Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

Ningaui: A Linux Cluster for Business

Andrew Hume

AT&T Labs - Research
andrew@research.att.com

Scott Daniels

Electronic Data Systems Corporation
scott.daniels@eds.com

Abstract

Clusters of commodity PC hardware are very attractive as a basis for large scale computing. In fact, this style of system, commonly referred to as Beowulf systems, are well on their way to dominating the supercomputing arena, which is almost solely concerned with large scientific computing. In other domains, issues other than performance predominate.

This paper describes Ningaui, an architecture for computing on partitioned datasets using a cluster of loosely coupled computers. Although performance is a goal, it is dominated by the goals of availability, reliability, scalability and manageability. We describe the architecture, together with a large-scale application, and describe how we met our goals. We also discuss how well our platform fared, and the lessons we have learned. As our environment is jargon rich, we include a glossary.

1. Introduction

The relentless pressure of Moore's Law, or more accurately, the surprisingly continuous exponential growth in CPU speed and disk capacity, enables, and ultimately forces, the development of systems architectures to handle problems of ever increasing size. No where is this trend more obvious than in scientific supercomputing where the fastest systems, traditionally large expensive monolithic machines from Cray, Fujitsu and the like, will soon be vast arrays of PCs linked in architectures popularised as Beowulf systems[Bec95].

The adoption of this kind of architecture for business systems has been rather less enthusiastically pursued. Despite the promise of cheaper hardware and system software costs, issues of reliability, availability and the necessity to run common commercial software packages have dominated and largely blocked the introduction of these architectures. In particular, the commonplace property of scientific Beowulf systems that a computation fails completely if any node fails during the computation, while apparently tolerable for the scientific community, is *not* viable for business systems.

Over the last year, we have been in a position to investigate these issues in the context of a large production system within AT&T, namely Gecko[Hum00]. Gecko traced the processing of telephone call and billing records for residential AT&T customers on a continuous and ongoing basis. In 2001, we were asked to resurrect Gecko, with the change of focus to analysing business (rather than residential) customers and having rather less funding for the project. We took the liberty of interpreting this request as a desire to port the Gecko software to a cluster of PCs running Linux (or somesuch) and adding some requirements of our own:

- the system must be highly available and reliable, requiring only 8×5 support (and not 24×7), even if hardware fails.
- the system must be highly scalable. That is, we can predict with confidence how much CPU and disk resources are needed for a desired workload, and that the resources are roughly linear in the size of the workload.
- running the system, which has two major components (managing the hardware and

managing the feeds coming into the system), must be a simple, low effort activity.

The resulting system, which we call Ningaui (a small Australian mouse-like marsupial), is based on a cluster of hefty PCs running Linux, connected by a high speed network. In the following sections, we briefly describe the business problem and its system requirements. We then describe the Ningaui architecture and how it meets its goals, and the current implementation, both what it is, and how well it has met our expectations. Finally, we'll do some comparisons with Gecko, and discuss future plans.

2. The Problem

2.1. The Business Problem

The flow of records through AT&T's billing systems, from recording to settlement, is fairly complicated. For most business calls, the records flow through seven major billers comprising 15-20 major systems and are processed by a few hundred different processing steps. Complexity arises not only from that inherent in the work (e.g., how to handle records sent to the wrong biller), but also from the tendency to implement new features by tacking on new systems, rather than reworking and integrating the new features into the existing systems. This flow undergoes considerable churn both at the process level, and at the architectural level. The key business problem is: in the face of this complexity and change, how do we know that every call is billed exactly once? (Gecko answered a similar question for residential calls, but business is harder to do for two major reasons: 3 times greater volumes, and probably 6 times as much architectural complexity. Furthermore, each of the major billers was developed and are run by distinct, independent, and competing organisations.)

Ningaui attacks this question the same way Gecko did; it tracks the progress of (or records corresponding to) each call throughout the billing process by tapping the dataflows between systems and within systems. This is a data-intensive method; for Gecko, it involved a daily average of 3100 files totaling 250GB per day. It is a measure of how far things have advanced in 5 years in that for Gecko, this was a novel idea, whereas today, it seems only a little excessive. The main difference between Gecko and Ningaui is the change in platform from a large multiprocessor system to a loosely coupled cluster.

2.2. The Technical Problem

The problem is fourfold: we need to convert the various dataflow taps into canonical fixed-length *tags* (parsing), we need to match tags for a call together into *tagsets* and maintain them in a database (update cycle), we need to generate various reports from the datastore (report generation), and we need a scheme for backups.

Mainly because tagsets have variable length and determining their reporting status requires quite complicated logic, we are unable to use conventional databases and in fact use simple sorted flat files. To keep these files, or database partitions, manageable, we split the database into about 5000 pieces, based on a hash of the originating telephone number for each call. There are several more requirements, outlined in[Hum00], but they are unimportant here, other than we need to do the daily update cycle as fast as we can.

3. The Current Architecture

At a basic level, the fundamental task of the cluster is to support data storage and management, and to run jobs on the data, and to do so in a scalable, available way. We made some initial structural decisions:

• the cluster is a loosely coupled federation of nodes, and even though there was a leader node for coordination reasons, the control paradigm is that nodes announce resources or ask for work from the leader, and the leader never unilaterally imposes work.

• all data is stored locally at each node; there is no networked storage (like SAN) and certainly no network file system. We distrust both in terms of performance modelling and reliability.

• all activities, such as job executions, and data storage, have leases. All cluster activities and facilities must support nodes going down and being added with fairly minimal impact.

• we assume the presence of a very fast and scalable networking fabric.

The cluster infrastructure has two fundamental concerns, data storage and job execution, and support for high availability; these three issues are described below.

3.1. Data Storage

There are two kinds of cluster data. The first is various configuration files and the like; these are distributed from the primary source machine via mechanisms equivalent to the *rsync* program[Tri96]. The second are all the various application data files, such as feed files, tags to be added to the database, and the database partitions themselves.

These application data files are managed by the *replication manager*, or *repmgr*. This is a user level, file based, replication service that distributes copies amongst nodes, somewhat similar to, although substantially simpler than, other such systems such as Ficus[Pop90] and Magda[Wen01]. Although current fashion favours schemes replicating at the system call (read, write) level, we felt a user level scheme was easier to manage, and easier to diagnose when things go wrong.

Repmgr handles a single database, namely that of registered files. Files have a simple data view; they have a simple, nonhierarchical, name, MD5 checksum, length, replication count, and a callback mechanism (called when the file is correctly replicated). Files are referred to by (name,md5) tuples, and these tuples are unique across the cluster.

Each node maintains its own database of the files, or *instances*; the database relates pathnames and MD5 checksums. Periodically, each node sends *repmgr* its list of instances (of files), together with a lease time, and measures of how much space is available.

Repmgr's work is fairly simple. It takes the list of registered files, and the set of instances from all its nodes, and does the appropriate actions. These are recorded internally as attempts with leases (the copies have leases, the deletes do not); any copy that times out is retried on another node if possible. *Repmgr* logs all its registrations and periodically checkpoints (every 5 minutes); restarting takes 30-120s to rummage through the log, and another 15-20s to start running.

Repmgr also takes hints. These are not allowed to interfere with correctness, and can be used to move files around safely. For example, if a file is replicated on `ning03` and `ning12`, and we give the hint to not store a copy on `ning03`, then *repmgr* will make a copy on another (the least full) node, and only after that copy is made, will it remove the copy on `ning03`.

We are paranoid; every file copied to a node has its MD5 checksum calculated. If a file gets copied and ends up with a different checksum (for whatever reason), *repmgr* will observe a new, unrelated, instance appearing and simply try again. Cleanup scripts are run daily to clean up apparent detritus.

3.2. Job Execution

Somewhat to our surprise, we have a three level hierarchy of programs to manage and execute jobs across the cluster. It is, however, quite a robust arrangement.

The bottom level is a per system batch scheduler (*woomera*) inherited from Gecko. This is a simple and flexible engine which supports a number of constraints, such as maximum load and number of simultaneous processes, in addition to arbitrary constraints, called *resources*, which are akin to counting semaphores. Although any node can submit requests to *woomera*, all subsequent action takes place locally and independently of the cluster.

The next level in the hierarchy is *seneschal*, which implements cluster-wide job executions, handling node failures and doing some low level scheduling optimisations. *Seneschal* takes job descriptions with three important characteristics: how to invoke the program, the input files/resources needed, and the output files generated. On the basis of resources declared by nodes, *seneschal* allocates jobs to these nodes and schedules the jobs via *woomera* on those nodes. The model is that jobs are posted (on *seneschal*), and nodes bid for these jobs.

Node failure is handled by each execution having an assigned attempt number and a lease. The output filenames for that execution have the attempt number embedded in them. Upon successful execution, the output files are renamed and the job is regarded as having succeeded. If the lease expires, or the job otherwise fails, then *seneschal* reassigns it (to another node if possible), incrementing the attempt number. If multiple attempts succeed, one is chosen as the successful execution and the other(s) are treated as though they failed. This scheme relies on the fact all jobs execute locally and may only affect local, and not cluster-wide, state.

```
%BUNDLE = /ningaui/poot
%DATE = 20010728

step2a: [%p in <partitions(abs)>] "generate add file"
                <- %node_add
                -> %add = %p-342.add
                cmd ng_add_gen -i %input %p

step2b: [%p] "partition update"
                <- %add
                <- %oldpart = %p-341
                -> %newpart = %p-342
                -> %delete = %p-342.del
                -> %report = %p-342.rpt
                cmd ng_pu -d %DATE -e 342 %oldpart %add %newpart %delete %report

step3:  [%p] "report for deleted tags"
                <- %delete
                -> %delreport = %p-342.delrpt
                cmd ng_report_step1 -e 342 -d %DATE %delete %delreport
```

Figure 1: A *nawab* fragment.

Seneschal also supports a number of convenience features such as specifying a node to run a specific tasks on, and also supports *woomera*-style resources to help manage job streams. *Seneschal* runs as a single copy daemon, with no checkpointing; *nawab* feeds *seneschal* all its input and *nawab* does its own checkpoints.

The final and top level in our hierarchy is *nawab*. This is both a language, and a daemon supporting management of jobs specified by *nawab* programs. *Nawab* is a small domain specific language designed to facilitate handling large interrelated job streams; figure 1 shows a fragment of the program to perform an update cycle of the Ningaui database. A full description of the syntax is beyond this paper but the highlights are

• iterators are enclosed in `[]`; they are typically are partitions or sets of nodes. The partition set here is specific to the `abs` application.

• inputs and outputs are denoted by `<-` and `->` respectively; in addition, they can be given symbolic names usable in command strings.

• definitions of variables and iterators stay in effect until redefined

Note that *seneschal* handles the issues of sequencing and ensuring inputs are made prior to initiating jobs.

Nawab is not just a front end or compiler for *seneschal*; it also supports managing (deleting, pausing, monitoring) jobs in terms of the actual *nawab* specification. *Nawab* runs as a single copy daemon and does its own checkpointing.

3.3. High Availability

We have a simple view of, and method for, high availability. All programs fall into two classes:

• ephemeral executions should either fail or succeed. In general, temporary resource issues should be handled by failing immediately, and letting the surrounding retry software do its job.

• long running executions, such as daemons and programs like *nawab* and *repmgr*, should checkpoint periodically and log every change pertinent to restarting in the master log file. Both the checkpoints, and as we described below, the log messages, are replicated to all nodes, not only for increased safety, but also so facilitate quicker restart when we change leader. Furthermore, these programs should be embedded within a startup script that execute these programs in a loop, so that there is automatic restart on a failure.

In order to support this model, we take a lot of care over logging. Many programs log into their own logfiles, but all programs log into the master logfile. Logging to the master file is done via both of two methods:

• the log message is written directly to the master log on the current node

• the log message is broadcast (via UDP) to all nodes in the cluster. The logging routine returns only after acknowledgements have been received from a sufficient number of nodes. If insufficient acknowledgements have arrived before a timeout, the message is written to a local nak logfile. The logging daemon writes messages to both the master logfile and to a secondary logfile.

Periodically, typically every 5 minutes, a process gathers up the secondary logfiles (and nak logfiles if any) and creates a node specific log fragment which is registered with *repmgr*. A few times a day, a second process gathers up log fragments and absorbs them into larger log fragments (one per week for the whole cluster).

Although we have tried really hard to tolerate individual failures of various programs and daemons, sometimes the system is in real trouble and requires human intervention. We do this through a two part monitoring system. The first part periodically checks for processes it knows should be running and if they are not, it drops a `critical` log message in the master log. Applications may also put such messages in the log, although mostly we would expect just `warn` and `error` messages. The second part is a single process *crit_alert* which looks for critical log messages and exits when it sees one. This process is registered with whatever monitoring software our operations support folks use (for example, in Gecko, they used BMC Patrol), and the absence of this process will generate alarms and cause people to get paged. We have auxiliary commands to extract critical log messages to quickly determine what the relevant log entries were. Critical log messages also include a code which identifies the scenario, likely cause, and normal fixup procedures.

4. The Current Implementation

4.1. Hardware

We use fairly regular hardware, except we went for the highest density of disk in our PCs. The configuration is either 1 or 2 Pentium III CPUs per 2U high box. Each system has 512MB memory per CPU, integral 100BaseT Ethernet, a 3WARE PCI disk controller, and an Emulex (previously GigaNet) CLAN network card. Apart from a floppy drive and a sprinkling of CD-ROM drives, the rest of the space in the enclosures is filled with 82GB IDE drives (currently being upgraded to 160GB drives). In order to support experimentation on the effects of CPU speed and multiprocessorness, we have 16 nodes; 4 of each combination of 1 or 2 CPUs and 933 or 866MHz processors. We shipped the CLAN network cards to the supplier (Net Express)† and they shipped us the systems ready to go with Linux and drivers installed. The dual CPU motherboards also have dual SCSI ports that we use for attaching tape drives.

The other hardware is a CLAN 5300 switch (30 ports) and a QUALSTAR 36180 tape library. The tape library is used just for rolling backups of the database. We backup the source and system images over the network to one of our local servers.

We use WTI power controllers; they were about the only controller available that could supply enough power. (Even with delayed turn on, we can only run 4 systems per 20A circuit.)

At the current time we are using RedHat 6.2 and in the middle of transitioning to RedHat 7.2. Although our intent was to support a heterogeneous set of operating systems, originally, there was only CLAN support for Linux. (Of course, now the best driver is that for FreeBSD!)

We make use of the AST distribution[Fow00] which apart from providing us very useful functionality, also insulates us even more from the underlying system. (The AST distribution includes both a sophisticated set of libraries and POSIX compliant versions of the main user level commands. Perhaps most importantly, the four main authors of AST have their offices within 30 yards of ours, and we have their home phone numbers!) We anticipate very few changes should we move to another OS, such as FreeBSD.

All the knowledge and use of the fast networking is localised to one command *ccp* (cluster cp).

4.2. Software

The rest of the implementation, including the programs *seneschal*, *nawab*, and *repmgr*, work as described above. The rest of the application specific software is largely ported from Gecko and is uniteresting for the purposes of this paper.

†Net Express (`http://www.tdl.com/˜netex/`) offered to install the cards for us and we're glad we did, because the cards were larger than we expected and Net Express had to change the boxes to make things fit.

5. The Results So Far

This is a work in progress; we are just ramping up into production and we have only just started finetuning things. Nevertheless, we've learned quite a bit already.

5.1. Hardware

We are generally quite happy with the hardware. It is amazing to have one rack contain 16 nodes totaling 24 CPUs and 96 82GB drives, 2 power controllers, an Ethernet switch, and a CLAN switch. While the density is great, we are a little concerned about heat (despite assurances to the contrary); we have seen a few flaky components. We are often asked why we have so much disk per node, and shouldn't we be using RAID. In retrospect, this was a great decision. For our sort of applications, a CPU per 250-500GB of disk is a good ratio; significantly different ratios would yield systems either CPU or I/O constrained. And having this in a single box simplifies scalability calculations; if you buy enough boxes to get the disk you need, you automatically get enough CPU.

Processor (year)	kr/s	/SPARC
250MHz UltraSPARC (97)	56.7	1.00
197MHz R10000 (98)	72.8	1.28
866MHz Pentium III (01)	97.7	1.72
933MHz Pentium III (01)	102	1.80
250MHz R10000 (02)	104	1.81

Figure 2: Relative performance for CPU intensive jobs (parsing) in kilorecords per second.

While we are still investigating ways to increase effective file I/O performance, we are delighted with the CPU performance. Figure 2 shows the performance relative to the two other systems we have measured it on. Please keep in mind that the Sun and SGI systems are 3-4 years old; undoubtably, their current offerings are faster. For our cluster of 16 nodes and 24 CPUs, the overall throughput was about 1.33 that of the 32 CPU Sun E10000 that Gecko ran on. On the other hand, the cost of the Sun and disk was about 20 times the cost of the Ningaui cluster. All hail Moore's Law and the economics of commodity hardware!

Another goal of our setup was to assess the importance of minor differences in clock speed in a production environment (as opposed to one of the standard benchmarks). Figure 3 shows the effective speed of the system for each of the four cases (1 or 2 CPUs, 866 or 933MHz); the speeds are normalised per CPU. It is rewarding to see we get better speeds from faster CPUs, although the increase is not as great as the increase in clock rate. We expect the modest difference in performance between the single and double CPU case is due to the different motherboard and memory controllers.

CPU	1 CPU	2 CPU
866MHz Pentium III	97.7	98.4
933MHz Pentium III	102	105

Figure 3: How CPU speed and configuration affect throughput in kilorecords per second.

One area where we still need refinement is how to arrange the disk space. The 3WARE controller is a pseudo-RAID controller, that is, it does just striping and mirroring. Originally, we configured it to present each disk as a single LUN, as in our experience, this leads to the most efficient use of the most precious resource (disk heads). However, two problems are forcing us to change this. The first is bizarrely large tap data files; the largest one we have seen so far is 140GB. While we deal with these files in a compressed form (around 10-20GB), it would be nice to have a filesystem where we can store one of these uncompressed. The second is lack of disk speed, which you would normally attack by striping. However, the effects of striping are obscured by the mandatory Linux buffer cache. We're still working this issue, but frankly, this seems a significant limitation for Linux.

5.2. Networking

As we described above, we use the 100BaseT Ethernet for control messages and UDP broadcasts of log messages. This network is comfortably loaded; the average utilisation is under 0.1%, and we've never lost a UDP packet in several months of operation.

Essentially all our data traffic moves over the CLAN network. With the version 1 drivers (on a 2.2.17 kernel), the CLAN behaved itself admirably, being both fast and reliable. Without tuning, we measured 80-100MB/s node-to-node (memory to memory), and often 25-35MB/s. Figure 4 shows the distribution of observed speeds of about 182,000 transfers. Basically, this is a function of how much buffer cache is available. If there is little to none at both ends, you get 4-12MB/s; if there is little to none at just one end, you get 20-30MB/s, and if there is plenty of buffer cache at both ends, and the source file is in the buffer cache, you get 50-60MB/s. The fabric is circuit-switched and we never observed traffic to any node affecting the transfer speed among other nodes.

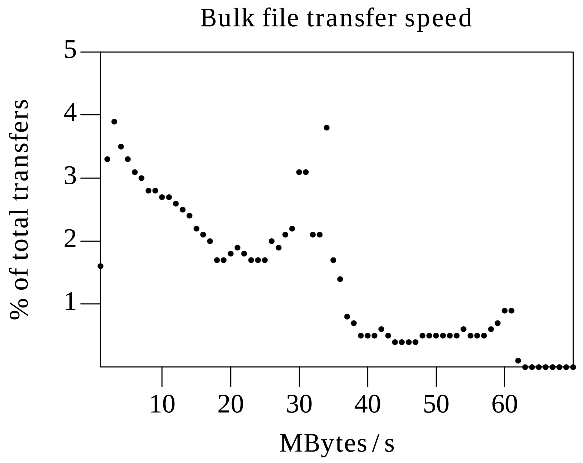

Figure 4: Effective CLAN performance for files larger than 50MB.

Unhappily, we were forced to migrate to the version 2 drivers (we needed to go to a 2.4 kernel for large files and the version 1 driver won't work on a 2.4 kernel). Regrettably, the version 1 and 2 drivers cannot coexist on the same switch (the drivers download microcode into ASICs on the controller card which talk directly to the switch). The version 2 drivers are not nearly as reliable; in particular, rebooting a system sometimes breaks the CLAN network on other systems (requiring them to be rebooted). And Emulex has not been very good about support, although the driver source has been released as "free software" and this is helping a fairly active user community improve on things. As far as we can tell, if you have this hardware, the most reliable (version 2) drivers are now those for FreeBSD[Mag02].

It wasn't an option for us, but if we making the decision today, we would probably go with 1000BaseT Ethernet, although that would introduce some scaling issues (not every node could be on the same Ethernet). There are other networks based on the same ideas as CLAN, such as InfiniBand†, but we know little about them.

†see `http://www.infinibandta.org` for more details.

5.3. System Software

As described above, we were forced to use Linux because of the CLAN network driver issues. We didn't view this as a problem, rather, we saw it as an opportunity to evaluate whether Linux was ready for prime time. We started with RedHat 6.2 and upgraded the kernel to 2.2.19. We also used ReiserFS for all the filesystems where we store data. It is fair to say we have been surprised at how problematic Linux has been to date (for the record, the authors have predominately used Solaris and Irix); the problems we've seen include:

- we use a scheme for distributing software that used dump/restore. A distribution of the root filesystem involved rebooting to a backup filesystem, *mkfs*ing the old root disk, restoring the distribution onto that disk and then rebooting again. Doing this by hand never failed, but doing it automatically failed about 5% of the time (for no reason that we could find).
- for a while, we couldn't generate new kernels. We needed a special Ethernet driver from Intel that only existed as a module. Apparently over time, our kernel had been growing because we came to a point where it was too big to build as a modularised kernel. (Changing to 2.2.19 fixed this.)
- part of the Linux community's doctrine is the mantra of if you don't like a bug you have, try another release of something (most often, the kernel) and see if it goes away *and* the bugs in this new release don't hurt you. This scattershot, hunt and peck method of trying new kernels until you're happy seemed, and still does seem, odd and unsatisfactory to us.
- we initially used *gdbm* for the per-node database of files. We gave up quickly; despite promises to the contrary, whatever locking it uses was ineffective for our setup. The mean time to a corrupted database was about 5 hours. File locking has been much more reliable.
- we encountered a bizarre problem with a server that did not keep up with a torrent of connections. On any other Unix-like system we've seen, this would fail in a polite way, that is, eventually, the clients would start getting connection refused or somesuch. Instead, no connections failed and the server would get the file descriptor from the *accept* system call as normal. The bad thing was that when loaded, reading the (approximately 100 bytes of) data from the client would take anywhere from 30-400 seconds. As always, the experts recommend trying another

kernel, but instead we went to multi-threading the server. (Of course, we didn't thread the server itself as our kernel didn't support core dumps of threaded programs; instead, we built a multithreaded front end that simply dumped all its requests down a pipe to the real server.)

• unlike most other Unix variants, Linux does not support direct I/O. That is, all I/O to files goes through the buffer cache. For applications like ours, file I/O falls into two camps; one needs to be buffered (logfiles, executable images, config files), the other (database partition files) should not. For the sake of performance, and not needlessly churning the buffer cache, we'd like to bypass the buffer cache for the second camp. And it is pointless to say we should have more main memory; more memory always helps, but we'll always have more disk than memory, and in our application, we tend to read and write all the data with very low cache hit rates.

• another consequence of the mandatory buffer cache is enormous variability in the execution times of commands like *df* during heavy I/O loads. In these cases, where a node has a dozen or more files coming in over the CLAN, we have seen *df* take over 30 minutes (only 6 filesystems, no NFS). This is not a huge problem, but some of our scripts cannot tolerate this variability and so we have to maintain a cached copy of the *df* output.

• when the system is heavily loaded, we have noticed that jobs of the form

```
gzip -d < file.gz | a.out > outputfile
```

have about a 2-5% chance of *gzip* complaining about a bad CRC (from a file that is known to be good). We don't care particularly, as our software essentially treats this as a soft error and retries the job again, but we suspect most people would correctly view this bug as pretty unsatisfactory.

• it turns out that ReiserFS has not been a good choice for us. (We chose it on the advice of local experts.) It is unsuited for our file size distribution (relatively few, big, files) and we get a panic in the ReiserFS code every couple of weeks. (This might be resolved by the newly usable *reiserfsck* software.) Having a logging filesystem is really great, though, for quick reboots. We will switch to the *ext3* filesystem when we finish our migration to the 2.4 kernel.

• we were severely constrained by the default (and undocumented) limits on how fast *inetd* can initiate processes. But at least we could look at the code and figure out that there was this limit and how to change it (which we did from 40 processes per min to 5000).

• we use Java for our monitoring software. The standard distributions ship with *kaffe*, yet if you ask around, *kaffe* is deprecated and has been for more than a year. It would seem more efficacious to ship something that works, rather than let the users find out for themselves. We use the IBM Java environment for Linux; it seems quite solid.

5.4. Cluster Software

5.4.1. Repmgr

The current replication manager is our throwaway prototype. It is about 3000 lines of C and 500 lines of Kornshell scripts. It works well enough that we are taking our time to design and implement the "real" one, despite going into production. As we had learned in Gecko, one of the more useful features is an explain mode, where *repmgr* explains exactly why it is doing, or not doing, every external action (copies and deletes).

While the details of the new version are peripheral to this paper, some are motivated by a particularly good idea we borrowed from the Venti storage server[Qui02]. The core idea behind Venti is that the name, or address, of a block of data is a function of its contents alone. Venti uses a cryptographic checksum (SHA1, although MD5 is equally useful) for its function. What has this to do with the replication manager? Currently, instances are stored on nodes as files whose basename is the same as the registered file. The per node database correlates each instance and its checksum. This is not too bad a scheme, but there are a couple of drawbacks: you have to be careful not to overwrite an instance of the same name, and it can take a long time to recreate the database by recalculating the checksums for all the instances on a node. The Venti idea suggests a better solution: name each instance by its MD5 checksum! There is now no problem with collisions (if the names match, so do their contents!), and rebuilding the database is now trivial.

It is worth asking the question how well does the registry of files correspond to reality? In a deep sense, the registry *is* reality — it is simply the list of registered files. The correct questions is how well does *repmgr*'s view of file instances correspond to reality? Like all databases of physical inventory, despite best intentions, the per node databases of the files on that node seem to start decaying as soon as you create the database! You might call this hubris, but instead, we run a process that corrects and makes the database and underlying filesystems consistent. This runs once a day and as a result, nearly all nodes have zero inconsistencies.

5.4.2. Nawab and Seneschal

The source for *nawab* and *seneschal* is about 10000 lines of C. The charming thing about servers bidding for work is that they are inherently good at load balancing; figure 5 shows a batch run of 700 jobs distributed over 9 nodes. As you can see, we keep all 9 nodes busy for the first 48.5 minutes and then the number of nodes drops off as we run out of jobs. Part of the reason for the sharp drop off is that *seneschal* orders job assignments by size (largest first) so as to get the best amount of parallel work.

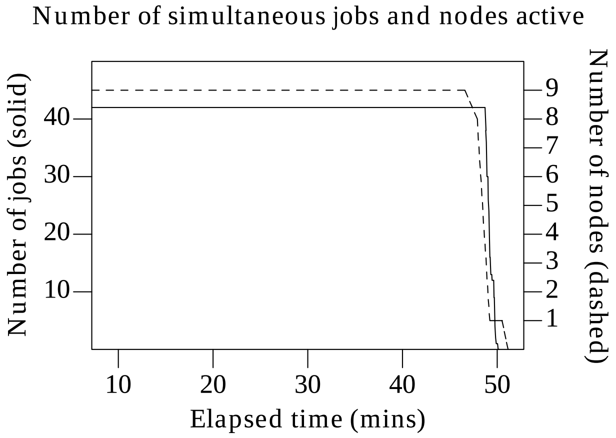

Figure 5: Effectiveness of *seneschal* scheduling.

5.4.3. Elections

Every 5 seconds, the nodes in the cluster elect a leader node using a nifty algorithm by Christof Fetzer, a variant of the algorithm in [Fet99]. (The algorithm involves each system keeping track of each system voting, their priority, and a lease on the current election. When the lease expires, we assume the leader has crashed and the system with the next highest priority will attempt to become leader. The whole election is resolved in just one round of 'voting'.) This may seem fairly frequent, but we only change something if the election yields a new leader. We added two rules to the original election scheme: re-elect the current leader unless it doesn't vote, and if changing leaders, we have specified a preferred winner (again, if that system voted). We normally specify a 2 processor system to be our normal leader (because of daemons like *seneschal* and *repmgr*).

5.4.4. Logging

Everybody knows it, but most do not do it: log everything! We expect to be generating over 120MB of master log file per day, and that is independent of daemon specific logs. And for every time we have thought we log too much, there have been multiple disasters and weirdnesses which could only be resolved by picking through the logs. Our decision to make the master log file replicated throughout the cluster has really facilitated diagnosis of multinode problems.

5.5. Application Code

Nearly all the application code was ported from Gecko without incident. The main surprise has been in the diversity of data feeds; one system sends us about 1000 small files per day, while another sends us one 140GB file per month! And it is just plain awkward to deal with a 140GB file.

5.6. Other Stuff

5.6.1. Zookeeper

Monitoring and managing a cluster of nodes is not substantially more difficult than managing an SMP environment, but can be much more tedious. While we do not view the cluster as a single (virtual) system, we do require that we can manage it through a single user interface. This user interface, which we call *zookeeper*, presents customised views representing various aspects of the cluster. There are four basic parts of the infrastructure supporting *zookeeper*:

• various scripts that gather various system statistics and details, which are then processed into *view data*. View data is distributed by HTTP.

• Java classes, or *views*, which display view data. Views may represent hardware/system information (disk usage, load averages) or daemon specific information (seneschal queue and load distribution information).

• a broadcast mechanism which supplies low latency incremental updates to view data (via UDP/IP packets).

• a method for hierarchically constructing a new view from an arrangement of other views.

Zookeeper's user interface provides a simplistic, but useful, ability to remotely control cluster components by allowing for command buttons within a view. When 'pressed,' a command button executes a preprogrammed command which can allow the user to do such things as start / stop / pause critical processes, pause or resume operations on a node, and even power cycle a node without the user needing to physically log into the machine.

The graphical user interface portion of zookeeper was implemented as a Java application, rather than a pure Xwindows application. While less efficient and rather more quirky, this empowers the unwashed masses (who do not run X) to to see what is going on. The user interface was not written as an applet as it has the need to retrieve and communicate with hosts other than the host that would supply the applet; something not allowed under the definition of an applet.

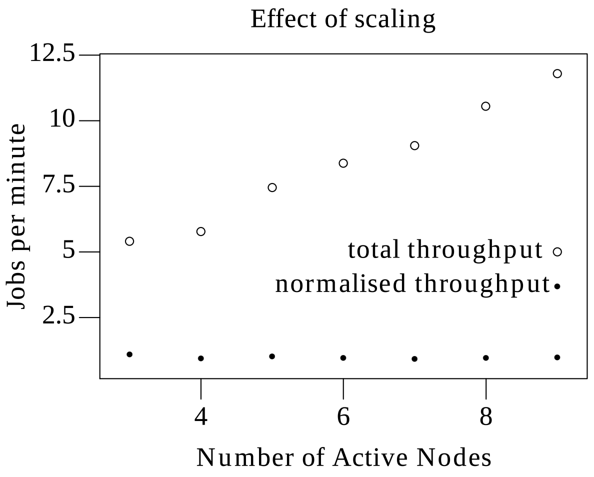

Figure 6: Experiments in scaling.

5.6.2. Scalability

Ningaui is inherently scalable because of its loosely coupled design, especially as it's unit of design, the node, is fairly well balanced in terms of CPU and I / O. Figure 6 shows how the performance scales with the number of nodes. It shows both the total throughput, as well as the throughput divided by the sum of the throughputs of the nodes involved. This normalised throughput is relatively constant. The current hardware and software architecture could scale to about 100 nodes comfortably. Above that, we have some issues:

• networking. The CLAN hardware can handle fairly large numbers of nodes but eventually, any single network will not scale. This mostly affects the replication manager, and in our new design, we are thinking of supporting subgroups of nodes with peer-to-peer interfaces with other subgroups. By having different subgroups using different network fabrics, the network can be increased to a large size, at least up to the thousands of nodes.

• replication manager. *Repmgr* gets sent file lists from all its nodes every several minutes. This does *not* scale well! On the other hand, the above subgroup functionality would also mitigate this problem, as the total number of nodes would increase as the square of the *repmgr* limit.

6. Epilog

One of the unexpected results of the Gecko work was a relationship between high performance architectures for SMP machines and networked clusters:

> "designing for a scalable cluster of systems networked together is isomorphic to designing for a single system with a scalable number of filesystems. Just as with a cluster of systems, where you try to do nearly all the work locally on each system and minimise the inter-system communications, you arrange that processing of data on each filesystem is independent of processing on any other filesystem."[Hum00]

The foremost goal on Ningaui was to verify this relationship and to evaluate whether or not a loosely coupled cluster of Linux systems was in fact competitive with large, industrial strength, high end SMP servers.

Our evaluation is not over; we have still to go through the grind of doing production runs for several months before we know for sure. But so far, the answer is yes. Despite our litany of problems with Linux and the GNU software, it is a smaller and less serious list than what we faced with Sun and Solaris in Gecko. And in almost all cases, the workarounds were fairly easy, even if tedious. They have also reinforced our professional grade paranoia; we believe in checksumming, and checksumming often. (Recently, we discovered that the *zlib* compression library does not detect and pass through *all* I / O errors, and as a result, a file system running out of space was not detected and some poor operator had to resend us several thousand files.)

Even if we judge the software reliability issues as even, the tremendous cost, availability, and scalability advantages of the cluster are just too great. Eventually, clusters will rule the business world, particularly the medium to high end.

Glossary

bid	A request for an amount of work. Sent by a node to seneschal.
biller	One or more programmes responsible for calculating a customer's bill based on information recorded by the switch for each phone call.
cycle	The process which takes all of the tags received during a period of time (usually 24 hours) and updates the database with them, deleting old tagsets, and generating reports.
feed	A series of data files received from the same source. (See stream)
hint	A suggestion given to an application such as *repmgr* regarding how it should manage something within its domain. Hints are not mandatory and may be ignored if complying with the hint would affect the correctness of the domain.
lease	A period of validity for things such as hints, and jobs. The application managing an item with a lease must assume that the item (hint, job, etc) is valid until the item is specifically changed, or until the lease expires. A lease expires when its endpoint is no longer in the future.
parsing	The process by which records within a data file are converted to tags.
stream	A group of feeds which are related and can be processed in the same manner.
tap	A point in the billing flow where data is syphoned off and sent to Ningaui.
tag	The set of information that is extracted from each data record. A tag represents a single phone call within the billing flow at a single instance in time.
tagset	A group of tags which represent the same phone call. The tagset describes the history of a phone call as it was processed within the billing environment.

Acknowledgements

This work was a team effort; we have had vital help from Angus MacLellan and assistance from Christof Fetzer, Mark Plotnick, and Debi Balog.

The comments of the reviewers and our shepherd improved this paper; the remaining errors are those of the authors.

This is an experience paper, and as such, contains various statements about certain products and their behaviour. Such products evolve over time, and any specific observation we made may well be invalid by the time you read this paper. Caveat emptor.

Contact Information

For any more information about this paper, or the software described, please contact Andrew Hume. We expect to be able to release some of the software tools described here, but the details will vary over time.

References

[Bec95]. Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer, "BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION," *International Conference on Parallel Processing*, pp. 20-25, IEEE, `http://www.beowulf.org/papers/papers.html` (1995).

[Fet99]. C. Fetzer and F. Cristian, "A Highly Available Local Leader Service," *IEEE Transactions on Software Engineering* **25**(6), pp. 603-618, `http://www.research.att.com/~christof/HALL` (Sep 1999).

[Fow00]. Glenn S. Fowler, David G. Korn, Stephen C. North, and Kiem-Phong Vo, "The AT&T AST OpenSource Software Collection," pp. 187-195 in *USENIX Conference Proceedings*, USENIX, San Francisco (Summer 2000).

[Hum00]. Andrew Hume, Scott Daniels, and Angus MacLellan, "Gecko: Tracking A Very Large Billing System," pp. 93-105 in *USENIX Conference Proceedings*, USENIX, San Francisco (Summer 2000).

[Mag02]. Kostas Magoutis, "Design And Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD," *BSDCon02*, San Francisco, pp. 65-76 (Feb 2002).

[Pop90]. Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann, “Replication in Ficus Distributed File Systems,” *Workshop on Management of Replicated Data*, pp. 20-25, IEEE (Nov 1990).

[Qui02]. Sean Quinlan and Sean Dorward, “Venti: a new approach to archival storage,” *Conference on File System and Storage Technologies*, Monterey, pp. 89-101, USENIX (Feb 2002).

[Tri96]. Andrew Tridgell and Paul Mackerras, “The rsync algorithm,” ANU Computer Science Technical Reports TR-CS-96-05, Australian National University (1996). `http://rsync.samba.org/`

[Wen01]. Torre Wenaus, BNL (2001). `http://atlassw1.phy.bnl.gov/magda/info`