AGUS: An Automatic Multi-Platform Account Generation System

Paul Riddle - Hughes STX Corporation
Paul Danckaert - University of Maryland, Baltimore County
Matt Metaferia - MCI Telecommunications

AGUS: An Automatic Multi-Platform Account Generation System

Paul Riddle – Hughes STX Corporation
Paul Danckaert – University of Maryland, Baltimore County
Matt Metaferia – MCI Telecommunications

ABSTRACT

As a computer network grows to accommodate thousands of users across many different types of hardware, account generation becomes an increasingly large burden on system administrators. Computer vendors often provide tools to create accounts; however, these tools tend to be specific to the vendor's hardware and OS, and most of them do not provide the degree of automation necessary to avoid a lot of repetitive work.

In a large network with a constantly changing user base, an automated account generation system would greatly ease the burden on the system administrators. Many such tools exist, but none of them provide support for many varied platforms such as UNIX, Novell, and VMS. We have attempted to address this issue by designing AGUS, an acronym for "Account Generation Utility System". AGUS is a distributed, networked, multi-platform account generation system which requires no administrator intervention during normal operation. It is currently in active use at our site, and handles account generation for several UNIX workstation and Novell-based PC/Mac networks, as well as a VAX running VMS. As of this writing, it has been used to create over 15,000 accounts.

Our Environment

The University of Maryland, Baltimore County (UMBC) is a medium sized university of about 12,000 students. Our computing resources consist of several hundred UNIX workstations (predominately Silicon Graphics and Sun), and around 2000 PC and Macintosh machines. These are spread out across about 25 subnets. Most systems are centrally administered by the University Computing Services (UCS) department; some, although few, are autonomously managed by the department or research group that owns them.

University Computing provides about 85 single-user Silicon Graphics (SGI) workstations, two multi-user SGI systems, 600 PC/Mac machines, and a Digital VAX 4000/500 for general use by anyone registered with the University. For faculty and research use, we provide an SGI Crimson, a MIPS R8000-based SGI Challenge "M", and a 20-processor SGI Challenge "XL" system. AGUS handles all account generation on these systems. Currently, about 7000 of our students have accounts on our UNIX network and on the VAX, and about 10,000 have Novell-based accounts which are required for access to our PCs and Macs.

Each person is assigned a unique username based on his or her real name. Accounts stay active until the user leaves the University, or we terminate it for other reasons (disciplinary, etc.). Account information is kept online using the CCSO Nameserver software from the University of Illinois[1]. User authentication on the UNIX systems is handled via Kerberos release IV[2], from MIT Project Athena[3].

How we used to do things

Up until the fall semester 1993, we assigned student accounts on a course by course basis. Instructors would request accounts for their classes, and we generated a set of accounts for each course. Accounts were generated and distributed to instructors several days before classes began. Usernames were formed by taking the course name and appending ascending numbers to the end. Instructors handed the accounts out in class, and the accounts were deleted after final grades were due at the end of the semester.

This method had several disadvantages, the main one being that accounts were difficult to trace. Some instructors kept logs of which students had which accounts, but most didn't. If we got a complaint about an account being abused, it was very difficult if not impossible to trace the account back to a particular person.

These accounts also required a lot of work at the beginning of each semester. We had to solicit account requests from instructors, generate the accounts, print account information, and distribute accounts back to instructors. Invariably, some instructors would put in a late request for accounts, requiring us to repeat the process. We ended up spending way too much time generating accounts, when there were much more important things that needed to be done to prepare for the semester.

Class accounts were also unpopular with students. Many students received multiple accounts because they were enrolled in more than one class. The account names were difficult to remember because they were not based on the student's name. Students were unable to keep the same email address from semester to semester, and were forced to download files they wanted to keep to floppy at the end of each semester.

Our Account Generation System Requirements

What we wanted to provide to users

We wanted to enable any University-affiliated person to register for and receive an account on our Unix, VMS, and/or Novell networks. The registration process should be as quick and painless as possible, and unless absolutely necessary, it should not require the user to interact with any third party (i.e. instructors or University Computing staff). The accounts should stay active for the user's entire stay at the University. Account names should be easy to remember as well; preferably they should be based on the user's real name.

Obviously, temporary class accounts didn't fit our vision of what we wanted to offer to users; so, we decided to stop using them after the Spring Semester 1993. During that summer we began to consider other schemes for generating accounts.

Other Requirements and Desired Features

The first step in designing an account generation system was to identify the most important features it should have. We decided that in order to be useful to us, our system must have the following features:

Ease of Administration

It is essential that our system require as little administrator intervention as possible. During the first week of each semester, we will receive as many as 1000 requests for new accounts, and a steady stream of requests over the course of the semester. Manually processing these requests would be extremely tedious, repetitive and error-prone. Therefore, we designed AGUS to run completely unattended. During normal operation, the system runs itself and the administrator doesn't need to do anything at all.

Scalability

The system should be able to scale to accommodate a network of any size. As a medium sized University, we have an average user base of around 9000 active users spread out over several administrative networks. We designed AGUS to be capable of dealing with networks many times larger than ours.

Flexibility

One of our most important requirements, and a feature that sets AGUS apart from many similar systems, is complete platform independence. The system should be able to deal with any type of computer system that is capable of being networked via IP. We wanted to use the same system to create accounts on UNIX, VMS, and Novell based networks. The system should also be designed in such a way that it is simple to add additional system types to the configuration. For example, if the University decides to support user accounts on HP MPE systems, it should be relatively easy to extend AGUS to handle account creation under MPE.

Robustness

An account generation system should provide a good degree of robustness and error recovery. If it encounters a fatal error, it should notify the administrator. It should never leave an account in a partially-generated state.

Things We Considered and Tried

Before designing AGUS, we considered several other methods for generating accounts. Each has its own merits, but fails to satisfy one or more of our requirements.

Vendor Tools

The simplest way to create accounts is to use whatever tool the vendor provides. Silicon Graphics, for instance, provides a very nice GUI-based tool for account generation on Irix systems[4]. Unfortunately, this tool must be run manually for each new user to be added. In a typical fall semester at UMBC, over 2000 freshmen will request computer accounts. Using a GUI-based tool to individually add 2000 accounts would be hopelessly tedious.

Locally Developed Account Generation Scripts

It is fairly easy to develop a shell or Perl[5] script which handles account generation. An advantage of this approach is that the script can be tailored to do site-specific things. Most vendor tools do not provide this kind of flexibility. However, this approach was not automated or flexible enough to fit our needs. We wanted something that wouldn't require the staff to do repetitive work, and we wanted a system that ran on many different computer platforms.

Moira

Moira[6] is used at MIT to handle account creation on their Project Athena systems. We looked at this and found that, although it would do some of what we needed, it does not extend well to non-UNIX platforms, and it requires the presence of other Project Athena packages for proper operation. It also requires a commercial database package, which we did not want to use; one of our goals was to develop our system completely around public domain tools, so it would be of use to as many people as possible. Moira was also somewhat of an overkill for us. It was designed as a general purpose *service management system*, and account generation is only a subset of its overall functionality.

AGUS

In designing AGUS, our initial goal was to create a system which supported automatic account generation and ran on every platform we supported. After the basic framework was in place, we refined the system to make it more robust and flexible. Now that it is stable, we are working to make it sufficiently generic that it will be useful to other sites with little or no modification.

AGUS Development History

AGUS, Take 1

AGUS was originally designed and implemented by a group of students as a class project for a Software Engineering course[7]. This implementation, which I'll call AGUS-1, was well thought-out and well-documented, and served our purpose for two semesters. Unfortunately, it had several critical design flaws which required us to eventually redesign it from the ground up.

AGUS-1 was implemented primarily using Bourne shell[8] scripts, and all processing was done on one of the multiuser student login machines, which was often overloaded. The system did not scale well at all. The machine would slow to a crawl when more than about 10 people tried simultaneously to register for accounts; no one would be able to get any work done. We ended up placing limits on how many people could register at one time, which greatly inconvenienced students, especially at the beginning of semesters.

After the first week of using AGUS-1, it became apparent that the system wasn't working too well; so, we decided to redesign it. Our goal was to provide the same menu-based interface to users, but eliminate the problems with the underlying implementation.

The New AGUS model

In redesigning AGUS, we worked to create a system with no major bottlenecks. A major goal was quick response time to users during the online registration process, even when several people are registering at the same time. We accomplished this by splitting AGUS into separate modules.

We initially coded most of AGUS in Perl version 4, with small parts of it in C. Developing and coding things in Perl is fast and straightforward, so we were able to get the initial system online quickly. However, Perl is difficult to extend, and we were forced to add extra modules to interface to KerberosIV and the CCSO database. It's also somewhat awkward to deal with complex data structures in Perl. Because of this we began to port the entire system to C. AGUS in its current state is about 90% C and 10% Perl.

AGUS From the User Perspective

AGUS presents users with a menu-based online registration system. A user walks up to an unused workstation or `telnets` to one of our multiuser login servers, and enters `register` at the username prompt. The user is then prompted for an identification (ID) number.[1] If the supplied ID number is valid, the system then displays a list of accounts for which the user is eligible to register.

Once the user has selected one or more accounts, the system generates a printout which shows the username and password for each account, as well as a form which explains our acceptable usage policy. The user presents valid identification to the print dispatch operator and signs the form. After 12 hours, the accounts are ready for use. Normally, this process requires no intervention by system staff.

What AGUS Doesn't Do

AGUS does not generate usernames or assign user identifiers. It requires that this information be present in the external user database. We did things this way for a couple of reasons; first, we didn't want to impose our naming scheme on other sites that want to use AGUS. Also, usernames and user identifiers are system-specific entities. For example, Digital's VMS uses two numbers to identify each user. These numbers are known collectively as the user's UIC (User Identification Code). UNIX uses a single number which it designates as the UID (User IDentifier). If AGUS were to generate user identifiers internally, it would have to know the rules for doing this on each supported system type, and it would have to be modified each time someone added a new system type. Since this goes against our system-generic design philosophy, we left it out of AGUS. Usernames and user identifiers are generated ahead of time, at the same time we load new student information into our database. One consequence of this is that we must assign usernames and identifiers for every registered student, not just the subset of students that will be using our computer systems. We have not found this to be a problem.

AGUS Requirements

We tried to design AGUS to require very few system resources and as little external software as possible. The existing implementation has minimal hardware requirements and requires an external database package capable of performing simple queries and updates.

[1]We currently use the student's Social Security Number for identification; however, the University has plans to begin issuing everyone a unique Personal Identification Number (PIN). If this ever happens, we'll probably start using the student's PIN rather than Social Security Number.

System and Hardware Requirements

The AGUS system requires a UNIX based system to run a single daemon process and a queueing system. The daemon accepts and queues new account requests, and the queue processor runs at periodic intervals and creates accounts. The daemon is *lightweight;* that is, it has negligible memory and CPU requirements. It can run standalone or via *inetd*[9]. The queueing system uses very little memory and requires enough disk space to hold the new account request queue. There is one queue entry per pending account request; each entry is stored in a separate file. Queue files are about 50 bytes long.

At UMBC we run the AGUS daemon and queue on a DEC 5000 with 96meg of memory. This system handles several other services such as e-mail relaying, backup DNS, and system backups. The AGUS system does not noticeably affect performance of the DEC.

AGUS also places a small load on the systems on which it is creating accounts. This involves things like modifying password files, creating home directories, etc., and is to be expected. In some cases (see "AGUS Implementation Details"), an additional daemon process handles account creation.

The External Database

AGUS requires read and write access to an external database package to keep track of users. The database must be able to do searches using a user's ID number as the key. For any given user, the database must be able to return four different fields: the user's real name; a unique username and identifier for each type of account the user is registering for; and a *group* which specifies which accounts the user is eligible to receive. The database must also keep tabs on accounts the user already has, so that people cannot register for the same account more than once. A sample database entry is shown in Figure 1.

```
id:             999999999
name:           bernie freeman
unix_username:  bfree
unix_uid:       451
vms_username:   bfreeman
vms_uic:        451.100
group:          research
inst_created:   1994/04/05
```

Figure 1: Sample database entry

We use the CCSO nameserver as our external database. We chose this package because it was free and satisfied other requirements we had (unrelated to AGUS). AGUS was written so that it should be easy to adapt it to use any database system that includes a programmatic interface. It includes a small database system (based on NDBM[10]) for people who wish to try AGUS out without installing CCSO or adapting AGUS to use a different package.

AGUS Naming Requirements

AGUS requires that each computer network it handles be given a unique name. The system uses these names internally and when prompting users for account types. At our site, our general-use SGI UNIX network is designated *instructional*; our faculty and research UNIX network is called

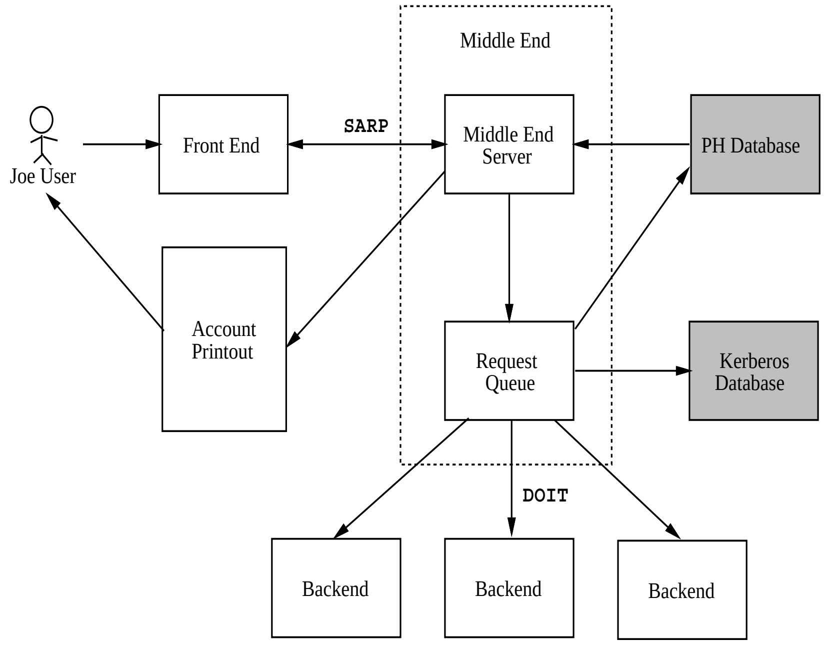

Figure 2: Overall structure of the AGUS system

research; and our VAX system goes by *vms*. Think of this name as being similar to a NIS[11] domain name; it refers collectively to all of the computers in a single administrative domain.[2] You should choose names which are descriptive enough that a new user will understand which computers each name refers to.

For each computer network, AGUS uses three database fields for a given user. The first specifies the user's username on the network. The second field is a unique identifier for the user; under UNIX this is the user's UID, and under VMS it is the UIC. Under Novell, identifiers are managed internally so the field is not used. The third field specifies the account creation date, and is set at the time AGUS generates the user's account. For example, referring back to Figure 1, Bernie Freeman's UNIX username and UID are determined by his `unix_username` and `unix_uid` database entries. The `inst_created` field shows the date that Bernie registered for an account on our instructional UNIX network. There is a similar field, `res_created`, for our research network. Bernie doesn't have a `res_created` field, which means he has never registered for a research account.

AGUS determines which database fields to use by consulting a configuration file, `agus.conf`. This file is presented in the next section.

AGUS Implementation Details

The overall structure of the AGUS system is shown in Figure 2. System-specific, time-consuming tasks such as creating home directories and assigning disk quotas are wrapped into separate *back end* modules. When a user logs in as `register` to request a new account, he interacts with a program we call the *front end.* The front end prompts the user for an ID number and passes it on to the next layer of AGUS, the *middle end.* The middle end validates the user's ID number and determines what accounts the user is eligible to receive, if any, based on the user's group field in the external database. The user chooses one or more account types. The middle end then generates initial passwords, creates an information printout for the user to pick up, and places the registration request into a queue. At periodic intervals (every 6 hours at our site), the system processes the request queue and creates each account by sending commands to the appropriate back end module for that account type. It also makes updates to the external database and, when necessary, the Kerberos database.

AGUS learns about each computer network by reading a configuration file called `agus.conf`. This file contains one line for each computer network. Lines consist of several colon-separated fields. Figure 3 shows a sample `agus.conf` file. Lines which are too long for the page are split with a backslash ('\').

```
instructional:ds1.gl.umbc.edu:\
  umbc8 and umbc9:gl.umbc.edu:\
  inst_created:kerberos:\
  unix_username,unix_uid,group,name
research:umbc7.umbc.edu:umbc7:\
  research.umbc.edu:\
  res_created:kerberos:\
  unix_username,unix_uid,group,name
vms:/umbc/agus/bin/vms-rcp:umbc2:\
  umbc2.umbc.edu:vms_created:plain:\
  vms_username,vms_uic,group,name
```

Figure 3: Sample `agus.conf` file

The first field in each line is the name of the network. In this example we have entries for three different networks: *instructional*, *research*, and *vms*.

The second field contains information about the AGUS backend for this network. It specifies either a hostname or a pathname. See the "Backends" section for more on this field.

The third and fourth fields contain information to display on the printout that the user picks up after registering. The third field is the hostname of a computer (or computers) which the user can access (via `telnet` or `rlogin`) for remote access to his account. The fourth field is the electronic mail address of the network, and is used to tell the user his email address.

When AGUS creates an account, it stores the creation date of the account into AGUS's external database. The fifth field of the `agus.conf` file specifies which database field to use to store this information. For example, when creating a research account, AGUS should store the account creation date into the `res_created` field of the new user's database entry. For instructional accounts, we use `inst_created`, and VMS accounts use `vms_created`.

The sixth field contains information about how passwords are handled on this network. We currently support three different methods: *kerberos*, for passwords stored in an external Kerberos database; *crypt*, for standard one-way UNIX password encryption[12]; and *plain*, for plaintext passwords (required in some non-UNIX environments).

The seventh and final field consists of several comma-separated words. Each word is the name of a field in AGUS's external database. These fields are used to create an entry for a new account on this network in AGUS's account request queue. See "The AGUS Request Queue" for further information about the queue.

[2]Note that for security reasons, you should choose a name that is different from the NIS domain name, if you run NIS at all.

AGUS uses a configuration file called `group.conf` to determine for which accounts a user is eligible to register. A sample `group.conf` file is shown in Figure 4.

```
staff:instructional,research,vms
systems:instructional,research,vms
research:instructional,research,vms
general:instructional,vms
guest:instructional
inactive:
```

Figure 4: Sample `group.conf` file

The `group.conf` file contains one line per group. Each line consists of a group name, a colon separator, and a list of account types that users in the group can register for. For example, someone in the *general* group can register for an instructional account or a VMS account. If a user is unlucky enough to be in the *inactive* group, he cannot register for any accounts at all. Looking back at Figure 1, we see that Bernie's group is *research*; this entitles him to an instructional, a VMS, or a research account.

Interaction Between the Front End and Middle End

The front and middle end modules communicate over TCP using a simple 7-bit ASCII protocol similar to SMTP[13]. We call this protocol SARP (Simple AGUS Registration Protocol). SARP provides commands to query the database for a specified ID number, select one or more account types, and commit registration requests to the registration queue. The full protocol specification is provided in Appendix A. The advantage of this approach is that we can develop front end clients which run on a wide variety of hardware and support several different user interfaces. Our current client uses the UNIX curses library, and we have started work on clients based on Motif[14], TCL/Tk[15], and World Wide Web CGI[16] scripts. We are also developing an alternative UDP-based server/client protocol, which would be more appropriate for stateless clients such as WWW browsers.

The AGUS Request Queue

AGUS's account request queue is similar to the mail queue in sendmail[17]. We considered creating accounts at the same time users registered for them, but quickly abandoned this idea because it would create an undesirable load on the backend server hosts when multiple people registered simultaneously. Our solution was to place account requests into a queue, and then process them one-by-one at specific times via *cron*[18].

The queue consists of one directory for each type of account. At our site there are directories called `instructional`, `research`, `vms`, and `novell`. These directories contain one file per pending account request. Files are named after the registrant's ID number. As an example, let's say that Bernie Freeman from Figure 1 decides to register for a research account. Assuming the AGUS queue is rooted at `/agusq`, the pathname of Bernie's queue file would be `/agusq/research/999999999`.

Queue files are fairly simple. They contain all the information the backend needs to create the account, with each piece of information on a separate line. The `agus.conf` file (see Figure 3) specifies the actual information that goes into the queue file. The first line of the queue file is always the user's password, encoded according to the information in the sixth field of the network's `agus.conf` entry. For Kerberos based accounts, the password is set to '*' and a separate routine updates the Kerberos database. For most other UNIX accounts, the password is stored encrypted in the queue file; for most non-UNIX accounts it is stored in plain text.[3] On UNIX systems, the backend stores the password in `/etc/passwd` exactly as it appears in the queue file.

The rest of the queue file lines consist of information from the external database. The seventh field of `agus.conf` contains a list of database fields; AGUS looks up each of these fields and writes its value to the queue file, one entry per line.

To illustrate, let's again assume that Bernie Freeman wants to register for a research account. According to `agus.conf`, the research network uses Kerberos to store passwords, and the database fields to use for the queue file are `unix_username`, `unix_uid`, `group`, and `name`. Based on this information, Bernie's queue file will look something like this:

```
*
bfree
451
research
bernie freeman
```

A cron job runs at specific intervals (every six hours at our site) and scans the queue directories for requests. For each request it finds, it sends the request to the appropriate backend server. The backend server returns a result code which says whether the account creation was successful. If so, the request is removed from the queue; if not, it is retried the next time the queue is processed via *cron*. Requests that fail more than once are reported to the system administrator.

[3]Cleartext passwords are a security risk on public networks and should be avoided when possible; we are working on a way to handle these securely which is easily portable to non-UNIX platforms.

Backends

Backend modules are responsible for actually creating new accounts. For example, on a UNIX system, the backend program would be responsible for creating password file entries, assigning and creating home directories, setting permissions, assigning quotas, and setting up initial startup files (`.cshrc`, `.login`, etc.). For these types of systems, the backend program runs on the system where this type of work is typically done, e.g. the NIS master server for networks that run NIS. The middle end communicates with backends using one of two different methods: the DOIT network protocol or an external backend program. AGUS determines which of these methods to use by looking at the second field of `agus.conf`. If the field starts with '/', AGUS assumes that it specifies a pathname to an external program and attempts to execute the program. If the field does not start with '/', AGUS treats it as a hostname and attempts to connect to the host using the DOIT protocol.

The DOIT Protocol

DOIT is a 7-bit ASCII protocol similar to SARP. Our UNIX-based backend program speaks the DOIT protocol. The protocol specification is provided in Appendix B. DOIT provides a command to create an account given several different colon-separated parameters (typically username, encrypted password, UID, group name, and full name). AGUS obtains these parameters by reading the queue file and massaging the contents into the proper format. A client can generate as many accounts as it wishes during one session.

External Programs

Designing a backend that speaks an ASCII protocol over the network typically requires TCP/IP connectivity and some sort of BSD-socket-like programming interface. We realized that it if this were a requirement, it would be difficult to develop backends for non-UNIX platforms. For these types of systems, AGUS can invoke an external program to communicate with the backend. This is similar to the behavior of sendmail's *prog* mailer. For example, to generate accounts on our VMS system we invoke an external program which uses *rcp*[19] to copy the AGUS queue file over to a special account on the VAX. Every several hours, the VAX runs a script that checks for new files in that account, and creates accounts based on what it finds. We use a similar program for our Novell systems.

Examples

A few examples should help illustrate how everything fits together. For these examples, let's assume that our old friend Bernie Freeman wants to register for an research UNIX account and a VMS account. He would first walk up to an unused workstation and log in as `register`. The login shell for the *register* account is an AGUS front end client, which prompts him for an ID number. The system then connects to the AGUS middle end server, which determines whether the ID number is valid. If so, the middle end returns a list of account types Bernie is eligible to receive. Since Bernie is in the *research* group, he is automatically eligible for an instructional UNIX account, a VMS account, and a research UNIX account. Since he already has an instructional account, though, he can only register for research or VMS. The front end formats this response and presents it to Bernie as a menu. Bernie is interested in a research and a VMS account, so he selects these from the menu. The system sends this information to the middle end server. The middle end generates random passwords for each account, creates two queue files, and generates a printout of account information for Bernie to pick up.

The SARP transaction between the front and middle ends for our example is shown in Figure 5. Input from the front end is shown in **`bold`** typeface and responses from the middle end are shown in `plain` typeface. Refer to Appendix A for a description of the individual SARP commands.

```
220 Server ready.
IDNO 999999999
210-bernie freeman
210-research
210 vms
TYPE research
200 TYPE research OK
TYPE vms
200 TYPE vms OK
RGST
250 Request queued
QUIT
221 Goodbye!
```

Figure 5: Sample SARP Transaction

```
220 Server ready.
DOIT bfree:*:451:research:bernie freeman
200 Okay
DONE
221 Goodbye!
```

Figure 6: Sample DOIT Transaction

The system creates Bernie's accounts the next time it processes the AGUS request queue. For the UNIX account, it establishes a connection to the backend server running on the NIS master server of our research UNIX network (specified in the second field of `agus.conf`), and creates the account using the DOIT protocol. Figure 6 shows the DOIT transaction. For the VMS system, the system executes a Perl script called `vms-rcp`. The script copies the queue file over to the VAX, where the account is created via a cron script. Note that AGUS does not assign home directories; this is handled internally by the backend program.

Conclusion

Although there were some growing pains, AGUS has proven to be a good solution for automatic account generation at our site. Users like the consistent interface it provides, and the system staff appreciate its completely automatic operation. In addition, it is the only system we know of that can be extended to many different OS platforms, both UNIX and non-UNIX. By making it freely available we hope it can be of use to others as well.

Availability

The AGUS system has been in production at UMBC for almost two years now. It is not currently release ready, but we are actively working on preparing it for public beta testing. We hope to have something to offer within the next several months.

Author Information

Paul Riddle is a Systems Programmer for Hughes STX Corporation, working as a contractor at NASA/Goddard Space Flight Center in Greenbelt, MD. Paul wrote the AGUS registration server and queue processor. Paul did most of his work on AGUS while an employee of the University of Maryland, Baltimore County, and now maintains the code in his spare time. This is his second LISA Paper.

Paul Danckaert is a Systems Programmer for University of Maryland, Baltimore County. Paul wrote the AGUS front end interface, and helped design the basic AGUS model and protocol specification. His main area of research is in computer security, and his most recent project enhanced IRIX modules for Rscan, a security scanner by Nate Sammons from Colorado State. This is his first LISA paper.

Matt Metaferia currently works at MCI Telecommunications in Washington, D.C. Matt designed and coded the UNIX based backend server and account generation engine. Matt did all of his work on AGUS while employed at the University of Maryland, Baltimore County.

Appendix A – Simple AGUS Registration Protocol (SARP) Specification

Overview

SARP is a network protocol similar to SMTP and FTP[20]. It runs over TCP and provides a simple mechanism for users to register over the network. Any client that can speak SARP can be used for registration.

SARP runs as a TCP service on the well-known port specified by the *agus* entry in the `/etc/services` file. Currently it runs from port 293. It should always run from a privileged port (port number less than 1024) so that unprivileged users can't create a "phony" SARP service in the event that the real one dies.

Connection Negotiation

When the SARP server receives a new connection, it should issue a greeting banner and begin speaking the SARP protocol. All server responses must be preceded by a 3-digit numeric result code. The greeting banner should include a `220` result code, indicating that the server is ready to register a new user.

Server Responses

Each response consists of a 3-digit number followed by a text explanation of the response. Example:

```
200 Command succeeded
```

A hyphen immediately following a response number indicates a multi-line response. The last line in the response will not have a hyphen. A sample multi-line response follows:

```
210-bernie freeman
210-instructional
210 vms
```

Numeric responses each have certain meanings. Clients can parse these responses to determine whether commands succeeded or failed. The meanings are similar to what they would be in SMTP or FTP:

`2xy`	Indicates general permanent success.
`4xy`	Indicates general transient failure.
`5xy`	Indicates general permanent failure.

`x0y`	A syntax related message.
`x1y`	An informational message.
`x2y`	A message related to the connection.
`x3y`	An authentication related message.
`x4y`	Unspecified.
`x5y`	A message related to the registration server.

Here are the actual response codes used by the SARP and DOIT protocols, in numeric order.

- `200` – Command succeeded.
- `210` – Successful informational response.
- `220` – Server ready for new session.
- `221` – Closing control connection.
- `401` – Transient error in command parameters.
- `450` – General transient failure.
- `500` – Unrecognized command.
- `501` – Error in command parameters.
- `503` – Improper order of commands.
- `504` – Command not implemented for this parameter.
- `520` – Closing control connection due to fatal error.
- `530` – ID number not found in database.
- `550` – Requested action not taken (failed).

Server Commands

Commands are four letter words or abbreviations. All input is case insensitive; `IDNO` is the same as `idno`, `iDnO`, etc.

Here is the canonical list of SARP commands and what they do.

IDNO – Specify an ID Number for Registration

Syntax: `IDNO` *id-number*

This specifies that a person with ID number *id-number* wishes to register for an account. The server should look up the ID number in its database. The ID number must be a 9-digit number; if it's not, the server should return a `501` response (parameter error).

If the ID number is valid and present in the database, the server should return a multi-line `210` response. The first line prints the registrant's name, and each succeeding line lists a valid account type for the registrant.

A valid account type is one that the user (1) is allowed to register for, and (2) has not yet registered for. If the user is valid but has already registered for all allowed accounts, the server should still return a `210` response, but should not list any account types with the response.

An `IDNO` command may not be specified more than once. Duplicate `IDNO` commands should return `503` (sequence error) responses.

Examples

Sample `IDNO` responses follow:

```
210-david duffy
210-instructional
210 research
```

This user is in the database and is allowed to register for either an instructional or a research account. The user is either not allowed to register for a VMS account, or has already registered for one.

```
530 IDNO 123456789 not present\
  in database.
```

This user is not in the database.

TYPE – Specify an account type

Syntax: `TYPE` *account-type*

This command selects an account type to register for. It must follow an `IDNO` request; if it doesn't, it should return a `503` response. It requires one argument; if no arguments are specified, it should return a `501` error.

The argument to `TYPE` must be one of the responses generated by the `IDNO` command. Invalid responses should return `504` (command not implemented for parameter) responses.

A user may register for a particular type of account only once in a session. Multiple attempts to register for a single account type, should result in `503` responses.

Assuming the requested account type is valid, the `TYPE` command should return a `200` response.

Examples

Sample `TYPE` responses follow:

```
200 TYPE instructional OK
```

The user wishes to register for an instructional account, and is authorized to do so.

```
504 Invalid account type
```

The user tried to register for an account type not returned by the `IDNO` command.

RGST – Register for Account(s)

Syntax: `RGST`

The `RGST` command actually registers the user after he or she has specified an `IDNO` and at least one `TYPE`. If the user attempts to `RGST` before doing either of these things, the server should return a `503` response.

If the registration succeeds, the server returns a `250` response; otherwise it returns a `550` response with the reason included in the text of the response.

`RGST` should do an implicit `PRNT` operation when it succeeds.

PRNT – Print Account Registration Information

Syntax: `PRNT`

This command prints registration information for the user. The purpose of this command is to reprint the information in case it didn't print the first time. `RGST` should handle printing for new registrants.

`PRNT` must be specified after the user enters an `IDNO`, but before they enter a `TYPE`. Any other usage should return a `503` result code.

If the print job succeeds, `PRNT` should return a `250` response.

QUIT – Exit From the Registration Server

Syntax: `QUIT`

`QUIT` is used to disconnect from the server. All pending action is cancelled with the exception of queued account requests. `QUIT` should return a `221` response code before dropping the connection.

Appendix B – DOIT Backend Registration Protocol Specification

DOIT is a network protocol similar which works similarly to SARP. DOIT runs as a TCP service on the well-known port specified by the *agus-doit* entry in the `/etc/services` file. Currently it runs from port 294.

DOIT only needs to be able to handle a single connection at a time. Clients should take pains to ensure that they only have one active connection active at any given time. This restriction allows the backend AGUS software to avoid dealing with mutual exclusion, file locking, and race conditions.

Connection Negotiation

When the DOIT server receives a new connection, it should issue a greeting banner and begin speaking the DOIT protocol. The greeting banner should include a `220` result code, indicating that the server is ready to accept requests.

Server Responses

DOIT server responses follow the same conventions as SARP responses, including a 3-digit numeric response code and a text message. For more details, please refer to Appendix A.

Server Commands

The DOIT protocol supports only two commands: `DOIT` and `DONE`.

DOIT – Specify Account Registration Data

Syntax: `DOIT` *username:password:uid:group:gecos:quota*

This specifies data for an account to be created. The DOIT protocol requires that six fields be specified; however it does not do syntactical checks on the individual fields. That is left up to the backend. The backend is responsible for generating all information (home directories, shells, etc) not specified via `DOIT`.

Once the backend receives a `DOIT` command, it should create the account on the fly and return a response code depending on whether or not it succeeded. A `200` response indicates that the request succeeded, and that the client may assume that the account has been successfully created. The client need not queue or retry the request after a `200` response.

A `450` response indicates a transient failure. This means that the backend failed for the current request, but the client may continue to send further requests during the current session.

Errors in the parameters should return `401` responses. These include things like improperly formatted usernames, invalid UIDs, nonexistent groups, etc. The backend should also notify the AGUS administrator of these errors, since they generally indicate some sort of internal problem.

Fatal errors should give a `550` response. This means that there is some sort of catastrophic problem with the backend, and the client should immediately issue a `DONE` and disconnect.

DONE – End a Registration Session

Syntax: `DONE`

The `DONE` request indicates that the client is finished sending `DOIT` requests. The server should perform any post processing and the return a `221` response to the client, which causes the client to disconnect. Typical examples of post processing include generating NIS maps, rebuilding quotas, etc.

Note that it is the backend's responsibility to ensure that post processing succeeds. The client should not requeue requests if post processing fails. Therefore, the server should return a successful response whether the post processing succeeds or not.

References

[1] Dorner, S., and Pomes, P., *The CCSO Nameserver – A Description*, 1992.
[2] Steiner, J., Neuman, C., and Schiller, J., *Kerberos: An Authentication Service for Open Network Systems*, Proc. USENIX Winter Conference, January 1988. 30, 1988
[3] Steiner, J., and Geer, D., *Network Services in the Athena Environment*, July 21, 1988.
[4] "cpeople(1) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
[5] Schwartz, R., and Wall, L., Frogramming Perl, O'Reilly and Associates, 1991.
[6] Rosenstein, M., Geer, D., and Levine, P., *The Athena Service Management System*, 1991.
[7] O'Hern, T., *AGUS Account Generation Utility System*, UMBC, 1993.
[8] "sh(1) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
[9] "inetd(1M) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
[10] "NDBM(3B) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
[11] Sun Microsystems, *Network Information Services*, 1991.
[12] "crypt(3C) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
[13] Network Working Group RFC 821 "Simple Mail Transfer Protocol," Postel, J., 1982.
[14] Open Software Foundation, *OSF/Motif Programmer's Reference*, Prentice-Hall, 1991.
[15] Ousterhout, J., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
[16] "The Common Gateway Interface," URL http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.
[17] Allman, E., *Mail Systems and Addressing in 4.2bsd*, January 1983.
[18] "cron(1M) Manual Page," IRIX Reference Manual, Silicon Graphics, 1993.
[19] "rcp(1C) Manual Page," IRIX Reference Manual, Silicon Graphics, 1993.
[20] Network Working Group RFC 959 "File Transfer Protocol", Postel, J., and Reynolds, J., 1985.