The following paper was originally published in the
Proceedings of the Eleventh Systems Administration Conference (LISA '97)
San Diego, California, October 1997

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Monitoring Application Use with License Server Logs

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

One feature of our campus-wide UNIX service is the wide selection of scientific and engineering applications such as AutoCad, Pro/ENGINEER, Maple, etc. We currently have 32 "major application packages" site licensed, representing an annual cost of almost $300,000. A number of the licenses were based on concurrent usage, so around budget time, people started to ask if we had an appropriate number of licenses.

By adapting some previously developed software for tracking workstation use, we were able to determine who was using which applications, and concurrent usage information for these products and to reduce the number of concurrent users allowed to reflect actual use (plus some headroom). By applying these figures to just four applications, we were able to obtain a savings of $43,000 without cutting any service to our users.

This paper discusses the methods we used to collect, process, and display this information, as well as some of the problems we encountered.

Introduction

Three years ago, we presented a paper on monitoring workstation usage with a relational database [4]. This system worked by going to each machine and reading the `/var/adm/wtmp` files, and storing the contents in the database. When faced with the task of collecting application usage information, this system looked to provide a good starting point. We briefly examined at least one commercial product, but it did not take advantage of user demographic information – and the real show stopper at an educational site – it cost money.

While some of the applications we license are *node locked*,[1] a number of them rely on some sort of license server. One of our objectives is to have every application available on any of our machines. This way, rather than having to get a license for every machine, the license servers allow us to license a limited number of concurrent active copies of the application. As part of this process, the license servers often wrote log files with session information. While these log files were much different in format from the WTMP files, they were recording similar information. We were able to adapt our existing `Wtmp_Collector` program to pick up application usage information from these logs, and save them in database tables. This code was definitely worth keeping.

On the other hand, the original WTMP analysis program had some serious limitations; one of the big problems was the command line user interface. Each processing run required a number of parameters to be set, resulting in longer and longer command lines. Some common setting combinations were grouped together, but adding or changing them required recompiling the program. Even changing just one of the parameters would require rerunning the program, which repeated the database query step, obtaining the demographic information again. It was generally just a slow process, and it was just not ready to accommodate the demands placed on it to handle many different types of license server information. It also had to do much of the graphics processing. Although it was using jgraph [5] for actual postscript generation, there was still a lot of work involved, and there were some limits to what `jgraph`[2] could do for us. Given these limitations, as well as the need to accommodate changes in the data storage, we decided to replace the processing program. An X-based spreadsheet program we had already installed had an API that allowed us to load data directly into the spreadsheet, and build on the spreadsheet command menus. This also allowed us to use the graphics capability of the spreadsheet for output, as well as allow people to manipulate the data in a familiar spreadsheet environment.

With the updated data collector in place, and the new interface using the spreadsheet, we are now able to track the usage of many of our applications in addition to our workstations. This also simplifies the preparation of reports and graphs for management on a timely basis, without too much time or effort.

[1] A fun process where we have to supply a list of host ID numbers to the vendor, and they in turn return a list of magic numbers which we load into some file, and if the planets are aligned just right, things actually work. Doing this for hundreds of workstations is worthy of a paper in its own right.

[2] Jgraph is a program that reads a stream of simple drawing instructions, and generates PostScript on stdout. It works nicely in batch environments but it could not handle pie charts.

Taking License with Servers – Data Problems

The first step in the process of monitoring application use was to extract the raw data from the license server[3] log files and load it into the relational database tables. Although the information in the license server log files was in many ways similar to what we were getting from the WTMP logs, there were some significant differences as well. In addition, each type of log file had a different record format.

Database Record Format

One of the gains of this project was to put all license server records into a consistent format thereby simplifying later analysis and allowing us to compare usage of different applications. In determining the table layout, we started with the original WTMP table layout, redefined some columns, and added some new columns as seen in Table 1. We continued attempts to normalize some of the data at collection time, converting host names to host_IDs[4] and user names to owner ID[5].

[3]A note on the term *License Server*. From an operational standpoint, we often say that ''machine xxx is a license server''. That means that it is running one or more license server processes. However, in this paper, when we refer to a license server, we are generally referring to a specific process on a machine, providing licensing for one specific application or set of applications.

[4]A host_ID is an internal database key that identifies a particular host. Since all hosts of interest are already in the database, this key is easy to obtain.

[5]Like the host_ID, the owner_ID is a database key that uniquely identifies the user, even if the UNIX username is eventually re-used.

The first difference we had to accommodate between WTMP records and license server logs, was that while all of the WTMP records on a particular host applied to only that host, there may be more than one license server running on a given machine. In order to determine where in the particular log file it needed to resume processing, we needed to use both the `Lsrv_Host_ID` and `Application` columns to determine when we last collected records.

An additional complication was that some license servers provided license serving for more than one application; this required the addition of a subtype field in each record to differentiate between applications served by a particular license server process. To do this, we simply redefined the `Type` column, and for each license server (as defined in the `Application` column), would use one or more subtypes. Due to the number of different applications, both the `Application` and `Type` columns are required for the identification of a particular program product. This distinction would be significant for the data extraction process. It may have been preferable to make the `Type` column globally unique.

The last major change was the addition of a third host field in the record. In the wtmp records, we recorded from which host the record came, and for remote sessions, the host from which the user had connected to the target machine. With license servers, we also had to record the license server host in addition to the host on which the user was running the application, and in some cases, the actual location of the user.

Column	Type	Table	Notes
Username	char(8)	Both	Unix username.
Owner	Number	Both	Internal DB key for that username.
Host_ID	Number	Wtmp	Host_ID of the machine where the session took place.
Host_ID	Number	License	Host_ID of the user's machine, if available.
Lsrv_Host_ID	Number	License	Host_ID of the license server.
Application	char(4)	License	A short tag to identify which license server recorded the record.
On Time	Date	Both	The time and date when the session started.
Off Time	Date	Both	The time and date when the session terminated.
Line	char(12)	Wtmp	The TTY of the session, used to match disconnects with connects.
Line	char(12)	License	A tag to match session start with session end.
Type	char(1)	Wtmp	A code indicating the TYPE of session (telnet, ftp, etc.)
Type	char(1)	License	A code indicating subtypes for a license server.
Remote_Host	char(16)	Both	Machine where user originated the session if not the same as the host in host_ID above.
Rem_Host_ID	Number	Both	The host_ID of the remote host if it can be determined and is local.

Table 1: License Log Table Layout

Care and Feeding of License Server Logs

In order to provide a handle on the use of disk space by system log files, many of our log files are "rolled" on a periodic basis. That is, the existing file (named *filename*) is renamed to *filename*.1, *filename*.1 is renamed to *filename*.2, and so on for some number of generations[6] until the oldest one is deleted. This is normally done via cron and everyone is happy. As it turns out, many of the general UNIX log files are written via syslog, and fortunately, one of these "roll log file" cron jobs also `kill -HUP`s syslog so the new files are used. Other files, such as WTMP, are opened before each use, so the new file is used right away.

Unfortunately, the license servers generally do not write their log files via syslog, and so that while the log files might get rolled, the server keeps the file handle open and continues to write to the old file, leaving the newer files empty. To make matters worse, we found some cases of license servers that had been up long enough that the active file was rolled off the end and deleted. This resulted in no information being available, and the space being held by the still open log file until the license server was restarted or the machine rebooted.

This was not a good state of affairs. As a short term work around, we moved the roll code into the startup script for each license server, and set the number of generations to a moderately large number. We then scheduled a weekly reboot of each license server machine. This would ensure that each license server would get restarted (and the log files rolled) at least once a week. The collection process is still done on an occasional basis, and the large number of generations lets us get away with not collecting data frequently. The actual deletion of old log records is hopefully now being done by the collection process, rather than the log file rolling process.

Operation of the Collection Program

We periodically run the license server log collection program on all of our license serving machines. The general mode of operation is to scan all license server logs (and wtmp logs), and clean up when done. For testing, we generally run the program for just one specific type and without the clean option.

In scan mode, the collection program looks for a log file for each of the defined license servers and, if it finds one, checks in the database for the last time we collected records for that particular license server on that particular host. It then reads through the rolled log files until it finds the appropriate place to start. It next reads and processes the records, moving up through the rolled log files until it has read all the available records. If clean mode is also enabled, it will delete all but the most recent rolled log file. This leaves a little bit of a record behind on the machine, as well as the current active log file.

A structure built into the program stores the list of known license servers. This structure includes the application name for the license server (which is stored in the `Application` column of the database), the log file name, a time conversion routine, a parsing routine, and a list of valid subtypes (for translation into the `Type` column). Since many of the record formats are similar, this allows for a lot of code sharing between different log formats. While it might be preferable to store this information in a file instead of hard coding it, the parsing of the different formats is tricky enough that it is easier to hard code things than to encode the parsing information into a file.

Since the collection program deals with many different sources for the records, it is useful to convert the time and date into an easy to work with standard format. Internally, all date fields are converted to a standard form of seconds since 1 Jan 1970. Not counting the binary log file formats, we found four different date formats[7] that we had to handle.

Most of the logs are in a readable text format. You can read in a record, check the timestamp with the time conversion routine, and if you are interested in the record, call a sub-parse routine that returns pointers to username, remote host, record type, etc., which are then inserted into the database. The actual processing is based on the type of record. There is no standard for records types of course; each server uses its own keywords to indicate the type of record. So far, we have seen "OUT:", "IN:", "issued", "returned", "connect to", "closed connection to", and others along those lines.

The first record type is a session start. Before we insert a new record into the database, we check to see if there is a session open on that `Lsrv_Host_ID`, `Application` and `Line` combination; if there is, we terminate it on the assumption that we must have missed the session end record, since someone else is now using that same `Line`. This might happen in the case of an application that does not terminate cleanly, and so does not notify the license server on termination. We then insert a new session record into the database.

The second type of record is for a session end. For this case, the database is searched for a record for the same `Lsrv_Host_ID`, `Application` and `Line` that does not have an `Off_Time` set. Ideally there is only one, and the record is updated to reflect the end of the session. If we find more than one, we mark all of them as done, although we should set an error flag in the record.

[6] Actually, they are done in reverse order for obvious reasons.

[7] We wrote time conversion routines to handle: MM/DD HH:MI:SS, MM/DD HH:MI, MON DD HH:MI:SS (but no year!), and MM/DD/YY HH:MI:SS.

The third record type is a server restart. In this case, we go through and close all open records for that `Lsrv_Host_ID` and `Application`. The assumption here is that we will *not* get proper session end records, since the server state has most likely been lost.

Some license servers also generate time stamps. This is especially nice when you have backup servers running. For example, a server may be up for a long time, but may never be handling any requests since it is a backup server and the primary has stayed up. In such an instance, the timestamp records allow you to record that you at least checked the server.

Problem Data

We have run into some problems with attempting to process these log files. In one case, we were attempting to catch up on some old records (before we had implemented rolling for that particular license server), and discovered that the time stamps did not include the year, and we had three years' worth of data in the one file. The code that normally made a guess about the year ended up in a time warp. Fortunately, that was a minor service.

Some of the log files actually use multiple different formats in the file, and this can be very annoying when attempting to extract times or parse the fields. Other files are not in a text format. In some cases, such as the WTMP files, the record format is defined in a library. In other cases, such as the `NETLS` license server, the files are written in a proprietary database format, and instead of a record definition, they give you a program to extract data on demand. Although I was tempted to use this program, they had no facility to read an alternate database file, so all rolled information was lost.

Since we could not locate any documentation as to the file format, we had to reverse engineer the log file format. We were eventually able to figure out enough to eventually decode the records. A vendor supplied `struct` definition would have made things a lot easier.

Another source of error was with the assumption that keys we picked to identify session terminations were in fact unique. We had the case where one application was pretty slow to start up. Some users would double click on the application name, starting two copies. Sometimes they would even do it again, starting two more. Eventually, the first copy would appear, and then the second would appear right on top of the first. From the user point of view, they double clicked, and finally the application appeared and was operational. What the user did not know was that they had extra copies running in the background.)

We also had some license servers (or applications, we never determined who was not cooperating), that would not report the machine name on which the user was working. For example, if a user was using two workstations, or just had two sessions going, we faced the possibility of marking the wrong sessions as closed. We also had cases where the actual username and hostname that got logged was `anyuser@anyhost`; not very helpful.

Another potential source of problems comes from software upgrades. We have experienced problems with our WTMP logging after OS upgrades. On several occasions, we discovered that although we were successfully recording the start of a session, we were missing the end of a session, making it impossible to determine usage of labs. We could possibly see similar problems when we upgrade license servers.

In another case, we stopped getting session records from console users altogether, making it very difficult to determine if the workstation labs were being used. On reporting this to the vendor, we were told that it was not a bug, but rather that there was no interest in tracking console users. Finding this explanation unacceptable, we have since modified the login procedure to capture the information we need.

As we identified these various problems, we attempt to get help from the vendors in correcting them. We have also standardized our log file rolling script, and seem to be getting better at operating in a consistent manner.

Demographic Breakdowns or Breakdowns in Demographics

All of our user accounts are managed with a relational database, which can provide all sorts of demographic information about each userid, such as status (Student, Employee, Alumnus, etc.), departmental affiliation, campus address, and so forth. We are able to access this data when processing the log records from the WTMP database to provide additional information about the type of user.

We still faced some of the same problems with the demographic analysis, that we did with the original WTMP project. Specifically, while we do record the UNIX userid, the demographic information is obtained dynamically from the relational database, and that information is the current data, rather than the information that was in force at the time of the actual session. This results in what appears to be an increasing amount of use of our facilities by upperclassman and alumni as we examine older log records.

One approach to solving this problem is to add some additional fields. For example, when updating the demographic tables, rather than just making the change such as changing `Class_Year` from ''freshman'' to ''sophomore'', you could set an `End_Date` field in the current record to the current date and insert a new record with the `Start_Date` also set to the current date and the `Class_Year` field set to ''sophomore''. Similarly, when doing demographics lookups, instead of reading the latest demographic information, you select the record that matches the

appropriate time frame. In this way, you get the correct class year (or other desired demographic information) that applied at the time the session took place. This does however, involve changing the methods used to maintain the demographic data, and will cause the demographic tables to grow, rather then stay at a steady state size (since we will be adding records rather than just changing them.)

A second approach would be to record the demographic information of interest in the actual log records. This might be especially worthwhile if there a few fields that will be needed on a repeated basis, or that change frequently. Clearly however, some thought needs to go into the type of reports and breakdowns that will be needed. Another drawback to this approach is the delay between writing the record to the log file, and the eventual collection. If you are not collecting records on a timely basis, the demographics may change between the time of use and the time of collection.

Unlike the WTMP collection, which was confined just to hosts under central control, some of the licensed software is available for use by machines run by other departments. Unfortunately, there is no requirement that they use the same usernames that we use in the central systems. Although we can provide breakdowns of usage by machine and even by department, the user demographics are not valid for these ''foreign'' systems. We need to add a check in the collection program to determine if the user host is legal for demographic breakdowns or not, and include that status in record. Since the list of hosts where we can get valid demographics is subject to change as hosts are added to (or removed from) the central system over time, a `Demographics_Valid` flag needs to be set at data collection time.

Even if we are not able to determine valid demographic information on the users of a particular service, this data can still be of use. For one particular application, we were able to generate a pie chart showing usage by users in a particular department (based on host domain), versus the general population. With this, the administrators in the other department were able to justify paying for part of the cost of the application license, reducing the load on our software budget.

Programming to Xess – Talking to a Spreadsheet

One of the licensed applications we had for our UNIX workstations, was an X-based spreadsheet called `Xess`. One of the interesting features of this was an API toolkit [1] that allowed us to write a program that could connect to a running copy of the spreadsheet, add new menus and menu items, load data into the spreadsheet, and generate graphs.

Right out of the box, `Xess` looks like a normal spreadsheet, with tool bars and pull-down menus, and so on (see Figure 1). By opening the *Connections* menu, you can enable external connections[8] to the spreadsheet, which follow the X security rules. The external application is able to invoke many of the functions available to the console user, as well as insert data and formula into cells and read information back out. More importantly, it allowed us to define additional menus that could then invoke event-handling routines to perform the desired functions. This ''dual control'' feature was very useful during the development process, as we were able to have the program load some data into the spreadsheet, we could manually manipulate it, figure out pretty graphs and so on, and then encode the desired graphs and functions into one of our own menu items.

The data analysis generally consists of two phases, data selection followed by data processing. In the data selection phase, you need to specify the type or types of records you want, and the time period of interest. You may also want to specify additional conditions on the records. Once this is done, you then query the database for the desired records. As you can

[8]This can also be defaulted to **enabled** with an X resource setting.

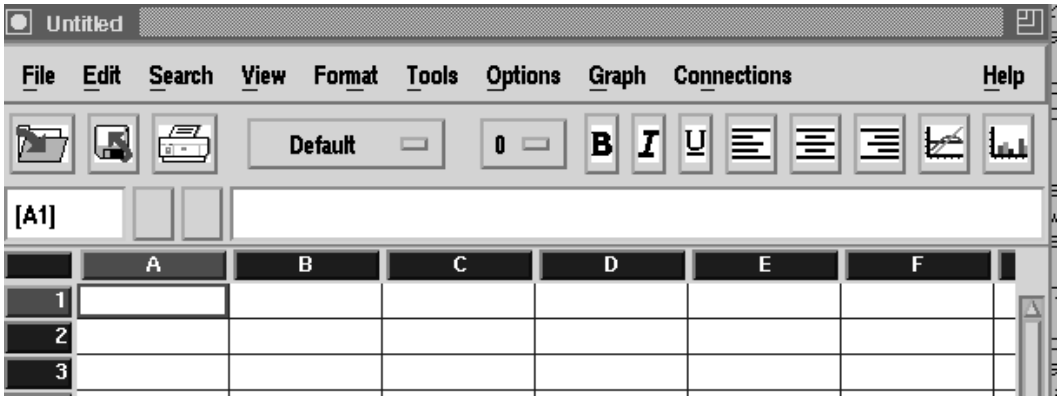

Figure 1: Default Xess Screen

see in Figure 2, we have added a number of additional menus to the spreadsheet session. Most of the selection phase is handled via the `Select` menu.

Once you have set the desired selection options, the last item you pick from the `Select` menu is the `Search Database` option, which extracts the data from the Oracle database. Once this is done, the `Select` menu is inactivated[9] and the `Process` menu is activated. At this point, you can select the type of data processing you want. The spreadsheet/menu interface made it easy to add more controls and options to the program. For the most part, the rest of the new menus just set flags and values in the program and modify the actions of the `Process` menu items. We frequently want to compare different demographic elements, and the `Column` and `Row` menus let us set what we put in the columns and what we put in the rows of the spreadsheet. We use the `DataType` menu to select what values we put into the spreadsheet (such as duration, number of sessions, etc.). A lot of different switch settings go into the `Options` menu. Unlike the previous three menus which act like radio buttons (selecting one deselects the others), the options are all set and cleared independently. During development, the `Debug` menu provided a handy place to put commands to provide debugging information, as well as to increase or decrease the verbosity level of the program.

The current version of Xess does not support cascading menus. This became a problem when dealing with some of the license servers that had a lot of sub-codes. We got around this in the `Select` menu, by prefixing each license server name with a *fraction* made up of the number of selected sub-codes over the total number of sub-codes for that particular server. The extraction program would also create (or rename) a new menu for that server, with each of the individual sub-code entries available for selection as desired. This approach might not be as elegant as cascading menus, but it works.

We frequently want to see usage as a function of the time of day, and often we want that to be an average for more than one day of use. To this end, we added a `Time In Columns` option to the `Datatype` menu. When this is selected, the columns of the spreadsheet each represent a specific time period. For example, in Table 2, we have few days of usage data.[10]

Although we could include the date with the time, and use more than 24 columns, we instead *wrap* the data at midnight and start the next day (we actually duplicate midnight at the other end). Once we have wrapped all of the data, we can simply insert a function to calculate an average load. In cases where there is a maximum number of concurrent sessions (such as a set number of licenses, or a fixed number of machines in a room for WTMP records) we were able

[9]Inactive menus and menu items are indicated by being grayed out. In Figure 2, the `Process` menu is currently inactive.

[10]This happens to be usage in a workstation lab, but the same processing applies to license server stats. Some data excluded for formatting reasons.

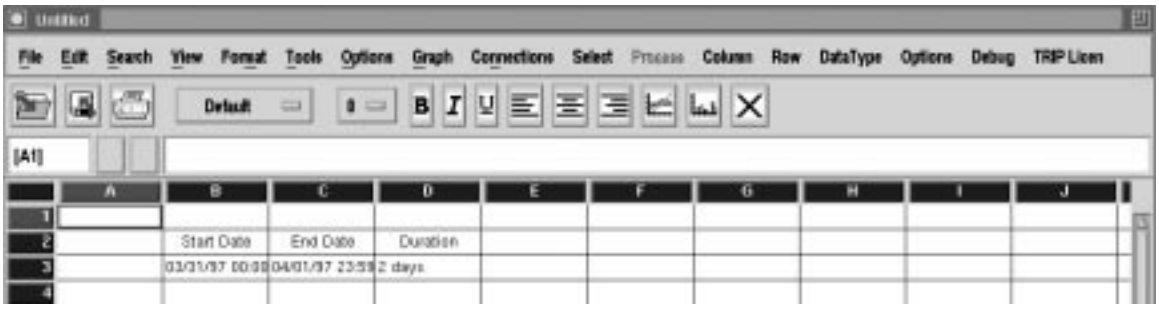

Figure 2: Enhanced Xess Screen

Date	00:00	01:00	02:00	03:00	04:00	05:00	06:00	07:00	. . .
12/02/96	5.471	1.480	0.096	0.300	0.001	0.000	0.024	1.524	
12/03/96	5.114	5.108	3.794	2.000	1.408	1.194	0.534	1.769	
12/04/96	5.816	2.973	3.604	2.053	1.056	1.000	1.200	1.768	
Mean(n=3)	4.995	3.848	3.537	2.794	1.741	1.244	1.124	1.336	
% Load	**71.4**	**55.0**	**50.5**	**39.9**	**24.9**	**17.8**	**16.1**	**19.1**	
Avg Load	0.714	0.550	0.505	0.399	0.249	0.178	0.161	0.191	

Table 2: Sample Spreadsheet Data.

figure a `Site Load`; the percentage of the resource in use, in this case, workstations. At this point, it is a trivial matter to have the spreadsheet generate a graph of average usage versus the time of day.

We often want to compare usage patterns over different time periods. To facilitate this, we added an option that, after the data is displayed, and the desired graphs are produced, it logically resets the origin of the spreadsheet (cell `A1`) to the row after the last active row, and then prompts for a new date range or search target. You can then select new data, load it into the spreadsheet, and generate new graphs for that selection. The program keeps track of the summary line of the past graphs, and combines them with the current summary, and generates a new graph with the different data sets displayed on the same axis. In Figure 3 for example, we have four weeks of data for two different labs, displayed on the same axis.
We added options to dump data in a raw form, and with selected demographic information added in case the researcher wants to do some other types of analysis. We were also able to dump session information to generate histograms of session duration, using the `histogram` graph option of Xess. Since all of the original functions of Xess are still available, after someone selects and processes data, does additional calculations, figures out what graphs they want and so on, they can then save the spreadsheet, using the options in the standard `file` menu, and allow others to look at the information at a later date. In fact, the data in Table 2 and graph in Figure 3 were taken from a run done months ago, and reloaded into Xess for this paper.

It's Easier the Second Time Around

In the original WTMP analysis, we were able to convert session data to load information, by creating a time-line, divided up into discrete segments of time, and ''adding'' in sessions to the appropriate bucket. Since this worked well, we decided to keep it. A distinct advantage we had in this rewrite is that we were building a new application from the ground up, yet we had a pretty clear idea of where we wanted to go, and what problems we had encountered in the first implementation. This helped us with designing data structures and the data flow through the program.

In the original program, it was sometimes difficult to break down the data by any arbitrary demographic value. There was a lot of special case code, and adding new types was tricky. In addition, we often wanted to break down records by two keys at once, which did not work at all well (you could break down in one direction, but not the other).

The new program needed to be much more flexible, and ideally, all data should be an abstraction. We also had a solid idea of what we were going to do with the data; specifically, dump it into a spreadsheet for display. This gave us three ways to break down the data; just provide a total in a single cell, which would be a zero axis case, break down by one demographic item, producing a single row or column, which would be the one axis case, or break it down by two demographic items, which would be a two axis case, essentially a mesh.[11] The determination as to how to divide the data, is controlled by the `Row` and `Column` menus.

As a record is read in, the row and column demographic elements are determined, and then they are compared to existing entries in a multiply-linked structure, and inserted or added to existing entries. In Table 3, we are looking at the number of sessions of Data Explorer, and breaking it down by the school affiliation in the rows, and the class year in the columns. Each entry is added in three times; once for

[11]The code to manage all of the data elements in the mesh was a good exercise in pointers and linked lists. I was able to successfully handle the three cases of n=0,1 or 2. I never tried to take it into more dimensions, although that might be possible.

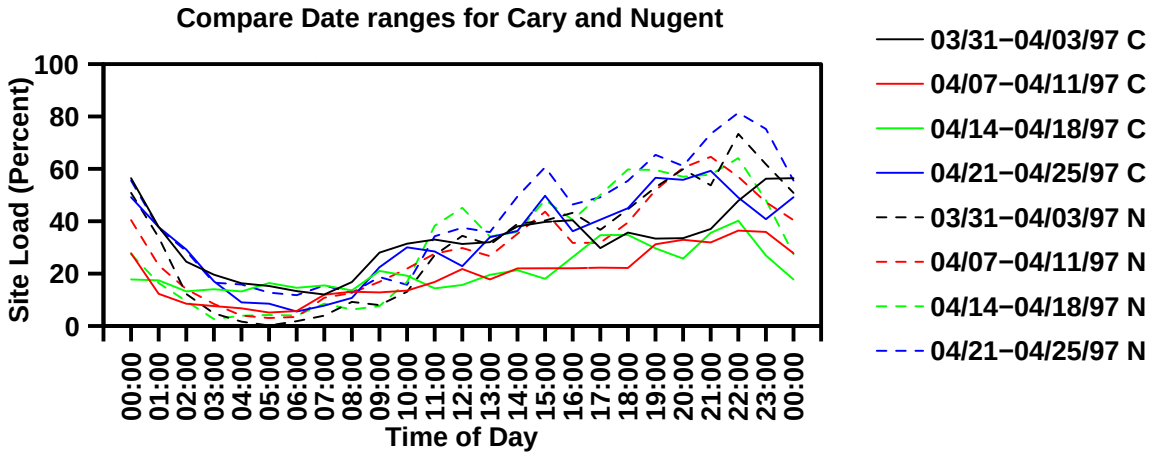

Figure 3: Combining Graphs on one Axis

the school, once for the class year, and once for that combination of school and class year. In this case, we can see the students in the school of science make the most use of the application, but that is mostly by Ph.D. candidates, whereas the Engineering students use it at the Masters and Undergraduate level as well.

When we query the database, we currently get back a linked list of database rows. This is due to our current use of our RSQL[12] interface to Oracle. When processing the selected data, each row is in turned loaded into a mesh data element, and passed along for processing into the mesh. Although this does increase memory usage, keeping the original query results in memory allows for very fast processing when changing demographic choices – all of the data is already extracted from the database. This is a big improvement over the old program, by caching all of the raw data and demographic information, we can readily play with the data and try different types of analysis without much of the delay we used to have.

Since this is one of the few places that we actually interact with the database, this routine provides a clean place to change the database API. While the RSQL is great for small queries, you do pay a performance penalty for larger queries. One area for future work is to replace this with a PRO*C[13] interface to allow block transfers of data. This can result in much better data transfer rates from the database.

If time-of-day processing is enabled, each mesh element also gets a pointer to an array of buckets, representing a time-line. This will hold session duration information that can later be extracted to produce the concurrent usage graphs shown earlier. I did not spend any time with changing the bucket size, although this could prove meaningful later on.

Space, the Final Frontier

In the original WTMP logging project, all WTMP records were stored in a single table. While this worked at first, after a year or so, this ever-growing table was becoming more difficult to manage. During system upgrades, we would have to export this single, multi-million record table to a single file, and finding single chunks of disk space big enough was becoming a definite challenge.

To work around this problem, we decided to break up the data into one-month sections, determined by the start date of the record. We now create a database table for each month, basically appending *_YY_MM* onto the table name. At collection time, insertions are easy; you just take the connect time and generate the table name. Session terminations are a bit trickier, as a long session may have started in the previous month, or even earlier. We therefore currently look back two months for a session start record.

The change for the analysis program was also moderately simple. Instead of just taking the date range and returning a table name, we return a list of table names. The selection process then queries each table in turn, returning a set of record for each month. These are then processed into the mesh in turn. The only challenge remaining is to be sure to have the new tables created ahead of time so there is a place to receive new data.

In addition, we also had started systematic backups of the Simon database [2, 3], and the WTMP data became the bulk of the data being backed up. The Oracle database maintains a record of all transactions since the last full backup. These transaction files multiplied quickly with the log collection, requiring more frequent full backups, and a lot more free disk space on the database machine to hold transaction files between backups.

The Simon database is used to manage all of our UNIX accounts, disk accounting, and many other aspects of our operation. Since the operational need for the WTMP and license server data was much less than the rest of the Simon data, we decided to move the WTMP data into a different database instance that was not being backed up. This removed the load on the backup system, while still allowing us to back up the mission critical Simon data.

Although the log record tables could continue to grow in size, in practice, we have not always reloaded the older data when moving the database between machines. Also, as the data ages, the problems with demographics get worse.

[12] We developed a layer that runs on top of the Oracle Call Interface that simplifies writing applications that need to interface with Oracle, it handles network authentication, space management and many other details of coding with Oracle.

[13] PRO*C is the Oracle pre-compiler for embedding SQL queries (and other calls to the database) directly in a C program.

Sum		DOC	MAS	SR
	Cat Value	540331	54698	53852
Sci	460870	460354	0	516
Eng	188011	79977	54698	53336

Table 3: Data Explorer use by School and Class.

Findings and Losses

Although we had continued to collect WTMP data, until the license server collection project had started to heat up, no one had really spent much time doing analysis of the WTMP data. When we started planning for some workstation lab upgrades, we went to generate some nice reports with the new tools and discovered that as a result of an OS upgrade, we did not have accurate termination information for most of our public workstations labs. As a fallback position, we did some analysis on just the session start times and we were able to draw some conclusions on workstation usage.

Despite the problems with the workstation usage statistics, we were able to use them in some decision making process. In addition, the very concrete benefits from the license server analysis has brought a good deal of attention and support to the statistic collection and analysis project. Additional staff have been assigned to work with the system on an ongoing basis, and session data collection will be integrated into our existing session management software.

The information collected allowed us to measure the actual usage of a number of our licensed applications, and in some cases, we could reduce the number of copies licensed, resulting in significant savings in license fees. This savings should help maintain interest and support for the project. We also need to find ways to measure the usage of applications that do not use this type of license server.

This actually opens up other questions about measuring application usage. Given that this project grew out of measuring workstation use, essentially seat time, most of the information has a bias to session duration. This makes a lot of sense for an interactive program such as autocad, but makes less sense for an application that might remain open but unused for most of a session such as a mail reader. Recording session duration is just about meaningless for applications such as compilers and text processors; the fact that someone spends a lot of time running latex may be more a reflection on the speed of their machine and not on the type of work they are doing.

References and Availability

All source code for the Simon system is available for anonymous FTP. See `ftp://ftp.rpi.edu/pub/its-release/simon/README.simon` for details. In addition, all of the Oracle table definitions are available at `http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html`.

Although this system benefits from being able to extract demographic information from Simon, and from actually storing the raw data in the database, both of these areas are pretty well isolated, so other databases, or even flat files could be used in place of the connections to Simon.

Xess is a commercial spreadsheet product available for many UNIX platforms and WindowsNT. More information on Xess can be obtained from:

Applied Information Systems, Inc.
100 Europa Drive Chapel Hill, NC 27514 USA
1-919-942-7801
1-800-334-5510
1-919-493-7563 FAX
info@ais.com.
http://www.ais.com

Acknowledgments

Many thanks to Adam S. Moskowitz and Debra Wentorf for their help in editing several drafts of this paper (although I still take full responsibility for all misplaced commas and poor grammar) and also to Jackie Blendell for her help with the original abstract. Thanks also to my manager, Kathy Bursese, who encouraged preparation and submission of the paper.

Author Information

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming, with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past six years. He is currently a Senior Systems Programmer in the Server Support Services department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems, and also deals with information security concerns. Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via http://www.rpi.edu/~finkej.

Bibliography

[1] Applied Information Systems Inc, Chapel Hill, NC. *Xess Connections API Toolkit Guide – Version 3.1*, 1996. http://www.ais.com.

[2] Jon Finke. Automated userid management. In *Proceedings of Community Workshop '92*, Troy, NY, June 1992. Rensselear Polytechnic Institute. Paper 3-5.

[3] Jon Finke. Relational database + automated sysadmin = simon. Boston, MA, July 1993. Sun Users Group. Invited Talk for SUG-East 93.

[4] Jon Finke. Monitoring usage of workstations with a relational database. In *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pages 149-158. Rensselear Polytechnic

Institute, USENIX, September 1994. San Diego, CA.

[5] James S. Plank. Jgraph – a filter for plotting graphs in postscript. In *USENIX Technical Conference Proceedings*, pages 61-66. Princeton University, USENIX, January 1993. San Diego, CA.