Monitoring Usage of Workstations with a Relational Database

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

The ability to monitor usage of groups of workstations is quite useful for planning growth, facility hours, staffing and other issues; but in our case, both the format of the data (*/var/adm/wtmp*) and the fact that the data was spread over hundreds of different workstations made any analysis difficult at best.

In this paper, we explore the use of a relational database to collect all the raw data, convert it to a standard form, and then provide selection tools to extract data sets. We also examine some ways to process session data to provide more meaningful reports and charts for administrators.

Motivation

The primary campus computing system at RPI is a collection of over 400 color graphic workstations from both Sun and IBM, as well as some larger Sun and IBM *Timesharing* machines. The workstations are deployed in ''workstation classrooms'' of 25 to 30 machines, in smaller ''dorm lab'' clusters located in student housing, and as individual workstations on the desks of faculty and staff, as well as in laboratories.

The volume of data, on the order of one million records per semester, and the fact that it is spread over a large number of machines, makes it difficult to handle. In addition, we generally want to see usage patterns in a group of machines, and don't really care about the use of any individual machine in a group. We also have to deal with Suns and IBMs using different formats for their usage data (*/var/adm/wtmp*).

We wanted to be able look at the data in different ways. We need a way to determine what workstation clusters are filling up, and what sort of usage there is for any given time of day, or day of week. This will help our users determine when and where to go to find workstations, and assist us in figuring which buildings need more workstations and which ones need less! We are also able to compare if the users prefer one type of workstation over another, and if that holds for all sites.

Much of the funding for Rensselaer Computing System (RCS), was in support of computer enhanced learning, with a strong emphasis on teaching instead of research. A number of undergraduate courses are having their curricula revised to integrate use of the graphics workstations into the course. This increased interest in not only whether a workstation was in use, but the type of use, or at least the type of user who was using it. While the basic data contains a username, it does not have any demographic categories of the user. Being able to find out more about the user is desirable.

Solution

Several years ago, in response to a series of break-ins, we started a project to collect WTMP data in a central location to assist in locating connections from off campus, and odd usage patterns. This involved periodically ''printing'' the wtmp files to a virtual printer on our mainframe, where duplicate records would be discarded and new records would be saved for later analysis. While this got us through the immediate problem at hand, it did not take into account operational practices on the workstations (rolling wtmp files[1]), and sent huge amounts of duplicate data. This eventually overtaxed the print queuing system and jeopardized our print service (and the data collected was so difficult to work with), that we had to shut the data collection down.

A few years later, interest in gathering usage information had risen to the point where we needed to take another shot at the problem. While the previous solution (virtual printer) had been a failure, it did teach us some very valuable lessons, such as the need to handle the aging practices for */var/adm/wtmp*, and the need to only send back **NEW** data to the central collecting site. Given that I had just finished the Simon Hostmaster[2] project, collecting host usage data via Simon seemed to be natural.

[1]We periodically *roll* log files from say *wtmp* to *wtmp.0*. If there already is an old version (*wtmp.0*), we roll that to *wtmp.1*, and so on. Depending on the frequency of the roll, and the size and type of file, we keep from 2 to 9 old generations around.

[2]Simon Hostmaster is part of RPI's database driven system administration package, known as Simon, that manages all the hostname and address information for the name servers and host files.

Our solution breaks into three main areas. The first, data collection, deals with gathering all new data from each machines, doing some initial cleanup on it, and storing it into the database. The second area, data selection, deals with how we extract only the desired set of records from the database, provide additional preprocessing of the data if needed, and then pass it along for further processing. The third area, data modeling, is where we actually do some analysis on the data. This may involve building a virtual workstation lab, loading the usage data into the lab, and then analyzing the lab use.

Collecting Data

The data collection is done using a program, *wtmp_load*, that runs on each of the subject machines. It determines the last time we loaded data from the current machine, and then it scans through the wtmp file(s) for the first new record. If the first record in the wtmp file is newer than the last time we loaded data from the host, we back up one generation to the *wtmp.0* and check again. We keep backing up until we run out of old files, or find one that has older data in it. From that point, it starts reading forward until it finds a record that is later than the last time, and then loads the records into the database. If we had to back up to an older version of the *wtmp* file, we process each file in turn until we have loaded all the records.

A WTMP record is written for each signon, each sign-off, and depending on the actual operating system, for a number of other system events such as reboots, time changes, etc. It is important to note that there is a distinct sign-off record. We do not get session records in the wtmp file. Since later analysis will want to deal with sessions, we will attempt to build session records at collection time. By linking the start and end of a session at collection time, we don't have to do that work each time we analyze the data.

Storing Data

All of the wtmp data collected is stored in the Oracle table **WTMP_LOG**. We have defined the following columns[3].

username *char(8)* The Username of the user. There are special usernames such as shutdown, reboot, etc....

host_id *number* The Simon.Dns_Domains. Domain_Id of the host that these records are taken from.

connect_time *date(7)* The time when the connection was made.

disconnect_time *date(7)* The time when the connection was terminated. This value is usually only added via an update to an existing record.

line *char(12)* The symbolic name of the device that the connection was made through.

type *char(1)* A flag indicating the type of connection.

remote_host *char(16)* The name of the remote host involved with the connection. This may not be the full name due to truncation problems.

remote_host_id *number* The Simon.Dns_Domains. Domain_Id of the remote host, if it appears to be in Simon, (we have to make some assumptions here due to length limits.)

Sign On Record Processing

When we process a signon record, we attempt to classify its session type (remote telnet, X console, remote X, ftp, etc) to simplify later analysis. This also helps eliminate operating system differences, which would complicate later analysis.

We also work to match up the partial remote host name from the wtmp record (esp relevant for the timeshare machines) with our own host database. Frequently the remote host name is truncated when it is stored, since at least some wtmp definitions only allow 16 bytes for the hostname. We declare it a match if there is at least one ''.'' in the partial name, and if we can get an exact match with the first 16 characters of an RPI hostname. For names that match, we store the resulting host id in **Wtmp_Log.Remote_Host_Id**. While we will miss hosts with very long names[4] and we may have some foreign hosts that match the first 16 characters of an RPI host, and so are miscounted, we should still end up with the **Remote_Host_Id** set correctly in most cases.

While we intended to run the *wtmp_load* program on a frequent (actually, continuous) basis, runs were often weeks, sometimes months apart. This resulted in a lot of wtmp records to be processed, which in turn generated many queries to the Hostmaster tables to resolve partial names. Host name lookups go from right to left, first finding ''edu'', and then finding ''rpi'' and so on.

We finally added two levels of caching, first of the 16 character partial names, even if they did not match, and a level deeper, of individual parts of a domain name such as ''edu'' and ''rpi''. By seeding this lower level cache with ''its''[5], ''rpi'' and ''edu'', we cut the number of database queries in half. Both forms of caching made very noticeable improvements to the performance of the program, and reduced the load on the database server.

[3] Some unused columns have been removed from this listing.

[4] There are 8 hosts at RPI that have a base name 15 characters or longer in a population of around 3700 hosts.

[5] Information Technology Services is the department that runs all of the RCS machines, so many of them are in the ''ITS.RPI.EDU'' domain.

Sign Off Record Processing

When we process a sign off record, rather than insert another record into the database, we attempt to locate the corresponding signon record in **Wtmp_Log** and set the **Disconnect_Time** field. We only have the ''line'' information, so we have to look for all records for this host that have that ''line'' and **Disconnect_Time** has not been set. While this works most of the time, we do on occasion encounter more than one record. This means that either the first session's sign off wtmp record never got written, or got lost somehow.

Missing sign off records are a source of error in the data. One way to reduce, but not eliminate, is to have the Signon processing first close out any pending records. It would also be good to mark all of these records as ''suspect''. Likewise with the sign off records, when you have more than one, the older ones should be marked as suspect, although if Signon processing closes open records, no extras should be found at sign off.

We encountered one type of system that never wrote a sign off record at all for a certain class of connection. Once we started working with the data, we quickly discovered this problem when it reported many people using the same X station at the same time. We also had problems with some FTP sessions not producing a sign off record.

Other System Activity

We ignore everything except reboot records. When we process a reboot record, we close all pending session records for that host. Again, these records should be marked as suspect.

Between the attempts to resolve partial host names, and gaps in records caused by system crashes, the data being collected is by no means perfect, but for the most part, is clean enough to be useful. You wouldn't want to generate bills from it, but as long as you understand where errors can creep, in you should be ok. In attempting to track down some of these, we discovered an undocumented bug in some of the time conversion routines. Specifically setting the tm_isdst to -1 fixed this problem. Before that, it would intermittently add or subtract an hour.

Selecting Data

Selecting data breaks into three parts: first determine which records we want based on attributes of the records; second determining which fields we actually want to return; last, any pre-process the records need to format columns outside of ways the database can handle, before passing the selected records on to the analysis section.

Which records

Any of the columns in the **wtmp_log** table can limit the selection of records. In fact, it is possible to extend the selection choices by joining[6] the **wtmp_log** table with other Simon tables. For example, via the **Username** column, we can join to the **Logins** table and only extract student users.

In practice, the first constraint is to just select the records for a single host, or a group of hosts. This is done by requiring that the **Wtmp_Log.Host_Id** is equal to a specific host id, or belongs to a specific host group. Host groups are described in more detail in a later section. In the case of an X station lab[7], we instead select the records where the **Remote_Host_Id** belongs to the host group for the desired X station lab.

Another common constraint, are the starting and ending times. We actually select all records where the **Disconnect_Time** is greater than the start time, and **Connect_Time** is less than the ending time. This ensures that we get all records that fall into the desired time range.

Often, just specifying a host group and time range is enough. There are other cases where we want to place a fancy constraint, such as the join example from above, but more often we just want to look at a single type of connection. For example, when looking at the data for a workstation lab, you often only want to know if the console is in use. It doesn't really matter if a staff member is telneted to the workstation, as long as the console is available for use. In this case, we would add the additional constraint that the **Wtmp_Log.Type** would be ''X'', indicating an X console session.

Which Columns?

Given that we have determined which records we are going to select, the next step is to determine which columns to select. We always want the session start and end times; the rest is up to the question we are asking at the time. In the current implementation, we return a linked list of a structure that has three different columns, as well as the start and end time for each record. By convention, the first column is a numeric, and the other two are strings.

A common choice for a workstation lab, is **host_id**, **Username** and **User_Class**. In point of fact, we don't do anything with the Username, and due to privacy concerns, we could not publish a report with usernames in it. Username[8] is actually used in the database itself to find out the

[6] With a relational database, you can ''join'' the contents of one table with the contents of a second table based on a common column between the tables. This is a very powerful function.

[7] For X station labs, the wtmp records are collected on the actual machine providing the CPU cycles, and the X station is considered the remote host in this case.

[8] We manage all user accounts via another Simon module, based on data from the Registrar. This enables us to match up a given username with the corresponding student records, which has been valuable cases as well.

classification of the user (Freshman, ..., Phd, Fac/Staff). It is in this type of join that the power of the relational database comes into play. This has enabled us to study for example, whether lab users live off campus, on campus or on campus in a dorm with a workstation lab. The potential here is amazing. For the sake of example, assume we have a table **User_Info** with the following columns defined:

Username *char(8)* The Unix Username. There is a record here for all active user accounts.

Classification *char(4)* The current classification (FR,SO,JR,SR,Grad,Misc) of the userid. The status ''Misc'' includes faculty, staff and guests.

To get the classification, we would select something like the following:

```
Select Host_Id, Classification,
       Connect_Time, Disconnect_Time
  from Wtmp_Log, User_Info
 where Host_Id in (Sub Selection)
   and Connect_Time < $ENDTIME
   and Disconnect_Time > $STARTIME
   and Type='X'
   and Wtmp_Log.Username=
                 User_Info.Username
 order by Host_Id, Connect_Time
```

The statement (*Sub Selection*) actually refers to a nested select statement which returns the list of host_ids for the group we are interested in. This will be discussed in detail in the section on host groups. The first two ''and'' statements establish the starting and ending time constraints, and the third ''and'' statement sets the type constraint. In the last ''and'' statement, we joined the **Wtmp_Log** table with the **User_Info** table to get the **Classification** returned with each of the records.

Fixing Data

While we can do a lot with joins in the database to get what we want, sometimes, there are ways of classifying data that seem too complicated to get directly from the database. For this, we simply run the data from the selection process through a routine that converts one of the data fields in place to some new classification.

An example of this, is when we wanted to look at what sort of people were using our remote access (timesharing) Unix service. We convert **Remote_Host_Id** into one of the following cases: Terminal Server, On Campus RCS Host, On Campus non RCS host, Student machine or off campus. With a combination of host groups, string compares, and other smoke and mirrors, we were able to convert the remote host info into what we wanted, and the existing analysis routines were happy to produce results for us.

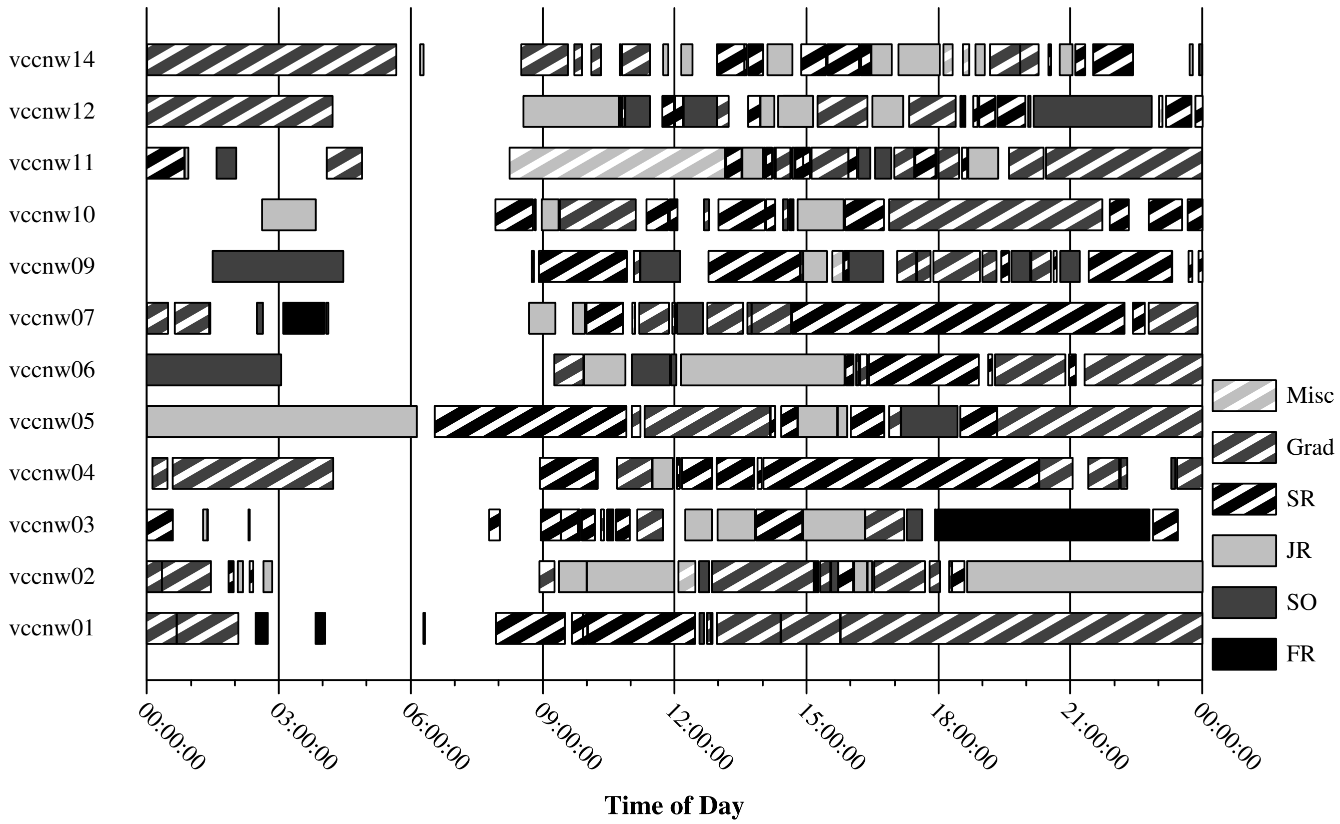

Figure 1: Raw Session Data

Sometimes, at this stage, we simply dump the records we have selected into a file to allow for analysis with other products such as SAS.

Verifying the Data

Now that we have selected a set of rows, and figured out which columns from those rows are of interest, we wanted some quick way to look at the lab use before we actually start modeling it. To this end, we generate a bar graph like Figure 1. In this case, we take a small workstation cluster, and look at all the records for a particular day. We select the host_id and a user classification, which is derived from the username. Each of the hosts has a set of bars that correspond to a user session. The shading and patterning indicates how we classify the user (in this case.) The time axis runs from midnight to midnight. This type of output has proven very useful in finding problems with the raw data. The X station problem mentioned earlier, showed up as more and more sessions overlapping. Since that is not possible, it indicated a problem with the data. This also can show unexpected gaps in the data. This is often due to a broken workstation. While this was not intended for the formal reports, this format has proven useful to show the user mix in the labs. The actual output is much more impressive in color.

Modeling Data

One of the initial objectives, was to generate a graph showing number of machines in use, at any given time of the day or night. Logically, if we take the graph in figure 1, and draw a vertical line at some particular point in time, then count the number of times it intersects a horizontal bar, we then have a user count for that time. We advance the vertical line to a new point, some fixed distance from the previous point, and repeat. This isn't a new concept; I seem to recall something like this from a freshman calculus class, long ago.

Moving that model into the computer is mostly a matter of picking some data structures. For ease of processing, I broke up the time line up into a set of discrete ''buckets'' with an array element for each bucket. Given that we had the start time, end time and bucket size (or duration), it is trivial to figure the number of buckets or elements in the array. For each record, we simply converted the start time to an array index, and looped through until we hit the end time. In practice, I ran several of these array structures, a master array (all hosts), a linked list working on the primary key (such as the host_id), and a second linked list working on the secondary key (such as the classification).

If we take the master array, and use 5 minute buckets, we get a simple graph like Figure 2. If you compare this with the data shown in **Figure 1**, we can see where every machine is empty at about 6:20 AM, and then the lab is in constant use for the rest of the day. There is a slight dip at lunchtime, which is a little more visible on Figure 2, than it is on Figure 1.

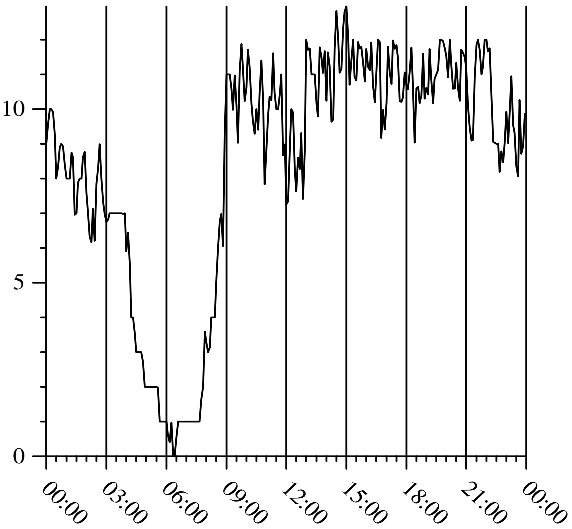

Figure 2: Simple Lab Use

We can also dump more than one of the chains on the same axis. Putting the 13 different hosts on one axis, where they only value they can have is 1 or 0, would be pretty boring, but if we take the other chain, classification, we display (see Figure 3) undergrads as a solid line, grad students as a dotted lines, and fac/staff as a dashed line.

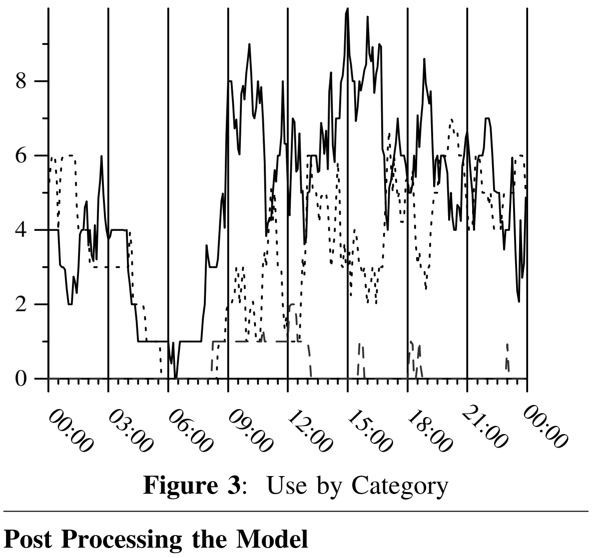

Figure 3: Use by Category

Post Processing the Model

Well, all of this provides a start, but what you often want to do is look at the average use of a site, so the program can take data for say 5 days, and calculate a mean value for each bucket, and since Rensselaer is an engineering school, figure a standard deviation of the mean, and put that on as well, as seen in Figure 4.

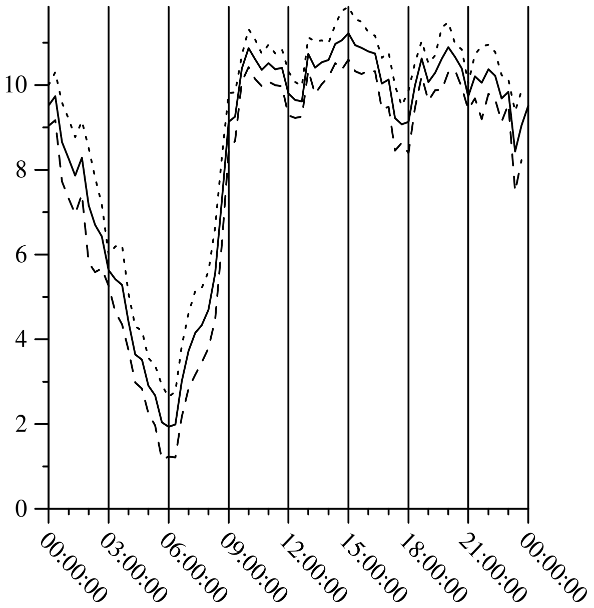

Figure 4: Mean Usage

You can also do things like put two different labs on the same axis to compare usage patterns, overlay different days of the week to look for scheduling differences, etc.

We found the ability to overlay two different labs on to the same axis quite useful. One of the objectives of RCS, is to allow the student to work on any of several different platforms (Sun, RS/6000, X station) and move between them on a daily basis. This allows the students to ''vote with their feet'', as to what is the preferred workstation. Given a choice between an X station and a workstation with local display, there is appears to be a preference for the workstation, except in the cases of dorm labs, where convenience appears to win out over technical attributes. An example of this, is in Figure 5, where we take a five day average of workstation console use (the solid line), and then on the same axis, put a five day average of X station (Remote host) use (the dashed line).

Implementation

The wtmp logging project got put on hold this past winter, when the database machine ran out of disk space. Since a new disk and a new database machine were imminent, we stopped collecting data (and letting it accumulate on each individual machine) until we could move to the new machine. That move is currently scheduled for mid August.[9] At that point we intended to start with a clean slate and start collecting data from all machines, all of the time. Before we ran out of space, we collected over 1,000,000 session records from over 400 machines. One critical item for performance, is an index on **Host_Id** and **Line** since this is the most common query for *wtmp_load.*

The data collection will be done with the *load_wtmp* program. I expect that when operation resumes, we will run it in ''sleep'' mode. In this mode, when it first runs, it will connect to the database, update whatever records are available, close the database connection, and sleep for some length of time. It will periodically wake up to see if anything has changed in */var/adm/wtmp*, and if so, process any new records, and go back to sleep. It also

[9]I think we need to work on our definition of *imminent*

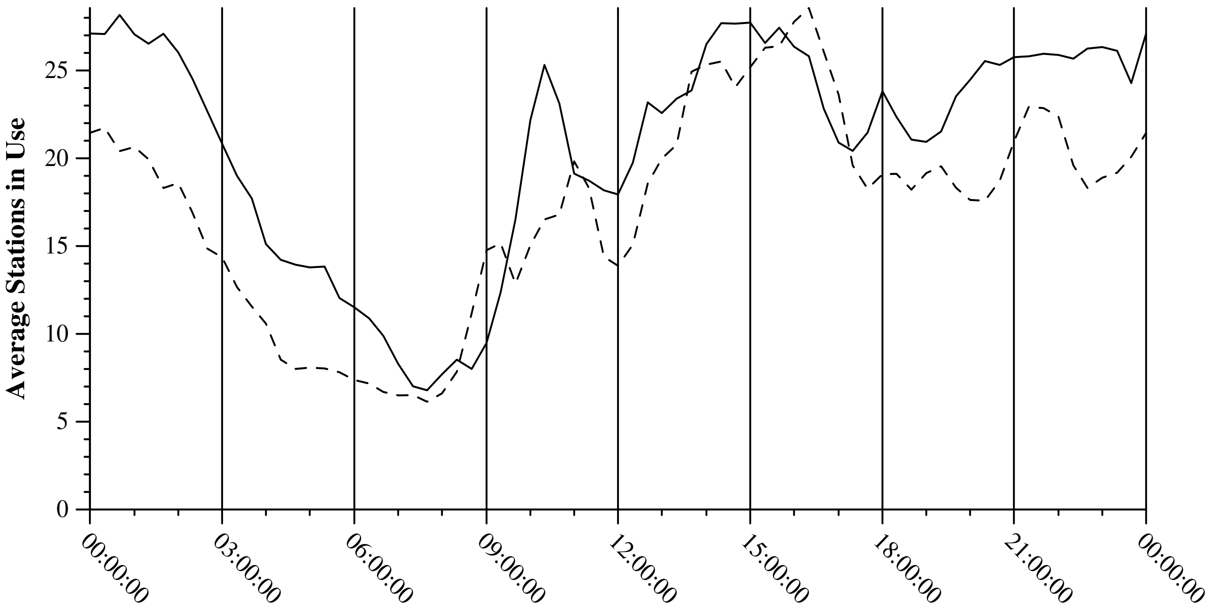

Figure 5: Workstations vs X Stations

has to handle the wtmp file be rolled out from under it. This should allow us to keep the database up to date, and this may also provide for a periodic health check for machines.

One of the parameters we will be working with, is the length of time the process ''sleeps'' on different types of machines. For a single workstation, the average session is 25 minutes or so, so a frequent check (every 5 minutes) would let us keep up to date, but not put too much load on the database machine. Things are different on the remote access machines. These run on the order of 50 sessions an hour, so a frequent wake up, would result in a lot of database activity. The other approach is to set a long interval (such as 24 hours) between updates. We are assuming that there is non trivial overhead in establishing a database connection, so that there are gains in batching records. One thing we need to avoid, is a bunch of machines all attempting to dump data at the same time. Since they are contending for some of the same resources on the database machine, this is likely to cause performance problems. One performance win we do get with sleep mode, especially on the remote access machines, is that they are building up a cache of host names over time.

The other half of this project is the *lab_use* program, which is used to make the queries and generate the graphs. This program is a product of evolution, once we produced one set of reports, the vice provost would ask another set of questions, which would involve more changes. As time went on, we learned what types of queries made sense with what types of output displays. For instance, the graph of mean use doesn't really make sense for just a few days, nor does a simple graph such as Figure 2 make any sense for three month time period. We did learn some things in the process, and while we made some mistakes, we also did some things well.

What We Did Wrong

In general, the single biggest problem with *lab_use*, is that we did not have a clear design specification in mind. As a result, the program has had some major internal restructuring, and some cruft from earlier versions still lingers. In some respects, this was unavoidable, as this was a crash project to get some numbers to the administration.

As we started writing *lab_use*, we got tired of providing a huge number of command line parameters, so we eventually built a structure to hold common queries, so we could just specify an entry from the struct. While quite useful in the beginning, this has turned out to be difficult to update and now serves to slow development in some areas.

Another problem, stemming in part from the development approach we took with *lab_use*, is the original lack of program structure. This has required us to go back and split up the program into different modules, to allow better reuse of code. As a result, not all of the graphing routines use the same color selection routines, some processing routines make graphing calls directly, while other simply fill an array and pass that off to be plotted and so on. However, given what we have learned, we could now go back and redesign it.

What we did right

On the other hand, a number of things have worked out well. For instance, the host group and time constraints have worked quite well, and certainly should be kept in any revision. Another useful development is a set of time conversion routines which convert the time values we get from oracle into a handy internal format[10] and back again. While these are pretty simple routines that make heavy use of the existing C routines, they do handle some vendor differences, assist in default formats, allocate space as needed and so on.

Another big win on this project, is the use of **jgraph**[1], written by Jim Plank at Princeton has proven quite useful and quite a time-saver. If you need to generate graphs from a program, consider this package. This also made it easy to scale and edit the figures for this paper.

As the program evolved, some structure did start to appear: libraries to perform calculations and store the results into standard structures; libraries to convert the standard structures into jgraph input files. The time library mentioned above and one to help with host groups. One of the snags we hit was in labeling all of the graphs. Since there are 4 or 5 different output formats, and dozens of different selections defined, we started to store graph labels in an oracle table, so you can specify the format and the selection, and find the appropriate label for that graph. I expect that this approach would work well for storing the queries.

One of the biggest wins was basing the whole project on the relational database. Besides the options that joining other tables provide, it gives us a lot of data independence. While many of the internal variable names reflect the original selection choices, that is hidden from the end user so we can process other columns than the ones in **wtmp_log**.

Future Directions

This is the type of project that will never end. This fall, I expect to start the data collection on all machines, into the newer, bigger database machine. The *lab_use* program is likely to remain unchanged until the next round of questions start coming from the directors. At that point I hope to continue some of the cleanup and restructuring. Given the existing queries that it supports, I want to identify the

[10] What could be more convenient than the number of seconds since 1970, stored in a long integer...

attributes of each type of selection and displays, and come up with a more idiot proof way of determining output options.

Another are I want to work on, is the driving force of the original project, rapid investigation of violations of conditions of use. Right now, we are sometimes faced with questions like ‘‘when and where has this user signed on’’. Given the hundreds of machines, this is non trivial to go out and collect; with the database, this will be trivial. There are other related questions that come up, and while I would rather not deal with them, we don’t seem to have much choice.

One popular feature of our old mainframe system, was the *TermIdle* screen. This enabled people to see which terminal rooms had available machines. This was possible since every terminal was hardwired directly to the mainframe. It has been a long time since we have been able to provide this, but this project may provide the means to do it. This would require the *wtmp_load* running in sleep mode with a short check interval. In addition to the **WTMP_LOG** table, it could also update a much smaller table, with one entry per host with the current state of the machine. We might also want to keep the time of the last successful session, as a complaint of the old *TermIdle* system, is that during crunch times, it gave the number of broken terminals in each room.

Another area to explore, is using these tools to track data other than Unix wtmp files. Any session based logs, such as the ones from our terminal servers, or resource pool logs from the campus phone switch could be able to be loaded into Oracle, and at that point the existing tools should be able to do all the same analysis on these records, as it can do on wtmp records.

Related Projects

For the past four years, RPI has been developing a suite of tools called Simon, to assist in the management of UNIX systems. Some parts of the Simon project were used in this project. Those, and other additional things are described below.

Host Name Database

The Simon Hostmaster[2] project is a set of programs and oracle tables that assists in the maintenance and generation of the resource record files for named, and host table files. For this project, we are just interested in the host naming part which is done with the **Dns_Domains** table. The column of interest here are:

domain_id *number* A unique identifier for the given domain/node. This identifies this particular entry on the DNS tree. This will be drawn from the simon.peoplecount sequence

name *char(64)* The simple text for this node name. It is an unqualified string with no ‘‘.’’s in it.

parent_id *number* The domain_id of the parent to this node. Given a node, you can build a name backwards by searching for the parents. Alternately, given a node, you can find all children.

Each node has a parent, with the root node having a domain id of 0. Consider the **Dns_Domains** records in Figure 6.

Name	Domain Id	Parent Id
edu	245	0
rpi	246	245
its	302	246
cs	442	246
jon	752	302

Figure 6: Dns_Domain Table Excerpt

The host *jon.its.rpi.edu* has the **Domain_Id** of 752. Given a host id (domain_id), we can run links backwards to build up the fully qualified hostname. This structure allows us to have as many domains as we want, and go as deep as we need to.

Host Groups

The host groups proved very useful in selecting hosts. They are described in more detail in another LISA paper[3].

Jgraph

For the graphical output, the jgraph[1] program written by Jim Plank at Princeton has proven quite useful and quite a time saver. If you need to generate graphs from a program, consider this package.

Availability

All the source code for the printmaster suite of programs, as well as table definitions, source code and additional reference material for the Hostmaster and host group modules are available for anonymous FTP from ftp.rpi.edu. See the file /pub/its-release/Simon.Info for details on where to find everything. Some papers and presentations that discuss other parts of the Simon project are available in /pub/its-papers.

If you have AFS available, you can browse many parts of the Simon tree. Look in */afs/rpi.edu/campus/rpi/simon/logging* for this project, and */afs/rpi.edu/campus/rpi/{sql,sandbox,netjack}* for other related parts. The anonymous ftp tree is also available in */afs/rpi.edu/campus/rpi/anon-ftp/1.0/common*.

If you just want to poke around, some information is available via

http://www.rpi.edu/~finkej/Simon.html

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 3 years. Reach him via US-Mail at RPI; VCC 315; 1108th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu.

References

[1] Plank, James S. ''Jgraph – A Filter for Plotting Graphs in PostScript'' *Proc* Winter 93 Usenix, 1993.

[2] Finke, J. ''Simon System Management: Hostmaster and Beyond'' *Proc. Community Workshop '93*, Simon Fraser University, June, 1993.

[3] Finke, J. ''Automating Printing Configuration'', *Proc.*, USENIX LISA VIII 1994.