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

For more information about USENIX Association contact:

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

Extensible, Scalable Monitoring for Clusters of Computers

Eric Anderson and Dave Patterson – U. C. Berkeley

ABSTRACT

We describe the CARD (Cluster Administration using Relational Databases) system[1] for monitoring large clusters of cooperating computers. CARD scales both in capacity and in visualization to at least 150 machines, and can in principle scale far beyond that. The architecture is easily extensible to monitor new cluster software and hardware. CARD detects and automatically recovers from common faults. CARD uses a Java applet as its primary interface allowing users anywhere in the world to monitor the cluster through their browser.

Introduction

Monitoring a large cluster of cooperating computers requires extensibility, fault tolerance, and scalability. We handle the evolution of software and hardware in our cluster by using relational tables to make CARD extensible. We detect and recover from node and network failures by using timestamps to resynchronize out system. We improve data scalability by using a hierarchy of databases and a hybrid push/pull protocol for efficiently delivering data from sources to sinks. Finally, we improve visualization scalability by statistical aggregation and using color to reduce information loss.

We have designed and implemented a system for flexible, online gathering and visualization of statistics and textual information from hundreds of data sources. CARD gathers node statistics such as CPU and disk usage, and node information such as executing processes. We will describe the system we developed to monitor our cluster, explain how we solved the four problems described above, and show how we synthesized research from other fields and applied them to our problem.

To make CARD flexible and extensible, we have chosen to gather data and store it in a relational database [Codd76]. New subsystems can access the data through SQL [Cham76] without requiring modification of old programs. We use SQL to both execute ad-hoc queries over the database, and to extract data for visualization in our Java [Gosl95] applet. Relational tables make CARD naturally extensible because new data can go into a new table without affecting old tables. In addition, new columns can be added to tables without breaking older programs. The column names also help users understand the structure of the data when browsing. We have used additional tables of descriptions to further assist in browsing.

We use timestamps to detect and recover from failures in CARD and the cluster. Failures are detected when periodic data updates stop. Changing data is synchronized using timestamp consistency control. Stale data is expired when the timestamps are too old.

We have achieved data scalability, which is the ability to handle more data as machines are added, by building a hierarchy of databases. The hierarchy allows us to batch updates to the database, specialize nodes to interesting subsets of the data, and reduce the frequency of updates to the higher level nodes. This increases the scalability of the database, but the last two approaches reduce either the scope or the freshness of the data. Users of the data may then need to contact multiple databases, to gain data coverage or to get the most up to date information.

We have improved the network efficiency, and hence the data scalability by creating a hybrid push-pull protocol for moving data from sources to sinks. Our protocol sends an initial SQL request and a repeat rate. The query is executed repeatedly, and the results are forwarded to the requestor. The hybrid protocol achieves the best of both a request-response (pull) protocol and an update (push) protocol.

We increase visualization scalability, which is the ability to gracefully increase the amount of data displayed without having to increase the screen space, through statistical aggregation of data, and the resulting information loss is reduced by using different shades of the same color to display dispersion. These two techniques have allowed us to meaningfully display multiple statistics from hundreds of machines.

The remainder of the paper is structured as follows. The next section describes our four solutions, the Implementation section describes our experience with our implementation, and the next section describes the related work. The last section summarizes our conclusions from building the system.

[1]CARD is available from http://now.cs.berkeley.edu/Sysadmin/esm/intro.html .

Four Problems and Our Solutions

We will now describe our solutions to four problems of monitoring large clusters. First, we will explain how we handle the evolution of software and hardware in a cluster. Second, we will explain how we deal with failures in the cluster and our software. Third, we will explain how we increase data scalability. Fourth, we will explain how we display the statistics and information from hundreds of machines.

Handling Rapid Evolution using Relational Databases

Cluster software is evolving at a rapid pace, so a monitoring system needs to be extensible to keep up with the changes. This means that new data will be placed in the system, and usage of the data will change. Hence a system with only one way of storing or querying data will have trouble adapting to new uses.

We believe that flexibility and extensibility can be achieved by using a relational database to store all of the data. The database increases flexibility by decoupling the data users from the data providers, which means that arbitrary processes can easily put information into the database, and arbitrary consumers can extract the data from the system. The database also improves flexibility by supporting SQL queries over the data. SQL queries can combine arbitrary tables and columns in many ways, and the database will automatically use indices to execute the queries efficiently. As the use of the database changes, new indices can be added to maintain the efficiency of the queries. The database increases extensibility because new tables can be easily added, and new columns can be added to old tables without breaking old applications. Queries only address columns in tables by name, and hence the new columns do not affect the old queries.

Using a database is a significant departure from previous systems [Apis96, Dolphin96, Fink97, Hans93, Hard92, Scha93, Schö93, Seda95, Ship91, Simo91, Walt95], which all use a custom module for data storage and few provide any external access to the data. While building an integrated module can increase efficiency for a single consumer of the data, some of that improvement is lost with multiple consumers. Furthermore, the flexibility of the system is reduced because adding new data producers and consumers is more difficult.

Recovering from Failures using Timestamps

The second problem we address is detecting and recovering from failures. We use timestamps to detect when parts of the system are not working, identify when data has changed, and determine when data has become old and should be rechecked or removed.

Timestamps help detect failures when data sources are generating periodic updates. If the timestamp associated with the data is not changing, then the check has failed, which indicates that the remote node is either slow or broken. This solution works even if the updates are propagating through multiple databases in the hierarchy because the timestamps are associated with the data and don't change as the data moves.

We also use timestamps for consistency control [Chan85]. Timestamps allow quick comparisons of data to determine if it has been updated. We have a timestamp associated with both the data, and the time for when the data was placed in the database. Remote processes maintain a last timestamp (t_0). To synchronize with the database, they get a new timestamp from the database (t_1), get all the data that was added since t_0, and set t_0 to t_1. By repeating this process, the remote process can be kept weakly synchronized with the database. Moreover, if the machines' time is synchronized [Mill95], then the remote process also knows the freshness of their data. Timestamp consistency control is very simple to implement in comparison to other consistency protocols [Gray75], and if the database is accessible, then the data is always available regardless of other failures in the system, whereas other protocols may deny access to ensure stricter consistency.

Finally, we use timestamps to eliminate stale data. Stale data can occur because of failures or removals. The timestamps allow the data to be automatically removed after a table specific period, which

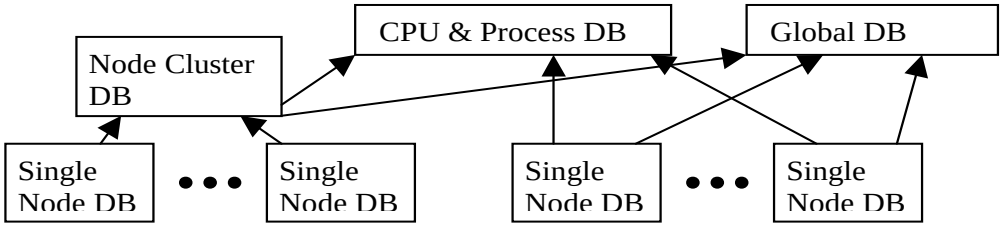

Figure 1: A hierarchy of databases. At the lowest level are single node databases. These hold information gathered from a single node. The top level shows a few forms of specialization. The node cluster database gathers information about all the single nodes in its cluster. The CPU and process database stores a subset of the data at the full frequency and takes advantage of the batching possible because of the node cluster database. The global database stores all the information about the cluster at a reduced frequency.

means that the system will automatically recover to a stable state. Multiple timers allow slowly changing data like physical memory to be updated infrequently yet not be declared stale.

Data Scalability using Hierarchy

Systems that can take advantage of multiple machines are usually more scalable. We have designed a hierarchy (Figure 1) of databases in order to make our system more scalable. The hierarchy provides many benefits.

A hierarchy allows updates to a database to be batched. This reduces the network overhead by reducing the number of packets that need to be transmitted. This allows more operations because the operations are not serialized and hence the latency of the network is unimportant. Finally, this allows the database to perform more efficient updates.

A hierarchy also allows specialization of nodes. While a single database may not be able to handle the full update rate for all of the information that is being collected, a single database may be able to handle a useful subset of the data at the full rate. For example, statistics about CPU usage and processes could be stored in a single database, allowing it to handle more nodes.

Furthermore, a hierarchy allows reduction in the data rate. For example, an upper level database could keep all of the information gathered at an interval of a few minutes. As the amount of data gathered grows, the interval can be reduced. This will allow a single database to always keep a complete, but more slowly changing, copy of the database.

Finally, a hierarchy over multiple machines allows for fault tolerance. Multiple databases can be storing the same information, and hence if one of the databases crashes, other nodes will still have access to the data.

Data Transfer Efficiency using a Hybrid Push/Pull Protocol

Given a hierarchy of databases, and a number of additional processes which are also accessing the data, the system needs an efficient method for transferring data from sources to sinks. Most systems use polling; a few also have occasional updates. Both approaches have disadvantages, so we have developed a hybrid push-pull protocol that minimizes wasted data delivery, maximizes freshness, and reduces network traffic.

The canonical pull protocol is RPC [Sun86]; SNMP is a mostly pull protocol. A pull-based system requires the sink to request every piece of data it wants from the source. If the sink wants regular updates, it polls the source. Since the source knows it wants regular updates, all of the request packets are wasted network bandwidth. Furthermore, if the data is changing slowly or irregularly, some of the polls will return duplicates or no data. However, polling has the advantage that since the data was requested, the sink almost always wants the data when it arrives. Between polls, the data goes steadily out of date, leading to a tradeoff between the guaranteed freshness of the data and the wasted traffic.

PointCast [Poin97] and Marimba Castanet [Mari97] use a push protocol. A push protocol delivers data all the time and allows sinks to ignore data if they don't want it. Multicast [Deer90] uses a pruned push model, and broadcast disks [Acha97] use a push model with broadcasts to all receivers. A push system is ideal when the sink's needs match the source's schedule since the data is current, and the network traffic is reduced because the sink is not generating requests. However, as the sink's needs diverge from the source's schedule, a push model begins delivering data that is wasted because the sink will just ignore it. Also, sinks have to wait until the source decides to retransmit in order to get data. The push model trades off between wasted transmissions and communication usage.

We use a hybrid model. Sinks send an SQL request to the forwarder along with a count, and an interval. The forwarder will execute the query until it has sent count updates to the requester. The forwarder will suppress updates that occur more frequently than interval. By setting the count to one, a sink can get the effect of the pull model. By setting the count to infinity, a sink can get the effect of the push model. Intermediate values allow requesters to avoid being overrun with updates by requiring them to occasionally refresh the query. The interval allows the requester to reduce the rate of updates. When the updates are no longer needed, the sink can cancel the request. In both cases, the sink can use the full power of SQL to precisely retrieve the desired data. Using SQL reduces one of the main disadvantages of the push model, which is that the data delivered is not the desired data. For example, in multicast, the only selection that can be made is the multicast address, no sub-selection of packets from particular sources can be made.

Visualization Scalability Using Aggregation

We have found that we need to use aggregation to scale the visualization to our whole cluster. We tried having a stripchart for every statistic we wanted, but ran out of screen space trying to display all of our machines. We therefore aggregate statistics in two ways: First, we combine across the same statistics for different nodes. For example, we calculate the average and the standard deviation across the CPU usage for a set of nodes. We then display the average as the height in the stripchart, and the standard deviation as the shade. Second, we aggregate across different statistics. For example, we combine together CPU, disk and network utilization to get a single statistic we call machine utilization. Using shade takes advantage of the eye's ability to perceive many different shades of a color [Murc84, HSV]. We also use color to help draw distinctions and identify important information. For

example, we use different colors for the I/O and user CPU usage, and we identify groups of down machines in red. Figure 2 shows a use of the first form of aggregation, and the uses of color as it is a snapshot of our system in use.

Implementation & Experience

We chose MiniSQL [Hugh97] for our SQL database. MiniSQL is freely available to universities, and has a very slight license for companies. The MiniSQL database is sufficient for our needs because we don't require complex SQL queries. In addition, because MiniSQL comes with source, we were able to extend it when necessary. For example, we added micro-second timestamps to the database so that we could extract data changing on a short time-scale. We also removed the need to wait for a response from the server after an SQL command, which allows groups of updates to be sent as a batch. Our use of MiniSQL also allows other sites to try our system without an expensive initial investment in a database.

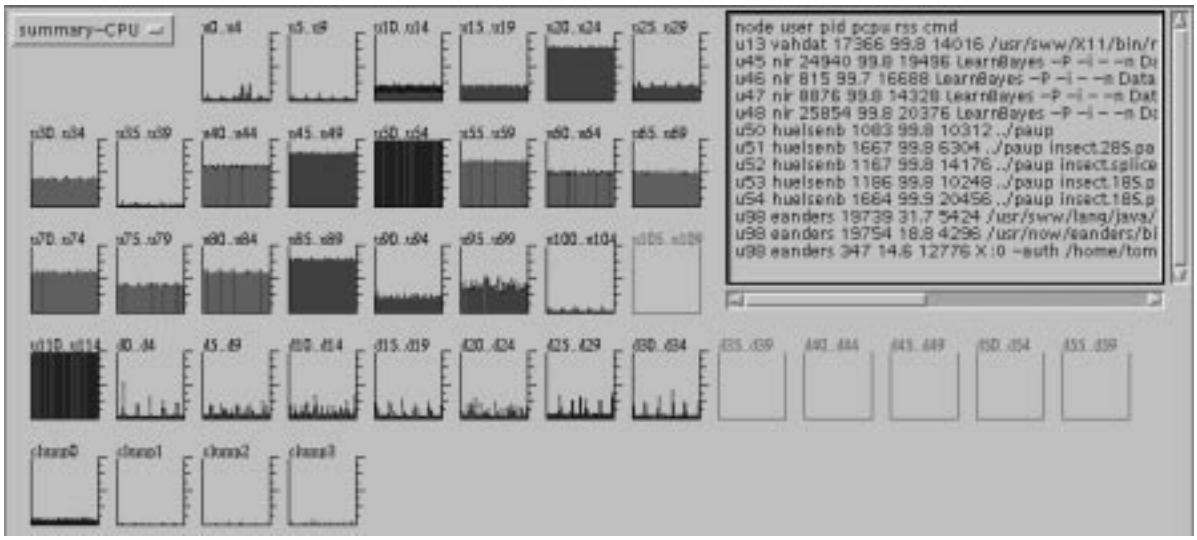

Figure 2: Snapshot of the Java interface monitoring our entire cluster which consists of 115 Ultra 1's, 45 Sparc 10's or 20's, and 4 Enterprise 5000 SMP's. Aggregration has been done with averages (height) and standard deviation (shade) across groups of 5 machines except for SMP's. The darker charts are more balanced across the group (u50..u54) all are at 100%), and the lighter charts are less balanced (u40..u44 have three nodes at 100% since the average is 60% and the usage is not balanced). All charts show the system CPU time in blue over the green user CPU time; u13 has a runaway Netscape process on it. Nodes u105-u109 and dawn35-dawn59 are down and shown in red. Processes running on selected nodes are shown in the text box in the upper right hand corner. A color version of this chart is available as http://now.cs.berkeley.edu/Sysadmin/esm/javadc.gif .

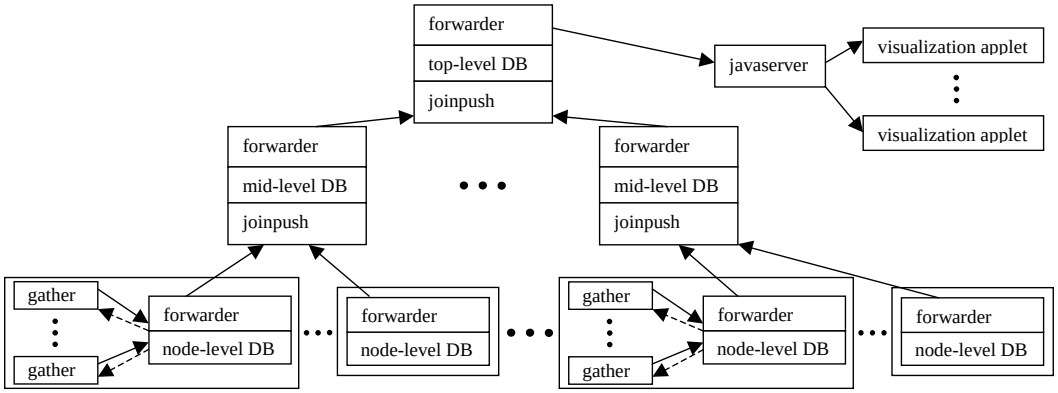

Figure 3: Architecture of our system. The gather processes are replicated for each forwarder/node-level database (DB) group. The top level databases can also be replicated for fault tolerance or scalability. The javaserver acts as a network proxy for the visualization applets because they can not make arbitrary network connections. The forwarder and joinpush processes are associated with a database and serve as the plumbing that moves data through the system.

Our experience using an SQL database has been very positive. We initially stored all of our configuration information in separate, per-program files. Given our experience with the reliability, and flexibility of the database, we have now moved the configuration information into the database.

Figure 3 shows the data flow among the major components of our system. The forwarder process, associated with each database, accepts SQL requests from sinks, executes them periodically and forwards the results to the sinks. The joinpush process merges the updates pushed from the forwarder processes into an upper-level database. The javaserver process acts as a network proxy for the Java visualization applet because applets cannot make arbitrary network connections. The visualization applet accepts updates from the javaserver, and displays them in stripcharts or a text window for the user. The applet also provides a simple way to select a pane of information to view.

Resource usage of MiniSQL has not been a problem. The database uses 1-2% of the CPU on an Ultra 1/170 to update 20-30 statistics a second. Our upper level databases seem to peak somewhere between 1500 and 2000 updates/second. There are many optimizations we could make to reduce or rebalance the load if necessary; we have already taken advantage of the indexing support in MiniSQL.

Initially we were not concerned with getting our system running, however we discovered after using it for a little while that making sure all of the pieces were functioning correctly, especially when we were changing parts, was tricky. We therefore created a process which watches to see if the various components on a node are reachable, and if they are not attempts to restart the node. We discovered that making sure that leftover processes didn't accumulate required careful design. We use two methods to ensure that old processes terminate quickly. First, each process creates a pid file and locks the file. The reset-node operation attempts to lock each file and if the lock fails, the process is sent a signal. Using a lock guarantees that we won't accidentally try to kill off a second process which just happens to have the same process id. Second, we write an epoch file each time the node is reset. Processes can check the epoch file, and if it doesn't match then they know they should exit. We added the second approach because we occasionally saw processes not exit despite having been sent a signal that should cause them to exit.

The forwarder, and joinpush processes are both implemented in C taking advantage of Solaris threads in order to achieve better efficiency. We initially tried implementing those processes in Perl, but the code was too inefficient to support 150 nodes, and using threads reduced concerns about blocking while reconnecting. The javaserver is implemented in Perl [Wall96] to simplify implementation and to support run-time extension. The applets are implemented in Java so that users can access our system without having to download or compile anything other than a Java enabled browser.

Data is added to the system by the gather process, which is also implemented in Perl. We originally examined a threaded implementation of the gather process, but we were unable to get a sufficiently stable multi-threaded implementation. We therefore use a multi-process implementation. We have a directory of files which all get spawned by the first process, which then waits around and flushes old data from the database. We currently have threads that extract CPU, I/O, network, process and machine statistics. We also have a specialized thread for extracting information about our high-speed Myrinet [Myri96] network and our lightweight Active Messages implementation [Chun97].

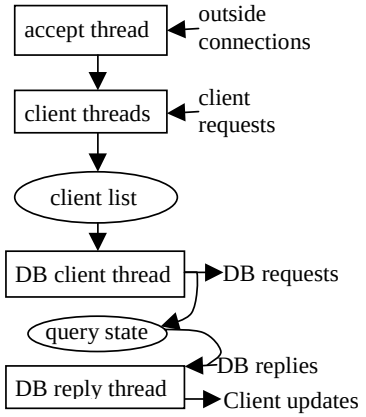

Figure 4: Architecture of the forwarder. The left column shows either threads or important data structures in the forwarder. The right column shows the interactions with other processes.

Figure 4 shows the architecture of the forwarder. The accept thread gets outside connections from clients and immediately forks a client thread to deal with each client. The client threads take requests for updates from the clients, and put those requests in the structure associated with each client. The use of threads allows us to easily handle slow clients without concerns of blocking. The database client thread walks the list of clients, and issues the requests which are pending to the database. When the response comes back from the database, it is matched with the information stored at request time, and the reply thread sends the updates to the appropriate client.

Figure 5 shows the architecture of the joinpush process. The list of forwarders and the data to request is configured externally to joinpush to simplify the implementation. The reconnect thread forks separate threads to connect to each of the forwarders and issue a request. When a connection is made, the connection is added to the connections list, and the update thread waits around for updates from any of the forwarders. It generates an appropriate database update. The reply

thread will generate an insert request if the reply indicates that the data was not yet in the table.

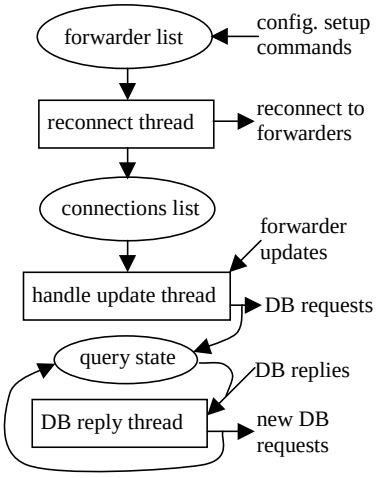

Figure 5: Architecture of joinpush. The left column shows either threads or important data structures in joinpush. The right column shows the interactions with other processes. The configuration commands are handled analogously to how clients are handled in the forwarder.

We have found the timestamps associated with the data to be extremely useful. For example, an early version of CARD failed to send the timestamp from the javaserver to the visualization applet. When the javaserver generated repeated updates of old data, the client was fooled into thinking the system was performing correctly. Now that we forward timestamps, we would detect this error.

Because MiniSQL doesn't support triggers [Eswa76], our actual implementation of the forwarder simply executes the query count times every interval seconds. This implementation is a reasonable approximation of the more optimal hybrid protocol. The main disadvantage is that the forwarder will sometimes execute the query and there will have been no updates. Since the forwarder is on the same node as the database, and the query restricts the results to new data, this is a fairly minor problem. A more optimal implementation would merge the forwarder and the database into the same process.

The fact that we display information through a Java applet raises a few privacy concerns. In particular, outside users can see all of the statistics of the cluster. Given that we are an academic institution, we have not been very concerned about maintaining secrecy of our usage information. However, all of the standard techniques for improving privacy could be added to our system. For example, access could be limited by IP address, or secure, authenticated connections could be established via the secure socket layer [Frei96]. To ensure privacy, it would be necessary to apply this protection to the javaserver, the forwarder, and the MiniSQL server. To prevent bogus information from being added into the database, it might also be necessary to protect the joinpush process.

Related Work

The most closely related work is TkIned [Schö93, Schö97]. TkIned is a centralized system for managing networks. It has an extensive collection of methods for gathering data. Because it is distributed with complete source code, it can be extended by modifying the program. Since the data is not accessible outside of the TkIned program, new modules either have to be added to TkIned, or have to repeat the data gathering. TkIned provides simple support for visualization and does not aggregate data before displaying it. TkIned's centralized pull model limits its scalability.

Pulsar [Fink97] uses short scripts (pulse monitors) which measure a statistic, and send an update to a central display server if the value is out of some hard-coded bounds. Pulse monitors run infrequently out of a cron-like tool. Pulsar can be extended by writing additional pulse monitors, and adding them to a configuration file. Pulsar's centralized design is not fault tolerant, and only simple support for external access to updates. Pulsar does not support monitoring of rapidly changing statistics.

SunNet Manager [SNM] is a commercially supported, SNMP-based network monitoring program. Other companies can extend it by writing drop-in modules to manage their equipment. Using SNMP version 2 [Case96], or Sun proprietary protocols, it has some support for multiple monitoring stations to communicate with each other. As with other monolithic systems, it has poor scalability and weak extensibility.

The DEVise [Livn96, Livn97] system is a generic trace file visualization tool. DEVise supports converting a sequence of records (rows in a table) into a sequence of graphical object, displaying the graphical objects, and performing graphical queries on the objects. DEVise uses SQL queries to implement the graphical queries, and supports visualizing trace files larger than the physical memory on a machine. Unfortunately, it does not support online updates to visualized data, and so does not directly match our needs, but we are using a similar idea of translating database updates into graphical objects.

Conclusion

Decoupling data visualization and data gathering through the relational database has greatly improved the flexibility and structure of our system. It led to our success in using relational tables for flexible data storage and access. It also led to the idea of using a hierarchy and a hybrid protocol for efficient data transfer. Timestamps have been very useful in detecting internal system failures and automatically

recovering from them. Since the machines are used on a research project [Ande95] exploring how to use hundreds of machines in cooperation to solve complex problems, aggregation in visualization was required by the scale and class of the system we wanted to monitor. We expect in the future to monitor more of the software and hardware in our cluster, including more research systems. CARD is available for external use from http://now.cs.berkeley.edu/Sysadmin/esm/intro.html .

Acknowledgements

The authors would like to thank Remzi Arpaci-Dusseau, Armando Fox, Steve Gribble, Kim Keeton, Jeanna Matthews, Amin Vahdat, and Chad Yoshikawa for their help clarifying and condensing the important contributions of the paper.

Eric Anderson is supported by a Department of Defense, Office of Naval Research Fellowship. This work was also supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Intel, Hewlett Packard, Microsoft and Mitsubishi.

Author Information

Eric A. Anderson (University of California at Berkeley) is a Ph.D. candidate in computer science working on System Administration. His thesis concerns monitoring and diagnosing problems in a cluster of computers. Eric's research interests include combining operating systems, distributed systems, and networks to build large systems capable of solving complex problems. Reach him electronically at <eanders@u98.cs.berkeley.edu> .

David A. Patterson (University of California at Berkeley) has taught computer architecture since joining the faculty in 1977, and is holder of the Pardee Chair of Computer Science.
At Berkeley, he led the design and implementation of RISC I, likely the first VLSI Reduced Instruction Set Computer. This research became the foundation of the SPARC architecture, currently used by Fujitsu, Sun, and Texas Instruments. He was also a leader of the Redundant Arrays of Inexpensive Disks (RAID) project, which led to high performance storage systems from many companies. These projects led to three distinguished dissertation awards from the Association for Computing Machinery (ACM). He is also co-author of five books and is chair of the Computing Research Association.
Patterson has won teaching awards from his campus, the ACM, and the Institute of Electrical and Electronic Engineers (IEEE). He is a Fellow of both the ACM and the IEEE, and is a member of the National Academy of Engineering. Reach him electronically at <patterson@cs.berkeley.edu> .

References

[Acha97] "Balancing Push and Pull for Data Broadcast," Swarup Acharya, Michael Franklin, and Stan Zdonik. ACM SIGMOD Intl. Conference on Management of Data. May 1997.

[Ande95] "A Case for NOW (Networks of Workstations)," T. Anderson, D. Culler, D. Patterson, and the NOW team. *IEEE Micro*, pages 54-64, February 1995.

[Apis96] "OC3MON: Flexible, Affordable, High Performance Statistics Collection," Joel Apisdorf, k claffy, Kevin Thompson, and Rick Wilder. *Proceedings of the 1996 LISA X Conference.*

[Case90] "A Simple Network Management Protocol (SNMP)," J. Case, M. Fedor, M. Schoffstall, and J. Davin. Available as RFC 1157 from http://www.internic.net/ds/dspg1intdoc.html

[Case96] "Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)," J. Case, K. McCloghrie, M. Rose, S. Waldbusser. Available as RFC 1907.

[Cham76] "SEQUEL 2: A unified approach to data definition, manipulation, and control," Chamberlin, D. D., et al. *IBM J. Res. and Develop*. Nov. 1976, (also see errata in Jan. 1977 issue).

[Chan85] "Distributed snapshots: Determining global states of distributed systems," M. Chandy and L. Lamport. *ACM Transactions on Computer Systems*, February 1985.

[Chun97] "Virtual Network Transport Protocols for Myrinet," B. Chun, A. Mainwaring, D. Culler, *Proceedings of Hot Interconnects V*, Auguest 1997.

[Codd71] "A Data Base Sublanguage Founded on the Relational Calculus," Codd, E. *Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description*, Access and Control. San Diego, CA. Nov 1971.

[Deer90] "Multicast Routing in Datagram Internetworks and Extended LANs," Stephen E. Deering and David R. Cheriton. *ACM Transactions on Computer Systems*, May, 1990.

[Dolphin96] "HP Dolphin research project," Personal communication with author and some of the development group.

[Eswa76] "Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated DataBase System," Eswaran, K. P. *IBM Research Report RJ1820*. August, 1976.

[Fink97] "Pulsar: An extensible tool for monitoring large Unix sites," Raphael A. Finkel, Accepted to *Software Practice and Experience*.

[Frei96] "The SSL Protocol: Version 3.0," Freier, A., Karlton, P., and Kocher, P. Internet draft available as http://home.netscape.com/eng/ssl3/ssl-toc.html .

[Gosl95] "The Java Language Environment: A White Paper," J. Gosling and H. McGilton, http://java.dimensionx.com/whitePaper/java-whitepaper-1.html .

[Gray75] "Granularity of Locks and Degrees of Consistency in a Shared DataBase," Jim Gray, et. al. IBM-RJ1654, Sept. 1975. Reprinted in Readings in *Database Systems*, 2nd edition.

[Hans93] "Automated System Monitoring and Notification With Swatch," Stephen E. Hansen & E. Todd Atkins, *Proceedings of the 1993 LISA VII Conference*.

[Hard92] "buzzerd: Automated Systems Monitoring with Notification in a Network Environment," Darren Hardy & Herb Morreale, *Proceedings of the 1992 LISA VI Conference*.

[HSV] "Hue, Saturation, and Value Color Model," http://loki.cs.gsu.edu/edcom/hypgraph/color/colorhs.htm .

[Hugh97] "Mini SQL 2.0," http://hughes.com.au/ .

[Livn96] "Visual Exploration of Large Data Sets," Miron Livny, Raghu Ramakrishnan, and Jussi Myllymaki. In *Proceedings of the IS&T/SPIE Conference on Visual Data Exploration and Analysis*, January, 1996.

[Livn97] "DEVise: an Environment for Data Exploration and Visualization," Miron Livny, et. al. http://www.cs.wisc.edu/~devise/

[Mari97] "Marimba Castanet," http://www.marimba.com/

[Mill95] "Improved Algorithms for Synchronizing Computer Network Clocks," David Mills, *IEEE/ACM Transactions on Networking*, Vol. 3, No. 3, June, 1995.

[Murc84] "Physiological Principles for the Effective Use of Color," G. Murch, *IEEE CG&A*, Nov, 1984.

[Myri96] "The Myricom network," Described at http://www.myri.com/myrinet/

[Poin97] "PointCast: the desktop newscast," http://www.pointcast.com/

[Scha93] "A Practical Approach to NFS Response Time Monitoring," Gary Schaps and Peter Bishop, *Proceedings of the 1993 LISA VII Conference*.

[Sch93] "How to Keep Track of Your Network Configuration," J. Schönwälder & H. Langendörfer, *Proceedings of the 1993 LISA VII Conference*.

[Sch97] "Scotty Tnm Tcl Extension," Jürgen Schönwälder, http://wwwsnmp.cs.utwente.nl/~schoenw/scotty/ .

[Seda95] "LACHESIS: A Tool for Benchmarking Internet Service Providers," Jeff Sedayao and Kotaro Akita, *Proceedings of the 1995 LISA IX Conference.*

[Ship91] "Monitoring Activity on a Large Unix Network with perl and Syslogd," Carl Shipley & Chingyow Wang, *Proceedings of the 1991 LISA V Conference*.

[Simo91] "System Resource Accounting on UNIX Systems," John Simonson. *Proceedings of the 1991 LISA V Conference*.

[SNM] "Sun Net Manager," Sun Solstice product.

[Sun86] "Remote Procedure Call Programming Guide," Sun Microsystems, Inc. Feb 1986.

[Wall96] "Perl 5: Practical Extraction and Report Language," Larry Wall, et al., Available from ftp://ftp.funet.fi/pub/languages/perl/CPAN/ .

[Walt95] "Tracking Hardware Configurations in a Heterogeneous Network with syslogd." Rex Walters. *Proceedings of the 1995 LISA IX Conference.*