email forwarding, and create and manage mailing lists, source code repositories, and groups.

This report first details how existing systems serving similar purposes do not satisfy our needs (Section 2). We then give an overview of the design of GUTS (Sections 3-5), and discuss its successes thus far (Section 6). We finally outline some of the lessons learned during the development process (Section 7) and identify some features of the design critical to its success (Section 8).

2 Related Work

Because managing large systems is traditionally labor-intensive, many different types of tools have been created to assist system administrators in performing their jobs.

One such type of tool, exemplified by Cfengine [2] and Puppet [11], automates the task of maintaining a large group of systems, keeping each machine as close as possible to a policy for the files and processes that should be present. This allows system administrators to avoid performing repetitive tasks such as software updates on multiple machines, and also allows the system state to be automatically repaired if an incident or user action causes it to deviate from policy. Though these tools are very useful for simplifying the administration and improving the robustness of a large installation of systems, they do not immediately address the problem of responding to user requests, which is the main cause for administrative action at small sites like SCCS.

Another category of tool is the ticketing system, used to log user requests and related actions by administrators. Some examples are Request v3 [8] and RT [1], both of which can automatically create tickets for email requests and manage tickets via email, command-line, or web interfaces. These tools keep track of the status of requests from users, so that the administrator is free to concentrate on satisfying these requests. Additionally, they allow reporting to track trends in requests and highlight problems. GUTS serves a similar purpose in our organization, but in addition to managing communication with users about certain types of requests, it also handles the execution of requested tasks.

A third category of tool simplifies administration by providing a graphical user interface for common administrative tasks, intended to make it easier for system administrators to configure their machines. One example of such a system is Webmin [3], which enables admin-

istration of many services on UNIX machines via a web interface. Webmin has a selection of modules that handle the administration of various services, each of which can configure that service on a selection of UNIX-like systems. The developers of Webmin have also created a tool called Usermin [12] that allows users to manage features of their own accounts via a graphical interface. GUTS combines these roles by providing a clean user interface for several features provided by the SCCS servers, both administrator- and user-initiated. In addition to providing a friendly interface to these features, GUTS also enables users to request administrator approval for actions for which they do not have sufficient privileges.

Finally, GUTS is the successor to Los [9], improving on the earlier SCCS effort in many respects. Both GUTS and Los support the automation of task approval and execution, but Los requires that administrators approve tasks through either a custom command-line tool or a GPG-signed email sent to the Los server. In contrast, GUTS provides a simple, easily-accessible web interface for one-click task approval. Since users interact with the same interface, effort spent on improving the frontend of GUTS benefits both users and administrators. GUTS is also much easier to extend than Los. While Los is written in a mix of languages and uses an inflexible, homegrown XML-based method to define tasks, GUTS code is pure Python, task definitions are simply functions, and task logic and presentation can be packaged together into modular services using the popular Django app model.

3 GUTS Core Components

At a high level, the core of GUTS combines a framework for automating system tasks with a convenient website that exposes these tasks to users. Built around this core are a number of services, each bundling together one or more tasks into a single modular package with a shared frontend interface.

The GUTS project core consists of the `libguts` library which contains the services framework and other helper code, the `gutsd` backend server which exposes registered services and functions, and frontend clients (most notably the `gutsweb` website) which use these registered functions to perform system actions. Figure 2 gives an overview of GUTS components, which the following subsections describe in detail.

3.1 GUTS Library

The `libguts` library provides a rich set of tools that allow programmers to easily write both GUTS system functions and the frontend interfaces that will call these functions. With `libguts`, the frontend programmer can call GUTS functions as if they were normal Python functions, even though there are a number of layers and transformations behind the scenes. In the following sections we describe first the semantics of GUTS functions, including how to register and call them, and second the structure of GUTS services, which organize these functions into modular packages.

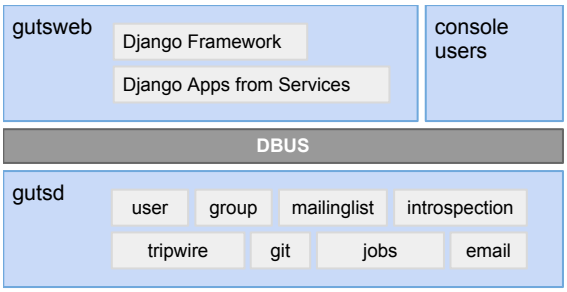

Figure 2: Architecture of GUTS components. Everything above the D-Bus line runs as unprivileged users while everything below runs as the root user. `gutsd` automatically loads and extracts the registered actions from each of the installed services, while `gutsweb` loads everything else contained in each service.

3.1.1 Functions

A GUTS function is a single task that can be performed on the system, such as creating a user, creating a group, removing a repository or modifying a mailing list. Functions are registered via one of the `libguts` Python decorators. The GUTS functions run as root within the `gutsd` process, so they can access any necessary parts of the GUTS job database or the running server as needed (for example, the `user` actions access the live LDAP database, and copy files around the server). All other code runs in the context of the calling program (in our case, `gutsweb` which runs as an unprivileged user). Because GUTS functions run with elevated privileges, we must ensure that there are no security holes and that parameters are checked for compliance before performing any system action. GUTS allows registered functions to provide type information and automatic parameter checking. The `libguts` library includes a number of checkers which verify commonly-used parameters. For example, one included checker ensures that a parameter is a valid UNIX group name.

A GUTS function is a normal Python function that has been registered with the GUTS daemon by a decorator from `libguts`. These functions fall into three types:

- *Info functions* have no side-effects on the system when executed. Examples include functions that list groups to which the user belongs or return metadata

for a source control repository.

- *Actions* are functions with side-effects on the system, such as a function that creates a user, removes a repository, or adds a user to a mailing list. Actions automatically add jobs to the database when run. Some actions are *delayed*, which means the parameters are added to the jobs database when run, but not executed until an administrator has approved the action.
- *Meta functions* perform some internal-to-GUTS task, such as listing the loaded services or running a delayed job. An augmented context is passed to meta functions, which includes the list of services and registered functions for the current GUTS instance.

3.1.2 Services

A GUTS installation is implemented and extended through the use of *services*. A service consists of a set of registered GUTS functions packaged inside a Django app that implements the logic and presentation of the service's web interface. An example of a GUTS service is the `git` module, described in depth in Section 5. In short, this service provides a collection of web pages for creating, editing and exploring Git repositories hosted on SCCS infrastructure. The Django-based website functions call out to GUTS functions to perform the actual heavy lifting of repository creation and modification. Services can interact with any existing system executables and libraries and are thus a powerful way to implement website and system functionality through GUTS. The SCCS GUTS installation includes services such as `git` (for Git repositories), `user` (account creation and management) and `mailinglists` (mailing list administration). The `git` service includes actions such as `create_repository`, `remove_repository`, and `modify_repository`.

We expect that services will often be written by programmers with little knowledge of internal GUTS systems. Because of this, we designed the services architecture to avoid boilerplate code as much as possible. A service leverages many of the elements of a standard Django app, including a settings file with service-specific settings, a url-routing module, view functions (that typically call GUTS functions) to display the relevant pages on the website, and template files for those views. Besides this, the service also contains the system-level code that implements the GUTS functions that the service provides. In this way, the service folder is a completely self-contained definition of everything related to that service (with the exception of any system executables or libraries it uses). The organization of this folder will be familiar to Django programmers. Figure 3 shows the elements of a service.

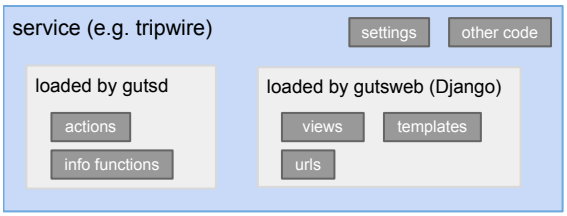

Figure 3: The anatomy of a GUTS service. A service is a Django app that also contains registered GUTS functions. Most of the service is loaded by `gutsweb` to power the website, but the actions are loaded by `gutsd` so that they can be run with superuser privileges.

Due to the self-contained nature of the service folder, GUTS services are modular and pluggable (though, if desired, they can have interdependencies where one GUTS services may use another's actions). When `gutsd` starts up, services are automatically discovered and sent to `gutsweb`. Several of the services that we have implemented are described in Section 5, as examples of functionality that can be plugged into GUTS.

3.2 GUTS D-Bus Daemon

The `gutsd` daemon provides access to GUTS services over D-Bus [5], an open source system for inter-process communication. The daemon runs with superuser permissions and registers itself on a known D-Bus interface. On startup it automatically detects services and loads their registered functions. It then presents these functions and overall GUTS introspection information over the D-Bus interface.

Security for `gutsd` rests on the use of a context object and the sender's username as provided by the D-Bus function `dbus_bus_get_unix_user()`. The context object is used to provide additional information to functions and parameter checking functions, such as the originator of the request (website or command line) and the identity of the user performing the request. On request submission, `gutsd` receives the context alongside the request parameters (both JSON-encoded) and uses it for validation. While `gutsd` gives the GUTS website's Unix user account trusted access and allows it to perform functions on behalf of any user, other users accessing GUTS from the command line are only allowed to make requests on behalf of their own user account, which `gutsd` enforces by checking the sender information from D-Bus.

While we have chosen D-Bus as our inter-process communication framework because of its wide deployment and security features, there is nothing integral to

GUTS that uniquely depends on D-Bus. Any other communications framework with the ability to securely identify message senders could be easily used instead. A framework without that ability could also be used by first adding an authentication layer on top of the framework. In our environment the D-Bus communication happens entirely internally to a single machine, but if messages between GUTS clients and the GUTS server are sent in an untrusted network environment, an additional layer of encryption (such as SSL) should be used to protect against snooping on function parameters (which can contain sensitive material such as passwords) as well as man-in-the-middle attacks.

3.3 GUTS User Interface

The main user interface to GUTS is the `gutsweb` frontend, a simple website with a design influenced by the modularity of the underlying service framework. Each GUTS service is allotted a tile on the dashboard (Figure 1) that describes the service and links the user to its own pages. These subpages can either provide further information or present forms for submitting a request.

For example, the mailing list service tile offers the choice of either viewing one's mailing list memberships or creating a new mailing list, as shown in Figure 4. Clicking the "Show my mailing lists" button leads to an overview of the user's mailing lists subscriptions, which including an indication of any administrative privileges for those lists. List administrators can simply click on the respective list to be taken to a new page where they can add or remove list members. Creating a new mailing list is just as easy; the corresponding button takes the user to a form for entering basic information about the new list and submitting the request for approval.

The allotment of one tile per service allows users to easily survey all the available tasks, while preserving the bundling of related tasks into clearly distinguishable services. It also presents a straightforward way to add the frontend hook for a new service; the developer can simply design a new tile and insert it into the dashboard, requiring minimum effort in working around the frontends for existing services.

3.3.1 Administrator Interface

When a user submits a request for an action that requires administrator approval, all staff members receive an email that contains the parameters of the requested action, a link to the new job on the administrative web interface, and a notification of the current status of the job. This is implemented by embedding in the email an HTML image tag pointing to a URL on the GUTS server, which will read "Executed", "Waiting for Admin", "Rejected" or similar, depending on the current job status. Because the image is downloaded when the message is viewed, administrators checking their email immediately see an up-to-date notification of the status. This notification prevents unnecessary trips to the web interface to approve already-executed jobs (a frustratingly common occurrence when SCCS used Los). Figure 5 shows a snapshot of the email, including the embedded image.

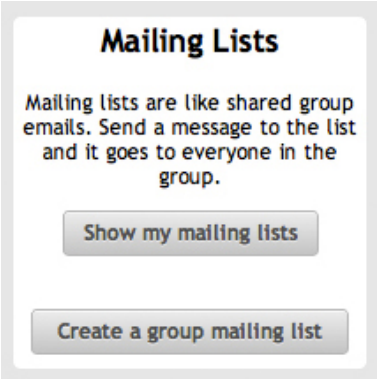

Figure 4: The mailing list tile. This tile is provided by the `mailinglists` service and provides access to commonly used tasks.

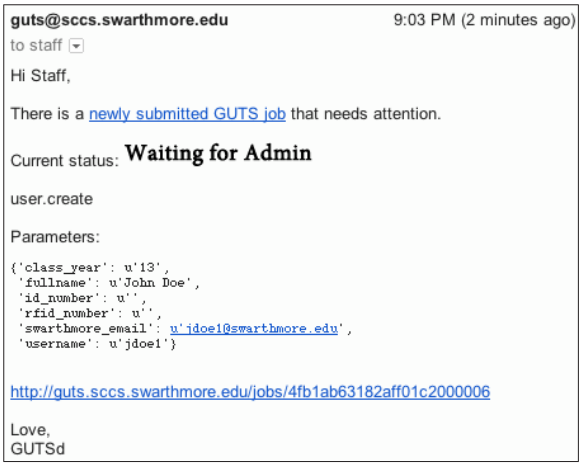

Figure 5: An example GUTS job notification email. Similar emails are sent out for all delayed actions so that administrators can approve them. The parameters dictionary has already been verified at this point, but it will be verified again once the action is approved.

Following the link in the email takes an administrator to the status page for the selected job. This page presents the history of the job and allows the administrator to approve or reject the job, or edit the parameters of the job before executing it. Each job is also accessible via the

"Pending Jobs" link in the "Staff Requests" dashboard tile, which leads to a page that can show not only pending requests but also a log of all previously executed jobs.

4 Security Model

From a security standpoint, it is important to ensure that user input is verified before it is used to make automated system-administration decisions. This provides protection against security vulnerabilities and prevents actions from running which would leave the server in an inconsistent state. This level of our security model is implemented with a layer of *parameter checker functions* that serve as a boundary between the interface-level code and the server-side functions. The code as implemented enforces our security guarantees while providing a convenient code interface for service programmers. It also serves as a common repository of validators for various data types, such as Unix users and mailing list names.

The input verification is all done on the server side. Similar to the Linux kernel, which provides a system call interface for userland to call into the kernel, the privileged code (in Linux's case, the kernel; in our case, the checker functions) needs to carefully check the validity of the parameters before doing anything. `gutsd` receives job requests over the Desktop-Bus (D-Bus) interface [5], and does not trust that input.

There are two kinds of verification which must be performed. The first is traditional input verification for web applications, in order to verify the input is correctly formed and to prevent database injection attacks. These are handled by individual parameter verifiers, which might check the format of a Unix username or verify that a flag coming from a dropdown list actually belongs to the set of legal options. After this basic verification, there are function-level checkers, which can reason about the relationships between parameters (for example, "Does this user have administrative permissions on this Git repository?") with full confidence that the parameters are valid for their individual data types.

To guarantee security, `gutsd` should check its inputs whenever they come from untrusted sources, and whenever the underlying system state might have changed. To this end, the verification typically happens multiple times during the lifetime of a call into GUTS. There are four ways to call into `gutsd`: (1) When an info function is invoked, the parameters are checked before the function is executed; these functions do not change system or database state. (2) When an action is invoked, the parameters are checked first, then the action is tried. If successful, the result is logged in the database, but if not a parameter error is returned to the caller (usually `gutsweb`). (3) A delayed action's parameters are checked, and if successful the server enters the request into the database and emails the staff. (4) Meta functions are like info functions, except that they are allowed to modify system state indirectly (e.g. by calling an action). Through these four interfaces, invalid data never enters the database, and data is verified both when originally submitted and immediately before being used.

Figure 6 presents a code sample from the `git` service (with some changes for brevity). This code demonstrates several parameter checkers and the beginning of a function which implements a GUTS action. This particular action can perform several functions related to the management of a Git repository (changing web visibility, descriptions, and user access). One of the parameters it must verify is a Unix user/group, so the action uses the `user-or-group` data type provided by `libguts` (logic within the function differentiates users and groups). The `permissions` and `visibility` arguments are verified to belong to a limited set of possibilities, while `name` verifies that a repository with the given name exists. Then, the action checker, `check_user_is_admin`, verifies that the user attempting to execute this action has the necessary privileges. It would be easy to create a `check_update_settings` function that called `check_user_is_admin` as well as performing any other required checks, if more checks were necessary.

The actual declaration of a GUTS action with specification of the verifiers is implemented using the Python function declaration syntax. A decorator provided by `libguts` interprets special default arguments of type `Param` to build up type information for each action parameter. This ties into one of our major design goals, which is to make implementing an action as simple as writing a Python function. The function `update_settings` in the code sample appears to be declaring a Python function with default values for its arguments. However, the decorators actually result in the creation of an object that automatically decodes its arguments from JSON, verifies them for correctness, and encodes return values, raising exceptions that GUTS understands if anything goes wrong along the way. All of this complexity is hidden from the service programmer.

Security Implications of `gutsweb`

Because `gutsweb` is a trusted client, a compromised `gutsweb` process could perform tasks on behalf of any user. However, the attack surface and potential damage are limited by the defined functions, and a compromised `gutsweb` process does not have root access. The frontend does not contain logic about user privileges; this is instead described and enforced in the validators for the functions themselves, preventing duplication of information and simplifying the implementation of `gutsweb`.

```
def check_user_is_admin(context, name):
    username = context.user.username
    repo = RepositoryGroup(settings.REPOSITORY_BASE_PATH).repository(name)
    if not (repo.user_is_admin(username) or
            any(repo.group_is_admin(g.groupname) for g in User(username).groups)):
        context.add_error("{0} is not an administrator of {1}".format(username, name))

def repo_name_check(context, name, params):
    repo = RepositoryGroup(settings.REPOSITORY_BASE_PATH).repository(name)
    if not repo.exists:
        context.add_error("Repository {0} doesn't exist".format(repo.name))

_permissions  = (None, "none", "read", "write", "admin")
_visibilities = (None, "public", "private")

@guts.action(service="git", checker=check_user_is_admin)
def update_settings(context,
                    name        = Param(checker=repo_name_check, required=True),
                    description = Param(                         default=None),
                    visibility  = Param(choices=_visibilities,   required=True),
                    entity_name = Param("user-or-group",         default=None),
                    permissions = Param(choices=_permissions,    default=None)):
   pass # GUTS action logic goes here
```

Figure 6: A code sample illustrating parameter checking for the `git.update_settings` action. A single GUTS action is defined which uses the GUTS Param domain specific language to define the function's parameters, complete with verification functions and defaults.

5 Case Studies for Implemented Services

User account creation User accounts were a core consideration when designing GUTS, because our accounts are non-trivial compared to those of many other sites. This consideration helped shape a number of GUTS features such as delayed actions, automatic parameter checkers and the split between `gutsd` and `gutsweb`. From the start we knew we needed GUTS to support delayed account creation, where the user signs up for an account and later an administrator approves it. Once the account is approved, GUTS needs to then create an entry in our LDAP system, copy skeleton files, and set up webspace, shell access and an email inbox. Doing all of this requires integration with a number of different tools (including LDAP, Postfix, shell commands, and Mailman) and therefore benefits hugely from the flexibility that GUTS functions allow. Writing functions in plain Python allows us to use pre-existing Python libraries as well as to call out to specific shell executables. The integrated delayed-action support ensures that checks are made both when the action is submitted and when it is finally run, in order to catch errors as early as possible, while also preventing errors that arise if the system configuration changes between submission and execution.

Git repository management In an effort to provide useful services to our users, SCCS previously created a set of tools for managing shared and private Git [10] repositories that are hosted on our server infrastructure. We make extensive use of Filesystem Access Control Lists (FACLs) to allow our users to create and use repositories, allowing private or shared read, write, or admin access as they wish. Users add, modify, and view repositories via our production GUTS website. This is implemented using a GUTS service which nicely integrates with our existing Git infrastructure which has a Python library interface. Info functions provide insight on existing repositories and action functions allow for changes to be made in a simple (from the service programmer's point of view) way.

Tripwire One of the tools we use to monitor the security of our servers is Open Source Tripwire [6], a file integrity scanner and reporting system. Tripwire is run daily and supplies the administrator with an overview of changed files in web directories and other high-risk areas. Originally the only way to approve or reject these

changes was through the shell. By writing a Python module that can parse and modify Tripwire's report files, however, we have now created a convenient web interface for approving these reports that integrates with the other approval systems we use every day.

Mailman One of the most popular services that we offer are GNU Mailman mailing lists [7], which are frequently used by many of the clubs and organizations at Swarthmore College. Since users are often not very comfortable or even aware of the Mailman administration interface, we decided to integrate that functionality into GUTS. Mailman provides a convenient Python interface to its list management systems, so this is fairly easy to accomplish. Now, users can comfortably create and administrate mailing lists directly from their dashboard, eliminating the need to leave the GUTS interface for common mailing list maintenance operations. For more advanced administrative tasks, the GUTS web interface provides a link to log directly into the Mailman web interface without having to use a Mailman administrative password: GUTS sets the login cookie used by Mailman before forwarding to the administrative page. We have also recently added to the Mailman interface an admin-side action which scans the mailing lists for invalid `@swarthmore.edu` email addresses, a constant problem for SCCS lists since graduating students rarely unsubscribe from their mailing lists before their email addresses expire. Although this could have been implemented as a standalone script, implementing it in GUTS meant that we already had convenient helpers in place to access our mailing lists and to check with the College's LDAP and SMTP servers for address validity.

6 Results and Reception

We made the decision to deploy GUTS in production and officially retire Los, the former task manager, after it became apparent that Los was not going to be able to keep up with the additional services that SCCS wanted to provide to its user base. The pivotal moment came when we wanted to automate the process of creating Git repositories for a user. This functionality would have been very hard to implement in Los, and thus it was decided to roll out GUTS somewhat earlier than planned. On October 10th, 2011, GUTS went live.

The results of the migration to GUTS were positive in all areas. Since GUTS made it possible to approve delayed tasks via the web dashboard, administrators could now tend to requests from any kind of web-enabled device, including smartphones. As a consequence, the median approval response time for delayed tasks dropped by a factor of 20 to 14.3 minutes for GUTS (mean 5.2 hours; based on 606 jobs between November 2011 and May 2012) as opposed to 4 hours and 54 minutes for Los (mean 59 hours; based on 1177 jobs between April 2004 and September 2011).

We also more than tripled the number of new user accounts per semester from the 2010-2011 academic year (using Los) to the 2011-2012 academic year (using GUTS). Although we have not attempted to measure the exact effect of GUTS, we believe a substantial part of this increase is from a higher conversion rate of students considering creating accounts, resulting from a greatly simplified signup process (1 page with 4 fields for GUTS; several pages for Los). We presently reach around one third of the incoming freshman class (our primary source of new users) every year, and expect this proportion to increase now that we can build on the infrastructure provided by GUTS to easily expand the services we offer and provide greater incentives for account creation.

But the greatest benefit of GUTS is that it acts as a unifying, straightforward platform for both SCCS and its user base: regular users are able to access most of the features an SCCS account provides with a single click from their GUTS dashboard, student group leaders can easily tend to their mailing lists and websites from the group dashboard, and administrators can quickly view, approve, correct, or reject tasks from the staff dashboard. Administrators can also access staff-specific areas via GUTS, such as the task for approving Tripwire reports, as well as built-in documentation on GUTS and a list of registered GUTS functions generated via introspection. This integrated approach has made life significantly easier for both the user and administrator, allowing the user to readily discover and take advantage of the services that SCCS offers, while lowering the hurdles for the administrator to develop, test, and roll out improvements.

7 Lessons Learned

As with all major projects, GUTS was a learning experience for those involved. This section offers a small window into our trials and tribulations in developing GUTS.

7.1 Mistaken Assumptions

There were several assumptions that we made during the development of GUTS that did not pan out once the service was launched. Some of these can be deduced from the types of email that we received right after launch. It became clear that the web interface needed to expose a lot of the functionality much more obviously. We still get many emails requesting password resets, even though this functionality has long been present in GUTS. We are working on improving the home page so that such simple functions are immediately apparent.

Also, as GUTS expands in scope, some new services that we want to integrate do not quite fit into the GUTS model. A case in point is the Tripwire module, which does quite a bit of processing on the server side in order to munge the report into a readable format and present it for staff approval in the web interface. If the report gets too long, this processing cannot complete before Apache decides that the process must have crashed and kills it. An expansion to our job model will allow us to mark certain jobs as asynchronous, so that they cannot lock up the main `gutsd` process.

7.2 Pipe Dreams

Of course, in the absence of time and resource constraints, there are certain aspects of the GUTS architecture that might be implemented differently. Many of these wishes have actually been resolved, as the internals of GUTS have been rather significantly overhauled since the system went into production. Making this work without disrupting the experience for our users requires discipline in testing all new changes before pushing them to the live system – as well as good error reporting (all staff members receive an email with a full stack trace whenever a user encounters a HTTP 500 server error). An automated test suite for GUTS would contribute greatly to reducing regressions, but would also require significant effort in mocking system interaction, due to the highly integrated nature of GUTS services.

Another facet of the system that might be overhauled if we had the expertise is authentication. We are quite satisfied with the security model that we have implemented, and all connections are secured with SSL so man-in-the-middle attacks are not a concern, but the only defense currently in place against impersonation is the Django authentication system.

7.3 Development Challenges

We had our fair share of hurdles to be overcome during the development process. The first of these was in design. The major reason for not writing GUTS earlier was the lack of a good security model. The core `gutsd` process has to run as root, in order to execute tasks (and even for some harmless activities, such as querying Mailman), but we certainly can't have the entire web server running as root. This common-sense security requirement is directly at odds with the purpose of GUTS, which is to allow users to execute tasks with sysadmin intervention only when necessary. In the past, Los solved this problem by restricting the user-facing parts. The web interface could only submit tasks, and the backend relied on the security of GPG-signed email for communication. For GUTS, we wanted tasks to be nearly full-fledged web apps, written in pure Python, which required devising a method for securely separating the user and server layers. The solution, discussed earlier in Figure 2, was to strictly partition privileged code behind a D-Bus interface. With D-Bus we can specify policies so that only the GUTS web interface's user can connect, for example, and only explicitly exported functions are accessible. Getting this to work for our site also involved setting up Apache MPM-ITK to run the GUTS web interface as a special user; since we allow users to write their own CGI scripts, this is how `gutsd` ensures it is talking to `gutsweb` and not another rogue Apache process.

Another significant driver of design decisions was ease of service writing. Ideally, a programmer familiar with Python and Django should be able to write a service without knowing about the internals of GUTS. Each service folder, as discussed previously, looks like a Django app. Of course, there is a fair amount of magic required to keep up this appearance. The `gutsweb` interface gathers parts from all of the services to make one Django app. On the other side, `gutsd` exports various functions from each service, based on decorators. We also take advantage of Python's default argument syntax to implement the parameter checking layer. Preventing these abstractions from breaking down was a major challenge in our development process.

Related to leaking abstractions, a design concern which came into play at the end of the process was the realization that GUTS could in principle be used at sites other than SCCS, and so there should be a clear separation between "core" functionality and anything SCCS-specific or essentially optional. Defining and enforcing this boundary was difficult and required reorganization of some of the GUTS internals.

Working through these challenges in the development process taught us a lot about the internals of Python, Django, and how to integrate these layers into a working system. The result, while the product of a somewhat non-linear development process, is a successful system that satisfies all of our key design goals.

8 Conclusion and Future Work

GUTS provides an integrated environment for user and administrator interaction with SCCS resources. Though its original goal was simply to speed up handling of administrative tasks so that a small group of volunteer system administrators could effectively serve the entire college community, GUTS is extensible enough that it has become the default platform SCCS uses for providing new user services of any kind. Several features of GUTS contribute to its applicability and ease of use.

First and foremost, GUTS includes a seamlessly integrated privileged execution model. Each service is sim-

ply a set of Python functions that the system calls as needed. The system also guarantees that the inputs to those functions are correctly formatted, and marshals interprocess communication over D-Bus between the unprivileged user interface and the privileged GUTS core. This makes it very easy to add new privileged services and design user-friendly interfaces to interact with them.

Secondly, GUTS streamlines the interaction between user and administrator, using an optionally-delayed execution model. Tasks that require administrator approval are performed via the web interface just like other tasks, but executed only after approval is received. Furthermore, the users and the administrators interact with the system through the same web interface. The GUTS job model is sufficiently general that administrative tasks such as approving delayed jobs are simply services that GUTS provides to the staff members. GUTS can be used as an interface to its own internals.

Lastly, our security model for services is designed around permissions that are afforded on a user or group basis. Parameter checkers can easily be implemented that enforce privilege separation for individual tasks. This means that the privileges required for an action are fully dictated by the service that implements it; there is no requirement that all services share a privilege model.

Although it is successfully serving many of our needs, GUTS is still an evolving system and has much room for improvement. The most immediate path for improvement is to add new services and extend those already supported. For example, we are currently working on implementing a service that allows users to create and manage MySQL databases, and one that lets users easily update their profiles on the SCCS-run campus photo directory.

To support more types of services, we would like to make it safe to create long-running jobs, which currently interfere with other jobs due to the single-threaded nature of `gutsd`. This can be solved by providing a facility for asynchronous jobs, which would return a response to D-Bus immediately, and then perform their work in the background. Managing these jobs should be relatively easy, since GUTS already has the ability to check on the status of jobs in the system.

A further way to improve the usefulness of GUTS will be to increase the flexibility of the user interfaces to the system. Some services, such as the `git` module, are generally used alongside command-line tools, and would therefore be convenient to access through dedicated command-line tools themselves — so we can avoid a trip to the web interface just to create a Git repository. We plan to implement a comprehensive command-line GUTS interface to fill this gap. Also, since many of the SCCS administrators approve jobs on their smartphones, a mobile-specific version of the web frontend would make that process even easier.

9 Acknowledgements

This work relies upon the invaluable contributions of several free software projects, most significantly Python, Django [4], and D-Bus [5]. SCCS is supported through equipment funding and space provided by the Student Council of Swarthmore College. We also receive network resources from the College's Information Technology Services Department.

10 Availability

GUTS is free software (released under the GNU General Public License), available from the project homepage at `http://sccs.swarthmore.edu/projects/guts/`.

References

[1] BEST PRACTICAL SOLUTIONS LLC. RT: Request tracker. http://bestpractical.com/rt/.

[2] BURGESS, M. A tiny overview of Cfengine: Convergent maintenance agent. In *Proceedings of the 1st international Workshop on Multi-Agent and Robotic Systems* (2005).

[3] CAMERON, J. Webmin: a web-based system administration tool for UNIX. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2000).

[4] DJANGO SOFTWARE FOUNDATION. Django: A web framework for the Python programming language. http://djangoproject.com.

[5] FREEDESKTOP.ORG. Software/dbus. http://dbus.freedesktop.org.

[6] KIM, G. H., AND SPAFFORD, E. H. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (New York, NY, USA, 1994), CCS, ACM, pp. 18–29.

[7] MANHEIMER, K., WARSAW, B., AND VIEGA, J. Mailman - an extensible mailing list manager using Python. In *Proceedings of the 7th International Python Conference* (1998).

[8] RHETT, J. Request v3: A modular, extensible task tracking tool. In *Proceedings of the Twelfth USENIX Large Installation System Administration Conference (LISA)* (1998).

[9] STEPLETON, T. Work-augmented laziness with the Los task request system. In *Proceedings of the Sixteenth USENIX Large Installation System Administration Conference (LISA)* (2002).

[10] TORVALDS, L., AND OTHERS. Git - fast version control system. http://git-scm.com, 2006.

[11] WALBERG, S. Automate system administration tasks with Puppet. *Linux J. 2008*, 176 (Dec. 2008).

[12] WEBMIN. What is Usermin? http://webmin.com/usermin.html.