VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

A Distributed Service Based on Migrating Servers

Andy Bond and John H. Hine

Technical Report CS-TR-91/4
October 1991

Abstract

DRUMS is a distributed system used to collect and distribute information about the performance of the individual systems making up the components of a modern network computing facility. DRUMS is designed to achieve a very high level of availability, to have minimum impact on the host systems, to adjust its processing capacity in response to changing demand, and to be easy to manage. Replication is used for both availability and performance. This paper discusses the design and structure of DRUMS and presents results based on our experience with its use.

Author Information

Andy Bond is currently a PhD student at Victoria University of Wellington. In 1987, he completed an Honours degree in Computer Science at Victoria and has since been researching task allocation and resource management in distributed systems. His research interests include distributed and multi-computer systems.

John Hine is Professor of Computer Science at Victoria University. His current research is centered on the measurement and management of resource use in distributed systems. He is also involved in the development of educational and research networks in the Asia and Pacific region.

1 Introduction

Many modern computer installations are built up from a network of heterogeneous systems. Typically, all the systems on the network are available to all the users. Software systems such as Sun's NFS have gone someway towards producing a "distributed system" from disjoint heterogeneous systems. For example, users may login to each system with an identical userid and password and they can expect to see the same file system structure on all systems.

However, in many ways these installations still operate as a collection of distinct systems. Perhaps the best example is in the allocation of resources to tasks. The user chooses a system on which to initiate the task. Once started the task is allocated resources such as memory and processor time, based on traditional algorithms used in timesharing systems.

Our research is focused on the problems of allocation of resources to processes and the measurement of resource usage in such environments. In our experience [1] and others [6] most distributed environments have excess capacity at any point in time. The problem is deciding how to best make use of it. This will require more sophisticated scheduling than has been used in the traditional timesharing system. However, we can use and justify more expensive resource allocation algorithms if the algorithms themselves make use of this spare capacity and do not interfere with the "constructive" work of the system.

As a first step we have built DRUMS, a distributed performance measurement and database system. DRUMS is designed as two sets of distributed, cooperating processes. One set is responsible for the performance measurement function, the second is responsible for maintaining performance information and supplying it on request. Both sets of processes have been designed with similar philosophies.

Replication is used to ensure a highly available service despite the partial failure of system components.

Mobility of the individual server processes is used to search out and utilise lightly loaded systems to support DRUMS.

Responsiveness to the current offered load is maintained by the dynamic creation and destruction of additional servers.

DRUMS has been successfully used to support an intermediate level scheduler, `any`, that is capable of initiating processes on workstations that best match a vector of resource requirements. Both DRUMS and `any` have been in use at VUW since December, 1989.

This article describes the design and performance of both DRUMS and `any`. We start with a brief overview of the operation of DRUMS and then look at its design in greater detail. The following section discusses the design of the scheduler `any`. We then look at statistics on the performance of DRUMS collected over several months of continuous operation. These statistics concentrate on the dynamic behaviour of the servers. The final section discusses the potential for further work.

2 An Overview of DRUMS

DRUMS currently runs on a network of over 30 Unix[1] workstations. The systems involved include HP-UX, MORE BSD and SunOS.

Viewed as a single service DRUMS collects performance statistics from all the host systems and stores these statistics in a database. It also responds to queries, supplying performance information to any number of clients. See Figure 1. In fact, DRUMS is implemented as three

[1] Unix is a trademark of AT&T.

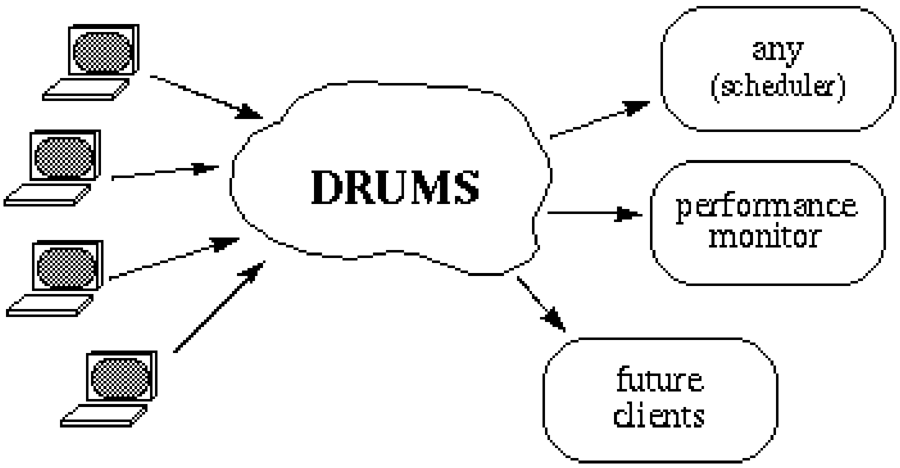

Figure 1: A High-Level View of DRUMS.

programs and each of the programs is replicated in a number of processes, as shown in Figure 2. This design provides for both seperation of function and high availability.

The three component programs of DRUMS are:

Rstat+ An enhanced `rstat` daemon runs on each host and provides performance measurements on request. `Rstat+` functions in much the same manner as `rstat`. The enhancements are described below.

Collector Each `Collector` process is responsible for periodically querying the members of a set of host systems. The size of this set is limited to ensure performance. Note that although multiple copies of the `Collector` process are running each copy is responsible for a disjoint set of hosts, i.e. each host responds to a single `Collector`. The host is said to be *registered* with that `Collector`. If one `Collector` process fails, the hosts it was responsible for must be registered with another `Collector`.

Database The `Database` processes store performance data and respond to requests from various clients. Each time a `Collector` process queries a host it broadcasts the data to all `Database` processes. In principle, each `Database` process is an exact copy of the others. In practice, there is no requirement for strict coherence and no attempt to maintain it.

Replication leads to a high level of availability. In DRUMS the `Collectors` are responsible for the availability of the `Databases` and vice versa.

`Databases` monitor the collection of data in the following manner. Each `Database` process is aware of all the hosts in the system. If a `Database` has not received data about a particular host after a period of time it assumes the host is no longer registered with a `Collector`. It attempts to remedy this by registering the host. If the host is in fact registered, but is currently not responding to requests for data the responsible `Collector` will notify the `Database`, otherwise the host is registered and data collection resumes. Registration may involve starting a new `Collector`.

In a similar fashion the `Collector` processes monitor the number of `Database` processes. Each time a `Collector` broadcasts a data sample it counts the responses it receives. If the number of responses is below a minimum level the `Collector` inititates a new `Database`

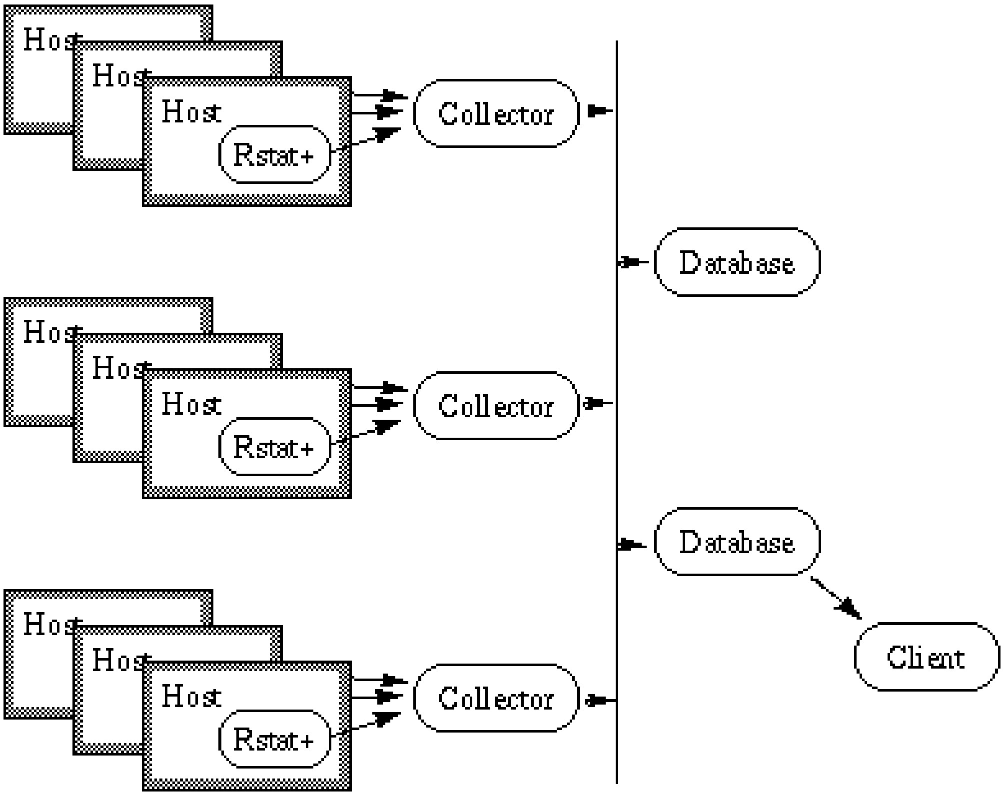

Figure 2: The Seperate Components of DRUMS.

process. Since the `Database` processes give priority to servicing queries over updating their data this control method will also initiate a new `Database` process if the current `Databases` are overloaded.

A unique aspect of DRUMS is the dynamic nature of the servers, both `Collectors` and `Databases`. The servers are not bound to particular hosts. Servers may migrate from one host to another or a new server may be created on a lightly loaded host to increase the performance of the overall service. These servers combine aspects of the *troupes* described in [5] and the *gypsy servers* from [8].

3 Design of DRUMS

This section discusses the design of DRUMS. We first look at the goals that were set for its design and then look at each of the three major components in greater detail.

3.1 Goals

While the principle goal of DRUMS was to provide a performance information service to a range of clients, we also set a number of other goals to guide our design.

- The service must be resilient to virtually all "partial interruptions" of the underlying computer systems and network. The service will be an integral part of our higher level scheduling, thus it must be available at all times. Even a medium size distributed system has frequent partial outages as individual systems fail, system software is updated, etc.

- The service must be responsive to requests for service. The service is being designed to provide information in a real-time environment. It must be able to satisfy requests in a timely fashion. Ideally the capacity of the service should adapt to changes in the offered load.

- The service should have a minimal impact on the constructive work of the host systems. Part of our overall philosophy is making use of lightly loaded hosts in a distributed environment. The service should be designed to use "spare" capacity where it is available rather than running on an otherwise busy workstation.

- DRUMS should require a minimal amount of management intervention. The management of a modest size cluster of workstations is already too much work. We would like our service to simply "be there" requiring no intervention from the system administrator when new system software is installed, new hosts are added to the network, etc.

Note that we did not set a requirement for the coherence of the data held by the `Database` servers. We are more interested in the availability of the system than in the timeliness or coherence of the state of the individual servers. For example, it is quite acceptable for one server to not have the latest reading on the CPU utilisation of host "Tinakori".

3.2 The `Rstat+` Daemon

`Rstat` is the traditional statistics server provided with Sun's RPC environment. It provides remote access to kernel statistics. We have extended this daemon to provide additional statistics, including:

- information on the current console user (including existence and idle time),

- virtual and real memory usage and availability, and
- job queue lengths.

Traditionally, `rstat` provided access to the simple count statistics kept by the kernel, *e.g.* disk transfer counts for each disk. These counts have been converted to rates, and related measures have been aggregated to provide a single uniform interface.

In order to calculate these rates, `rstat+` takes at least two measurements of the kernel statistics before responding to the first query. Additional queries received before a set time period has elapsed are answered with only one call to the kernel. Currently the pause between the initial two readings is 3 seconds. The following statistics were collected on the performance of `rstat+`. The time for the *Client* is the elapsed time for the second and subsequent calls as observed on the client including marshalling of arguments, etc. The time for the *Server* measures the elapsed time from the receipt of the request message until a response is sent.

	`rstat`	`rstat+`
Client (secs)	0.222	0.289
Server (secs)	0.148	0.195
Resident size (Kb)	124	204

The new server is marginally more expensive than the traditional implementation due to the extra information it deals with. The main overhead is the initial three second pause which is incurred while gathering rates for the first request only.

3.3 The Collectors

Each `Collector` maintains a list of hosts which have been registered with it. Each host is registered with only one `Collector`. The `Collector` is responsible for interrogating the `rstat+` daemon on each host and transmitting the data collected to all `Database` processes.

To ensure adequate performance, the number of hosts each `Collector` may take responsibility for is limited to a maximum value, M. M is determined from parameters set by a system administrator.

U is the target cycle time between updates of the statistics held in the `Databases`. The `Collector` will try to refresh each set of statistics every U seconds.

T is the timeout interval for the broadcast of host data to the `Database` servers. The `Collector` may need to wait for a period of up to T seconds to count the number of `Databases` acknowledging receipt.

W is the time in seconds between successive broadcasts of host data to individual Databases. To prevent an overlap in successive data collection operations for the same host we currently require $W > T$.

A collector can determine the maximum number of hosts it is able to handle from

$$M = \lfloor \frac{U}{W} \rfloor$$

The state diagram for the `Collectors` is shown in figure 3. The `Collectors` respond to four events. The first two events deal with the registration of hosts. The *host registered* event is triggered by a message asking if a particular host is already registered with this `Collector`. The `Collector` replies with one of three responses:

- The host is already registered with this `Collector`.

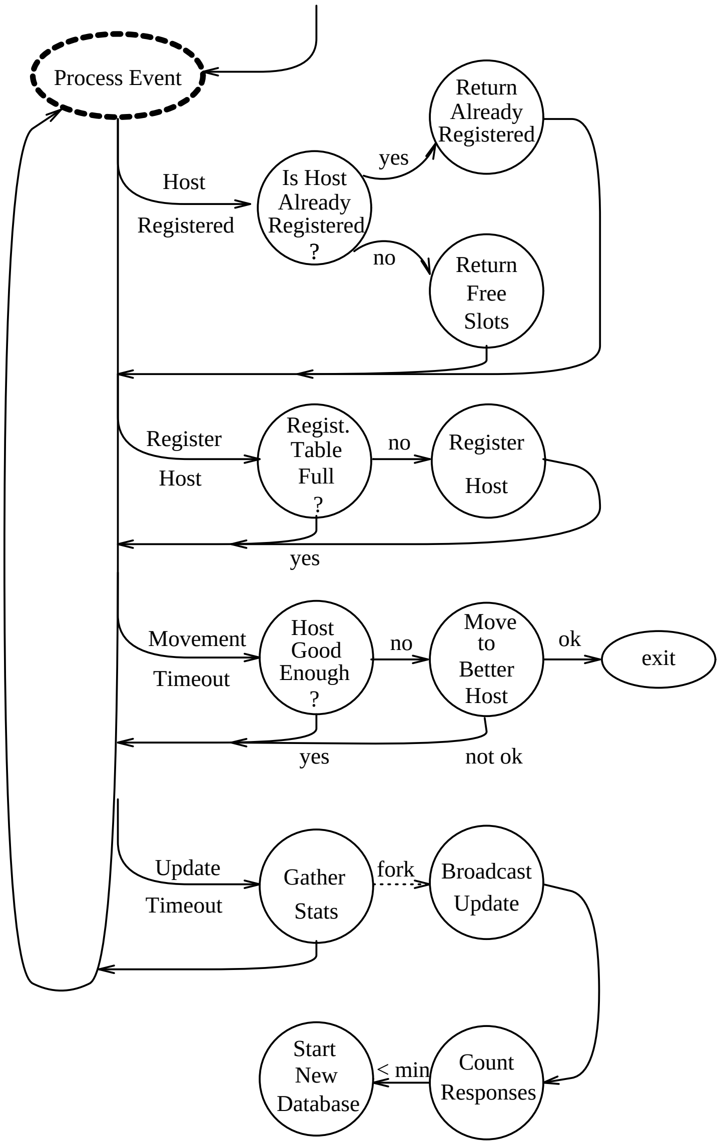

Figure 3: Collector State Diagram

- The host is not registered and this `Collector` has the capacity to take responsibility for an additional host.
- The host is not registered but, this `Collector` has no spare capacity.

The *register host* event is caused by a request to register a new host with the `Collector`.

The `Collector's` principle role is to gather statistics from the hosts registered with it. It uses an *update timeout* event to periodically poll the hosts. This event occurs every W seconds for each host. The `Collector` first makes a remote procedure call to the statistics daemon to collect the data and then spawns a subprocess to transmit the update to the `Databases`. This allows the `Collector` to overlap retrieval of data from another host with counting the acknowledgements from the `Databases`. By requiring $W > T$ we limit the number of concurrent subprocesses involved in the transmission of data.

The subprocess counts the number of responses it receives in time T and compares this with the minimum number of `Database` servers that should be running. If there are not sufficient responses the `Collector` assumes a new `Database` server is required and starts one on an appropriate host. The new `Database` may be required because an existing one has abnormally terminated or because the load on one or more Databases is such that they cannot acknowledge the new data within the timeout period.

The final event that a `Collector` may experience is the *movement timeout* event. This event causes the `Collector` to invoke the procedure `find_host_by_application`. This procedure is the heart of the `any` program mentioned earlier. The procedure is invoked with parameters that cause it to return a list of hosts that are the "best" candidates for hosting the server. This list is currently limited to the top 20% of available hosts.

The server will initiate the migration procedure if

1. its current host is not in the list of candidates, and
2. there exists a member of the candidate list that is not currently hosting a similar server.

The state held by a `Collector` consists of the set of registered hosts for which it is responsible. Rather than migrate with this state the `Collector` follows a procedure that may allow it to terminate, reducing the number of `Collectors`.

The `Collector` attempts to re-register each host it is responsible for with a different `Collector`. If it succeeds it assumes it is no longer needed and simply terminates. If it does not succeed, i.e. the remaining `Collectors` will not accept a registration, a new `Collector` is started on the "best" host from the list obtained at the start of the procedure. The remaining hosts are then re-registered with the new `Collector`. Experience over several months indicates that a new `Collector` is required about half of the time.

3.4 The Databases

Each `Database` holds performance information about every host in the system. The `Databases` are able to respond to queries of the form "From the set of hosts with at least 12 Mb of memory and a directly attached disk, which currently has the lightest cpu utilisation?" Information is periodically supplied to the `Databases` by the `Collectors`.

`Databases` respond to four events as shown in figure 4. Three of these events relate to the data held by the `Database`. The *query* event is raised in response to a query from a client and the *update request* event is raised when new data arrives from a `Collector`.

An *entry timeout* event is raised when the `Database` has not received data relating to one or more hosts for some period of time.[2] The `Database` assumes the `Collector` that was

[2] If a host is known to be unavailable the Collector responsible for it sends a message to this effect rather than data.

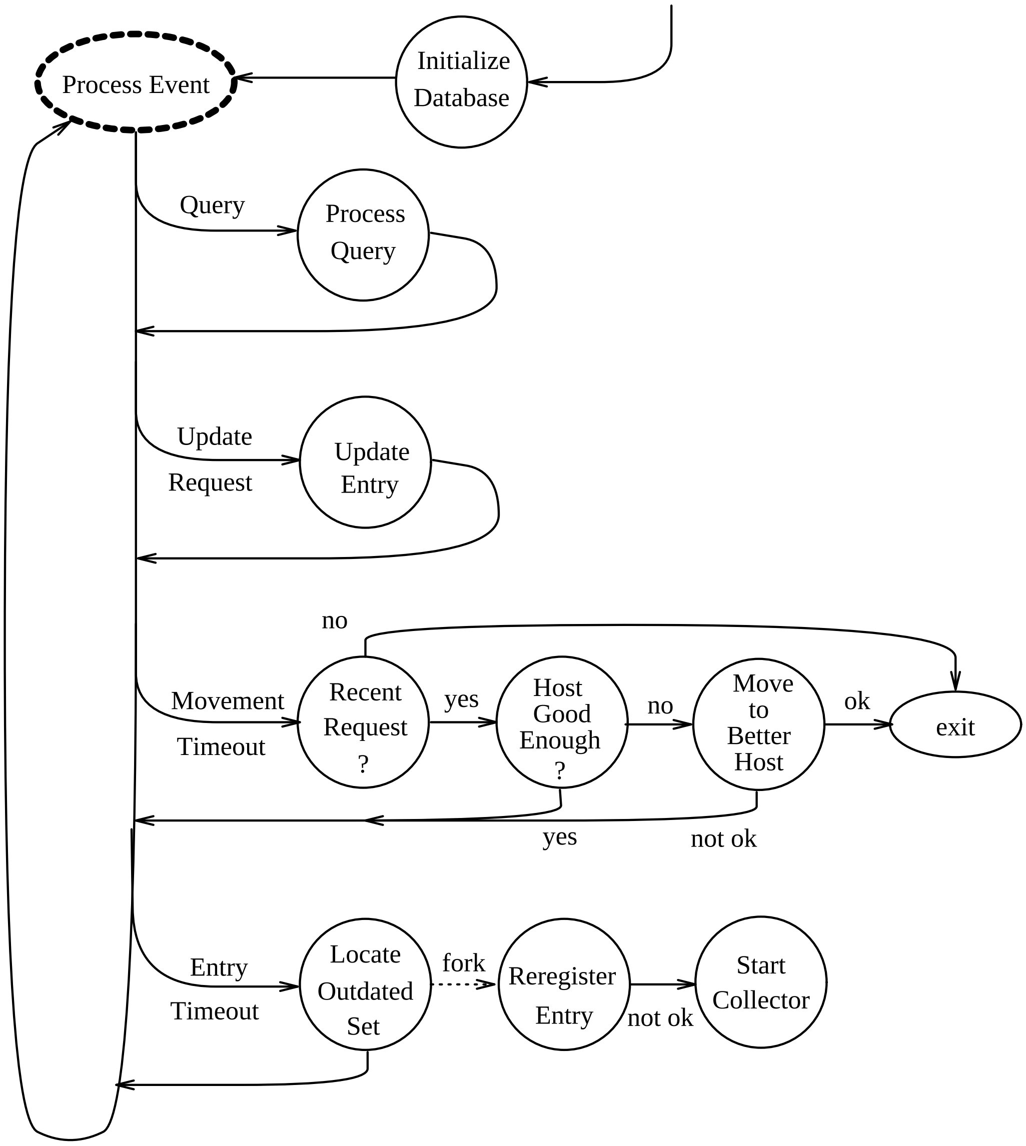

Figure 4: State transition diagram for the `Database` processes.

responsible for the host has terminated. It next queries the `Collectors` about the host. Each `Collector` will return one of the three responses described above.

Based on the replies it receives the `Database` will either continue, register the host with a `Collector` or, if necessary, start a new `Collector` and then register the host. In this way the `Databases` indirectly monitor the `Collectors` and start new `Collectors` when the number currently available appears to be inadequate.

A *movement timeout* event is raised after a fixed period of time. This is the same event that occurs in a `Collector` and is handled in the same way.

The state information held by a `Database` may either be retrieved from another `Database` or (eventually) reconstructed from updates received from the `Collectors`. Thus a `Database` may simply "forget" all it's information before terminating. Once a `Database` determines that it should migrate to a new host it starts a copy of the `Database` on that host and terminates. The new `Database` queries the other `Database` servers, to get an initial copy of the database.

4 `ANY` – Information For Task Scheduling

A significant part of our research studies the allocation of tasks to hosts based on the expected resource requirements of the task and the resources available on each host. `Any` is an application that ranks individual hosts as potential sites for the execution of a task. `Any` uses static estimations of resource requirements and dynamic information on resource availability provided by DRUMS to rank the hosts. This section discusses the design and performance of this application.

4.1 Resource Requirement Descriptions

`Any` returns a set of hostnames given a query containing a description of the resources required by a task. This description is made up of two components, a characterisation of the resources required and a set of weights used to combine individual measures into a single index.

1. **Characteristics.**

   The first part of the description acts as a mask to recognize those hosts that have the resources required by the task. We use resources in a very broad way, including the following categories:

   - *Statistical measurements.* Relations describing requirements based on statistical measurements of the host system. For example; cpu states, system load, swapping activity, memory availability and usage, etc.
   - *Hardware requirements.* Availability of specific hardware, such as a local disk or numeric processor.
   - *Architecture and system software.* Limit the selection by processor family or type (e.g. M680X0), and operating system (e.g. HP-UX).
   - *User interaction.* Currently this includes detection of an active console user on the workstation and the number of users on the machine.
   - *Hostnames and network addresses.* Restrictions can be placed on individual hostnames or networks.

   These characteristics are linked using a simple expression syntax in which the logical connectives such as AND, OR, and NOT are available. Such an expression might be:

```
((hostname == kaukau) || (os == HPBSD)) && num_users <=3
```

2. **Resource Weightings.**

   Once the characteristics have selected the eligible hosts, these hosts are ranked according to suitability for the desired task. The resulting ordered set is termed the **eligible set**. Suitability is defined using a vector of weights for one or more of the resource measurements recorded for that host. Each resource measurement is ranked against known availability and the weighted sum is returned as a rank between 0 and 100, where 100 represents the most suitable host given the previous history of resource measurements.

   These weightings are specified in the following format:

   ```
   max cpu_idle 0.3 , min load_5 0.4 , min num_users 0.3
   ```

The **eligble set** provides an ordered set of hosts which meet the requirements of the task. `Any` allows the user to specify some additional requirements to specify further quality constraints on the list returned and to specify the size of the list.

- *Quality.* This option specifies what portion of the highest ranked hosts in the **eligible set** to use for selection. This can either be a specific number, a percentage, or the entire set of highest ranked hostnames. We call this the **quality set**.
- *Quantity.* From the **quality set** we now select the hosts to be returned. This may be a subset or the complete set. If a subset, we select a random subset of the **quality set**. We now have the final result, the **quantity set**.

This two pronged approach allows us to specify a selection such as "the best host" by limiting the **quality set** to the highest ranked member of the **eligible set**. We can also select any host meeting our requirements by imposing no quality constraint and selecting one member of the resulting set.

As an example,

```
% any -l80% -g10 -r -s "(hostname == kaukau || os == HPBSD) && -
  num_users <=3" "max cpu_idle 0.3 , min load_5 0.4 , min -
  num_users 0.3"
kaukau.comp.vuw.ac.nz 86
embassy.comp.vuw.ac.nz 70
crawford.comp.vuw.ac.nz 46
cantina.comp.vuw.ac.nz 46
otari.comp.vuw.ac.nz 45
moa-pt.comp.vuw.ac.nz 45
horokiwi.comp.vuw.ac.nz 45
aro-valley.comp.vuw.ac.nz 44
offramp.comp.vuw.ac.nz 43
mt-albert.comp.vuw.ac.nz 43
```

where the – at end of lines are continuation characters. The **eligible set** in this example will consist of all hosts running the HPBSD operating system plus the host "kaukau", that have three or fewer users. The set will be ordered using cpu idle time, the five minute load average and the number of users as criteria. The quality constraint selects the top 80% of the **eligible set**. Finally ten members of this set are chosen at random, sorted and returned along with their ranking.

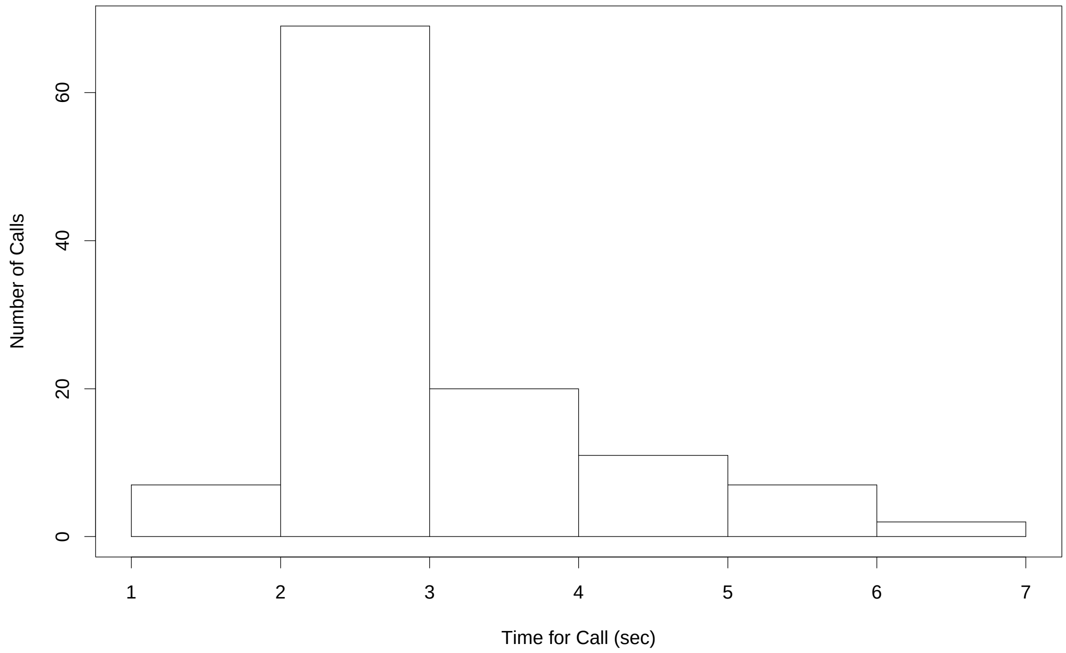

Figure 5: A histogram of `any` call times (real time in seconds).

4.2 Experience

`Any` has been in use throughout the department for over a year. It is especially useful for allocating compilers, desktop publishing applications, and other "heavy" resource usage applications. In the past 11 weeks, there have been over 5000 requests, averaging just under three requests per hour. Of the requests to DRUMS, this makes up approximately 20% with the remainder being generated by the DRUMS servers.

The average elapsed execution time for `any` is 3 seconds. Figure 5 shows the distribution of the calling times over a 24 hour period. `Any` executes by invoking the procedure `find_host_by_application` also used by the `Collectors` and `Databases` when they need to migrate. This procedure currently uses a three–tiered caching scheme to locate database servers. First choice is a cache in a local variable, second a file in the /tmp directory, and the final resort is a broadcast to all databases. Figure 6 shows that almost 70% of queries succeed with the process cache and only 7% require a broadcast.

The execution time of this host selection mechanism has significant room for improvement. To improve usability, an alias mechanism was introduced to name commonly used requirement descriptions. Both local and system wide alias files are searched on each `any` invocation. In a diskless workstation environment, these extra file accesses can have a major influence on execution times. Another area for improvement is caching database locations. Since each invocation of `any` is a new process, the first successful cache location is usually the file cache, requiring yet another file access. Once the database server has been found, the retrieval of ranked host information takes approximately 0.8 seconds. It is clear that the inclusion of this mechanism as part of a shell or operating system would significantly improve performance.

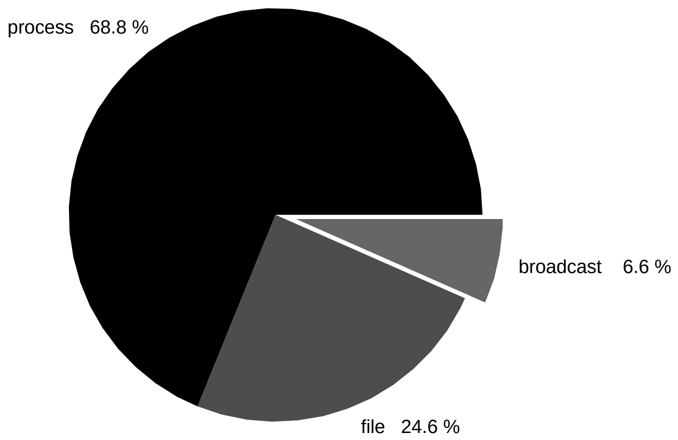

Figure 6: Cache hit percentages.

5 Performance of DRUMS

In this section we will look at the performance of DRUMS with respect to two general design issues:

Availability The service was designed to be continuously available despite partial failure of the host system.

Migration The servers were designed to be dynamic. They migrate in response to the loads on hosts and adjust their number in response to demand.

5.1 Availability

DRUMS has been in continual use within our Department for more than a year. The data presented in this section was collected over an eleven week period. During this time individual workstations and file servers have been rebooted at random times, on one night new system software was installed on three quarters of the systems and on another night a new automount facility was installed.

Figure 7 indicates the number of unexpected terminations of DRUMS' servers in each 12 hour interval over the eleven week period. From the data we can determine that the MTBF of the Collectors is 38.7 hours and the Databases is 6.9 hours. The much lower database MTBF being due to both extra server development (and the subsequent unexpected failures) as well as partial failures in the computing environment. This in itself gives an indication of the environment distributed applications such as DRUMS run in.

Availability in an environment fraught with unexpectedly terminating servers is difficult to maintain. To combat sudden server loss, it is important to ensure adequate numbers of servers survive at any given time. Figure 8 shows the frequency distributions of the server

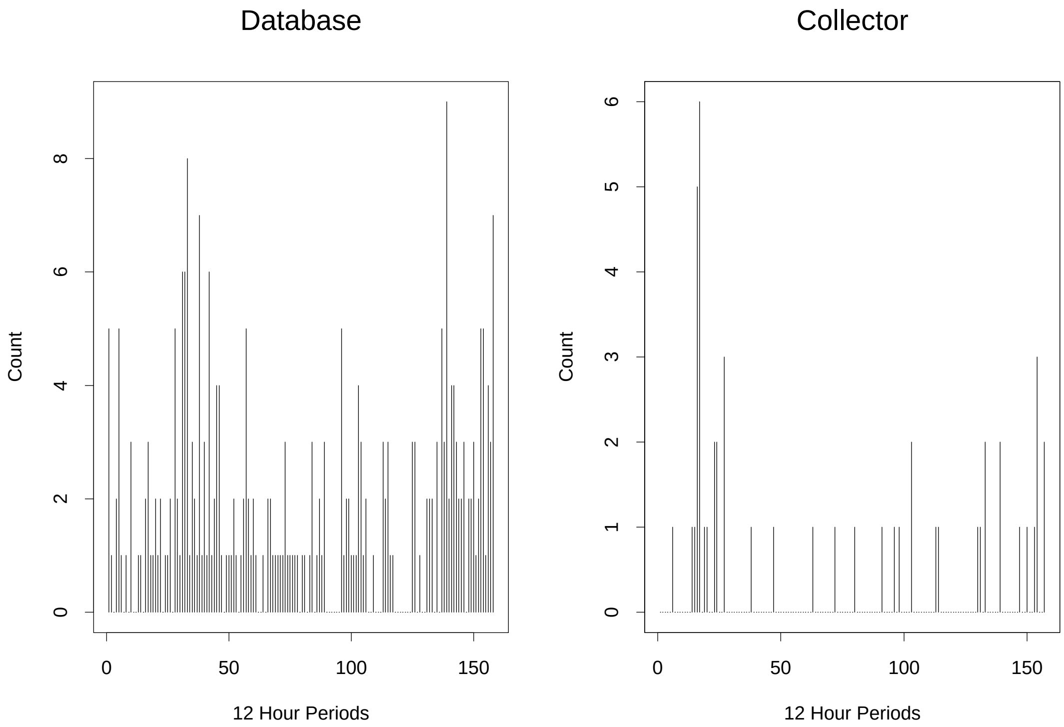

Figure 7: The number of abnormal deaths of servers in each 12 hour interval over a period of 11 weeks.

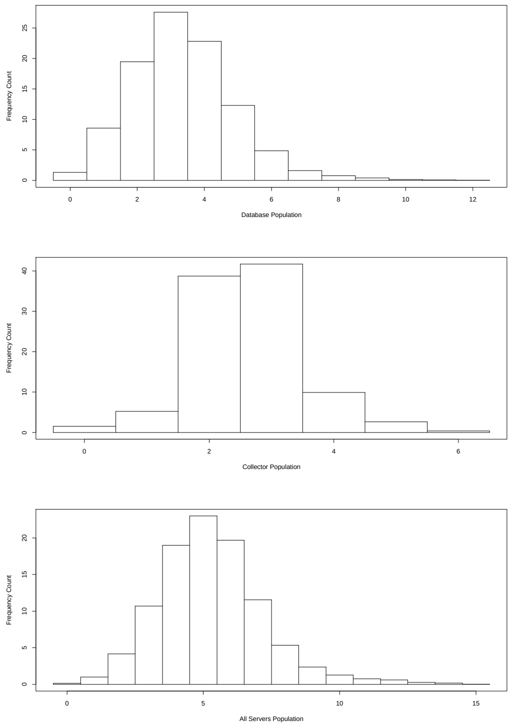

Figure 8: Frequency distributions of simultaneous server numbers a eleven week period

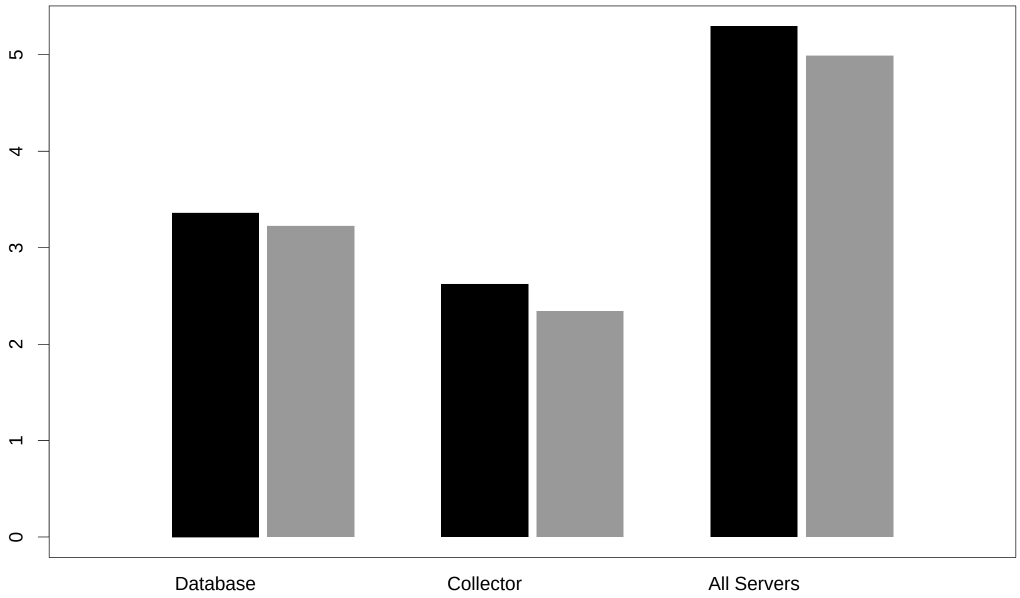

Figure 9: Average numbers of simultaneous servers over an eleven week period.

populations. On average, there were 3.3 database servers and 2.6 collector servers over the eleven week experimental period (see figure 9). The database server population is more volatile than that of the collectors and so has a wider distribution. This volitility is due to the combination of several factors:

- code development and the subsequent unexpected failures,
- idle server termination after one hour, and
- irregular database request load.

5.2 Migration

The migration of servers to lightly loaded hosts is a unique feature of DRUMS. In this section we look at how frequently the servers move, the distribution of their residence times between moves, and the effect that system load has on movement frequency and database populations.

5.2.1 Residence Times

Figures 10 and 11 show the cumulative distribution functions for the time spent on a single host between movements. Figure 10 includes the entire sample period. Occasionally a server will reside on one host for several days but, most often, it is only for a period of hours. Over half the collectors spend more than an hour on a host while 80% of the databases spend exactly an hour on a host. This dramatic cutoff is due to underutilized databases being purged after an hour. The current purge time limit creates an unstable database population (see figure 7) and doesn't take into account the redundancy that these databases provide.

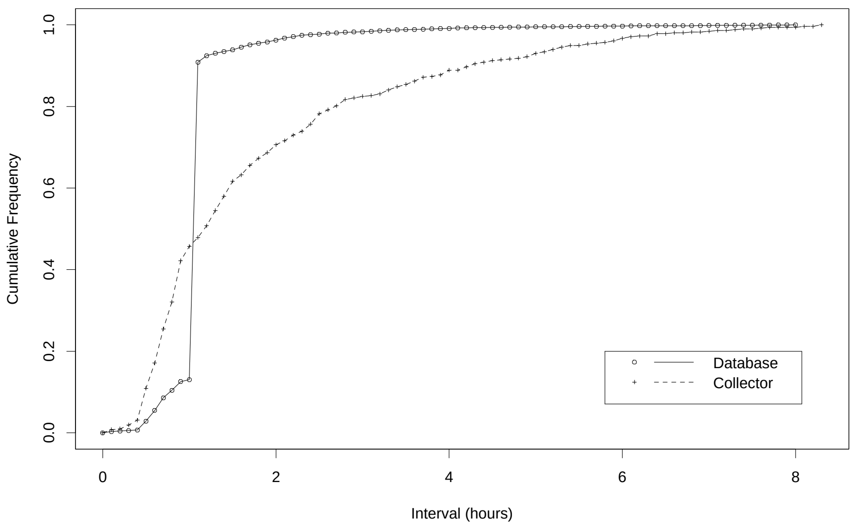

Figure 10: Cumulative frequency distribution of server residence times.

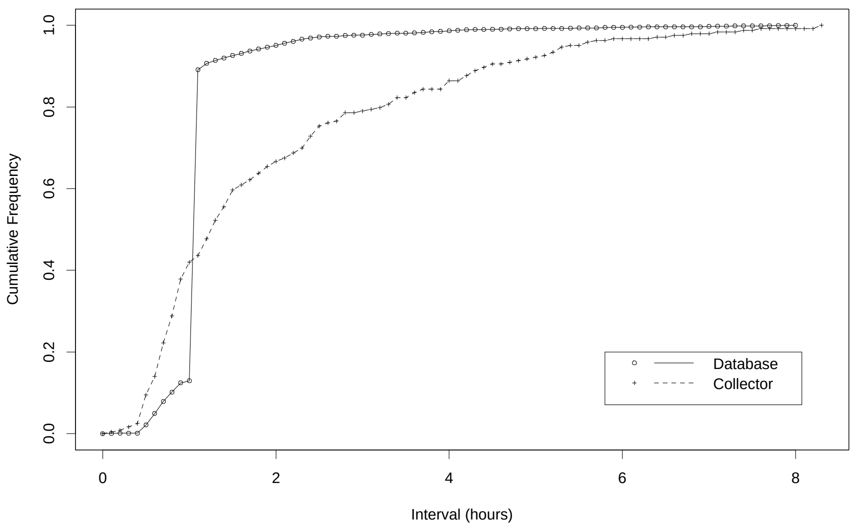

Figure 11: Cumulative frequency distribution of server residence times limited to 9am through 5pm.

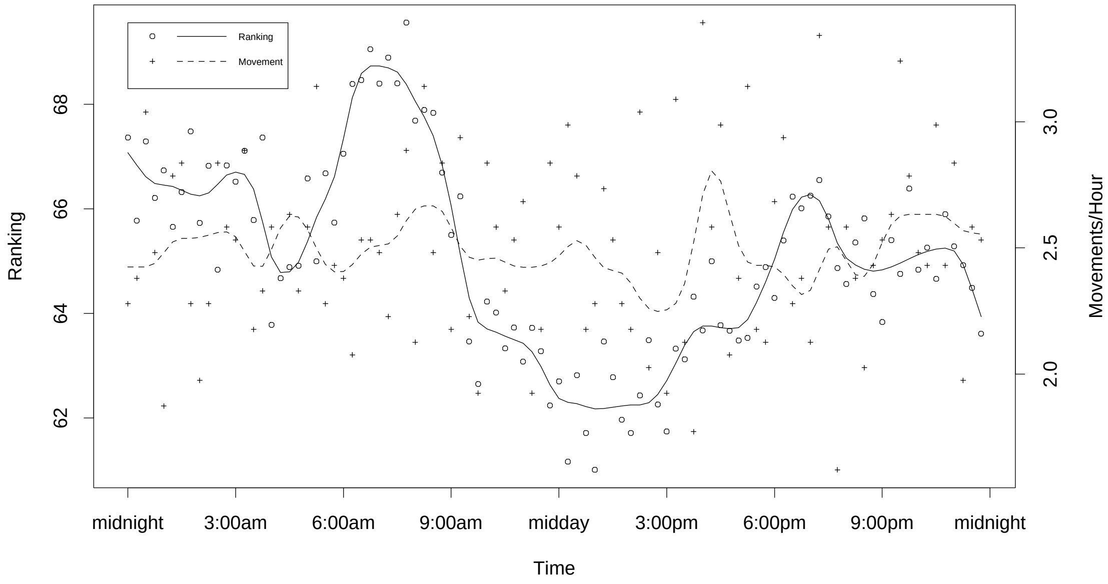

Figure 12: Server movement rates and average host rankings by load.

This purge limit must be increased to balance the redundancy benefits (and thus, reliability) with the load created by a large database population.

Figure 11 looks at just the eight hour periods from 9am to 5pm when our system is most heavily loaded. Only residence periods that lie entirely within the 9am to 5pm period are included. At first, it is surprising that the two residence period distributions are virtually identical. However, we must keep in mind that approximately 75% of the experimental period was during the vacation period, thus very rarely were a large number of workstations being used. The spare pool of workstations (as large as 30% even during the academic year) would provide plenty of capacity for the servers to use during the expected busy working day. An additional factor is the small request load currently being placed on the service.

5.2.2 Movement Rates and the System Load

Finally, we examine the expected relationship between the overall load on the system and the migration of the servers. Figure 12 presents two sets of data; the distribution of the host rankings on the system over a 24 hour day as seen by the servers, and server movements over 24 hours. The data points represent the averages of recorded values in 15 minute intervals over an eleven week period. Recall that a server invokes the function `find_host_by_application` to obtain a list of possible new hosts. This function computes a single number, the ranking, for each host indicating its suitability as a host for the server. The server movement rates per hour are for both database and collector servers.

The ranking measurement is of a static resource vector used for database host selection (see above). Simply, a higher ranking average indicates that the networked hosts are less busy. For example, we see lower rankings during the working hours and another dip at 4:00am when a security package is run on all workstations.

The movement data is interesting in that no obvious trend stands out. In fact the values have a very small but very random movement rate range. The smoothed curve fitted to the data doesn't highlight the randomness of the data. Given such a distribution, it seems unlikely that it will correlate well with another distribution. Several reasons contribute to this distribution:

- An unstable server population caused by the low underutilized database cutoff time limit.
- A large idle workstation population means movement caused by workload changes are minimal. Most movements are being caused by idle database terminations.
- As the time for a database to process requests increases, the probability of an existing database being busy increases. This causes more databases to be created, increasing the effects an unstable database population has on movement rates.

The combined graph of rank and movement was designed to demonstrate the hypothesis that a rise in the average host rank would be reflected in increased movement activity. Correspondingly, a lowering of the average rank should limit the available hosts, thus minimizing movement. Figure 12 shows that such a simple relationship doesn't exist. The only trend, and very weak at that, seems to be a decline in movement rates as the rank decreases.

6 Further Work

We have presented the design of a service based on migrating servers and a number of performance measures to indicate its behaviour. Our design, with one set of processes looking after another and vice versa, has proved highly reliable. It has solved the problem of availability of gypsy servers mentioned in [7]. There is another interesting problem with migrating servers that requires attention.

It is typical for a client process to locate a service by either communicating with a well known address or querying a name server. Neither of these work without modification when the servers migrate. One obvious possibility is to use group communication as seen in the V-Kernel [4] and VMTP [2]. An interesting question is whether muticasting would be more efficient than querying a name server which cached an address which may be out of date. Remember the median residence time of our servers is over an hour, even during the day.

A potential area of application is the class of servers whose databases are built up through a caching mechanism. Consider for instance a global name service such as that proposed by Cheriton and Mann in [3]. There are requirements for replicated servers for both availability and performance, and for additional servers as the load increases. There is no requirement for strict coherence among the caches.

References

[1] Andy Bond and John H. Hine. Drums: A distributed statistical server for stars. In *Winter USENIX 1991*, Dallas, January 1991. UNIFORUM.

[2] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Communication Architecture and Protocols, SIGCOMM 86 Symposium*, Stowe, Vt., August 1986. ACM.

[3] David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[4] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernal. *ACM Transactions on Computer Systems*, 3(2):77–107, 1985.

[5] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78. ACM, December 1985.

[6] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE, June 1988.

[7] D. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, Austin, TX, November 1987. ACM.

[8] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 5–12. Association of Computing Machinery, 1987.