Scalable Causal Message Logging for Wide-Area Environments

Karan Bhatia[1], Keith Marzullo[2], and Lorenzo Alvisi[3]

[1] Advanced Technology Group, Entropia Inc.
San Diego, CA 92121
karan@entropia.com
http://www.entropia.com/
[2] University of California, San Diego,
Department of Computer Science and Engineering
marzullo@cs.ucsd.edu
http://www-cs.ucsd.edu/users/marzullo/
[3] Department of Computer Sciences
The University of Texas at Austin
lorenzo@cs.utexas.edu
http://www.cs.utexas.edu/ lorenzo

Abstract. Causal message logging spread recovery information around the network in which the processes execute. This is an attractive property for wide area networks: it can be used to replicate processes that are otherwise inaccessible due to network partitions. However, current causal message logging protocols do not scale to thousands of processes. We describe the Hierarchical Causal Logging Protocol (HCML) that is scalable. It uses a hierarchy of *proxies* to reduce the amount of information a process needs to maintain. Proxies also act as caches for recovery information and reduce the overall message overhead by as much as 50%. HCML also leverages differences in bandwidth between processes that reduces overall message latency by as much as 97%.

1 Introduction

Large scale computational grids are gaining popularity as infrastructure for running large scientific applications. Examples of such grids exist in the academic world (SETI@Home [5], Globus [9], and Nile [13]) as well as in the commercial sector (Entropia and Parabon among others). A primary goal of all these grids is to leverage the increased computational power of desktop personal computers and workstations that are linked by high speed communication protocols to provide a virtual supercomputer with the aggragate computational power that is many times that of current supercomputers. For example, the largest current grid is the SETI@Home grid that includes over two million desktop personal computers. The aggragate computing power of this grid exceeds that of all the top 500 supercomputers [12].

Before applications can leverage this virtual supercomputer, many problems inherent to large-scale systems must be solved. One of the key challenges is fault tolerance. In any system of thousands or millions of computers, the likelihood of multiple failures is high. Many of the current applications that utilize these grids are decomposed into independent pieces where each piece is assigned to a particular host machine. For such applications, failures can be dealt with by re-running a failed computation on a different host. Consistency is not an issue since each piece of computation is independent of all other pieces. Such is the case for the Seti@Home application.

However, for many other applications, the application can not be divided into independent pieces. Each piece has dependencies on other pieces of the application, and it is not sufficient to simply restart a failed piece of the computation on another host. The system must ensure that the restarted computation maintains consistency with the other parts of the application. Examples of such applications include Jacobi grid and other relaxation style algorithms. Such applications are typically written using a communication library where each piece of the computation communicates with other pieces by sending and receiving messages. The messages define the dependencies between the sender process and the receiver process. The transitive closure of the message dependencies define the overall application dependencies. Restarted processes must respect the application dependencies.

Simple checkpointing mechanisms will not suffice for fault tolerance in these environments. The host machines are geographically widely distributed. Network partitions can make checkpoint files ubavailable when needed. The checkpointed state needs to be cached at various places in the network.

One class of protocols, called Causal Message Logging protocols [4, 8], cache recovery information in an appropriate manner. They operate by logging the recovery information in the volatile memory of the application processes and by dispersing the recovery information by piggybacking it on to application messages. The dispersed recovery information can then be used to generate replicas that are *causally consistent* [1] with the rest of the application. Causal message logging protocols also have a low overhead during failure-free executions and send no extra messages to distribute the recovery information.

While causal message logging protocols have been used successfully in local-area-networks, they are not suitable for use in large wide-area environments. These protocols maintain data structures that grow quadratically in the number n of processes in the system. For large n the memory needed to maintain these data structures can easily become unmanageable. In addition, the higher latency and decreased bandwidth of wide area computing can lead to a large increase in the amount of data that these protocols piggyback on the ambient message traffic.

This paper presents an implementation of causal message logging that is designed for use in large scale, wide-area grid infrastructures. An expanded version of the paper is available as [6]. Hierarchical Causal Message Logging (HCML) utilizes a network of *proxies* situated throughout the network that cache recovery information while routing application messages. Using proxies exponentially reduces the size of the data structures needed to track causality. We have also found that using proxies reduces significantly the bandwidth overhead of distributing recovery information throughout the network.

2 System Model

We assume a system with a set $\mathcal{P}$ of n processes, whose execution is represented by a *run*, which is an irreflexive partial ordering $\rightarrow$ of the *send* events, *receive* events and *local* events based on potential causality [11]. Processes can communicate only by sending and receiving messages; communication is FIFO and reliable. The system is asynchronous: there exists no bound on the relative speeds of processes, no bound on message transmission delays, and no global time source. A deliver event is a local event that represents the delivery of a received message to the application or applications running in that process. For any message m from process p to process q, q delivers m only if it has received m, and q delivers m no more than once.

We assume that processes are *piecewise deterministic* [7, 14], *i.e.* that it is possible to identify all the non-deterministic events executed by each process and to log for each such event a *determinant* [4] that contains all the information necessary to replay the event during recovery. In particular, we assume that the order in which messages are delivered is non deterministic, and that the corresponding deliver events are the only non-deterministic events that a process executes. The determinant $\#m$ for the deliver event of a message m includes a unique identifier for m as well as m's position in the delivery order at its destination. The contents of the message need not be saved because it can be regenerated when needed [4].

Given a message m sent from process p to process q, we define $Dep(m)$ to be the set of processes that have executed an event e such that $receive_q(m,p) \rightarrow e$. Informally, this is the set of processes that causally depend on the delivery of message m (including q) once it has delivered m. We also define the set $Log(m)$ as the set of processes that have stored a copy of $\#m$ in their volatile memory.

Definition 1 (Causal Logging Property). *The causal logging specification defined in [2] requires that $Dep(m) \subseteq Log(m)$ when any subset of processes can crash.*

We define a *locality hierarchy* as a rooted tree $\mathcal{H}$ with the processes in $\mathcal{P}$ as the leaves of the tree. Each interior nodes of the tree represent a *locale*, such as a specific processor, local-area network, or a stub domain. Given $\mathcal{H}$, we denote with $\mathcal{C}(x,y)$ the least common ancestory or x and y and $\hat{x}$ to be the parent of x.

Each locale in $\mathcal{H}$ has associated with it a characteristic that defines the available bandwidth for communication among the locale's children. If two application processes s and t have the same parent p in $\mathcal{H}$, then the communication cost of a message m from process s to process t depends on the bandwidth characteristics of their parent p. If s and t do not have the same parent, then the communication cost of message m depends on the bandwidth characteristics of the locale $\mathcal{C}(p, q)$. We assume that all locales at the same height i have the same bandwidth BW_i (measured in MB/sec).

The overhead of a message m, denoted as $|m|$, is the size in bytes of the fault tolerance information piggybacked on m. The transmission overhead of m is a the time it takes to transmit $|m|$ from its sender to its destination $m.dest$ (based on the slowest locale in the path). The *total message overhead* of a run is the sum of the message overhead for all the messages sent in the run. The message overhead at depth i of the hierarchy is the sum of the message overhead of messages that traverse locales at height i. The *total transmission overhead* is the sum of the transmission overheads for all messages in the run.

3 Hierarchical Design

In this section we first review a simple causal message logging protocol that we call SCML. It is equivalent to the protocol Π_{det} with $f = n$ described in [3] and to Manetho [8]. We then discuss its limitations with respect to scaling, and present a hierarchical and scalable causal message logging protocol.

3.1 Review of SCML

Like other message logging protocols, causal message logging is built using a *recovery unit* abstraction [14]. The recovery unit acts like a filter between the application and the transport layer. When an application sends a message, the recovery unit records fault tolerance information on the message and hands it off to the transport layer. Similarly, on the receiving end, the recovery unit reads the fault tolerance information on the message and updates its in-memory data structures before passing the contents of the message to the application layer.

The recovery unit for causal message logging maintains a *determinant array* $\mathbf{H}_s$ at each process s. For every process t, $\mathbf{H}_s[t]$ contains the determinant of every message delivered by t in the causal past of s. $\mathbf{H}_s[t]$ is ordered by the order of message delivery at t. We denote with $\mathbf{H}_s[t, i]$ the i^{th} determinant in $\mathbf{H}_s[t]$.

A simplistic way to maintain $\mathbf{H}_s$ is as follows. When a process s sends a message m to t, it piggybacks on m all determinants in $\mathbf{H}_s$. When process t receives m, it extracts these piggybacked determinants, incorporates them into $\mathbf{H}_t$, generates the determinant for m, and appends $\#m$ to $\mathbf{H}_t[t]$. By doing so, when process t delivers m it has all the determinants for messages that were delivered causally before and including the delivery of m and therefore satisfies the causal logging property. This method of maintaining $\mathbf{H}$, however, needlessly piggybacks many determinants. To reduce the number, each process s maintains a *dependency matrix* $\mathbf{D}_s$. This is a matrix clock where the value $\mathbf{D}_s[t, u]$ is an index into $\mathbf{H}_s[u]$. If $\mathbf{D}_s[t, u] = j$, then process s knows that all of the determinants in $\mathbf{H}_s[u]$ up through $\mathbf{H}_s[u, j]$ have been sent to t.

This protocol does not scale well with n because the size of $\mathbf{D}$ is $O(n^2)$, and grid computation middleware is being designed for systems in which n in the thousands or millions. Even if n were the relatively small value of 10,000, then each process would need to maintain a dependency matrix whose size would be measured in gigabytes.

3.2 Proxy Hierarchy

HCML addresses the scalability problems of causal message logging through hierarchy. Each process tracks only a small subset of the processes, thereby effectively reducing n for each process. Doing so also reduces the number of times a process is affected by another process joining or leaving the system; a process is affected only when the joining or leaving process is in the subset of processes it tracks. The hierarchy we use is based on the locality hierarchy $\mathcal{H}$ discussed in Section 2. The leaves in the HCML hierarchy are the application processes, and the internal nodes (corresponding to locales in $\mathcal{H}$) are HCML proxy processes called simply *proxies*. There is no proxy corresponding to the root of $\mathcal{H}$. In the degenerate case of a single locale, the only processes are the application processes, and HCML degenerates to SCML.

An application process can directly send messages to other application processes within its immediate locale and to the one proxy associated with that locale which acts as a surrogate for all of the other application processes outside of the locale. Proxies operates similarly to other processes: each proxy has a set of sibling processes with which it can communicate directly. To communicate with any non-sibling, the proxy forwards a message to *its* proxy.

The routing of messages is done automatically by the communication layer and is invisible to the application. This can be done efficiently when the locales are defined in terms of IP subnets.

3.3 Peers and Proxies

Each proxy in the system simultaneously runs two causal message logging protocols: one with its siblings and parent in the role of a *peer*, and one with its children in the role of a *proxy*. Since application processes are at the leaves of the hierarchy and have no children, they only run one causal message logging protocol with their siblings and parent in the role of a peer. Hence, for a hierarchy containing i internal nodes, there are i distinct protocols running at any time. We call this basic causal message logging protocol CML, and we associate a CML protocol with each proxy in the system. Thus proxy x runs both $\mathrm{CML}^{\hat{x}}$ with its siblings and parent and CML^{x} with its children. Application process s only runs $\mathrm{CML}^{\hat{s}}$.

CML is SCML with two differences: proxies have access to the determinants that are piggybacked on messages and proxies do not generate determinants. The first difference is needed for coupling instances of CML; as for the second, it is not necessary for proxies to generate determinants because their state can be safely reconstructed even if their messages are delivered in another order during recovery.

To satisfy the Causal Logging property, a proxy x couples CML^{x} and $\mathrm{CML}^{\hat{x}}$. Process x acts as a proxy to all of its children for all processes outside of its locale. Therefore, all determinants stored in $\mathbf{H}_x^{\hat{x}}$ (that is, the determinant array of process x associated with protocol CML$^{\hat{x}}$) and assigned to remote processes are also stored in $\mathbf{H}_x^x$ (the determinant array of process x associated with protocol CMLx and assigned to process x: it is always the case that

$$\forall r, d: \ (d \in \mathbf{H}_x^{\hat{x}}[r] \Rightarrow d \in \mathbf{H}_x^x[x]). \tag{1}$$

Process x also acts as a proxy to the processes in its peer group for its child processes. Therefore, determinants stored in $\mathbf{H}_x^x$ are also stored in $\mathbf{H}_x^{\hat{x}}$: it is always the case that

$$\forall r, d: \ (d \in \mathbf{H}_x^x[s] \Rightarrow d \in \mathbf{H}_x^{\hat{x}}[x]). \tag{2}$$

We call the conjunction of Equations 1 and 2 the *Coupling invariant.*

It is easy to see that the Coupling invariant combined with CML satisfies the Causal Logging property. Consider a message m sent from application process a to application process b. Let $T = \langle t_1, t_2, \ldots, t_k \rangle$ be the sequence of proxies that lead from a to b via $\mathcal{C}(a, b)$; thus, $t_1 = \hat{a}$, $t_k = \hat{b}$, and $t_{k/2+1} = \mathcal{C}(a, b)$. Due to the routing of messages through the proxies, m is forwarded via CML^{t_1}, CML^{t_2}, ... CML^{t_k}. For the protocols CML^{t_1}, CML^{t_2}, ... CML$^{t_{k/2+1}}$ Equation 2 ensures that the determinants needed to satisfy the Causal Logging property are available and forwarded appropriately. For the remaining protocols CML$^{t_{k/2+2}}$, CML$^{t_{k/2+3}}$, ... CML^k Equation 1 ensures that the determinants needed to satisfy the Causal Logging property are forwarded.

3.4 Recovery

When a process crashes, a new process must be created as a replacement for the failed process. In order to maintain consistency, causal message logging protocols gather the relavent recovery information from the set of processes (or proxies) and use it to ensure that the recovered process is consistent. Existing recovery protocols (see [3]) can be easily adapted for HCML.

HCML, however, has additional processes to maintain over those of standard causal message logging protocols, namely the proxy processes. Fortunately, a crashed proxy can simply be restarted and its neighbors in the heirarchy need to be informed. The proxies state includes only the cached recovery information, and subsequent messages will simply refill the cache after recovery.

4 Performance

Using the proxy hierarchy ensures that no process needs to track the causality of a large number of processes. This technique provides an exponential space reduction as compared to tracking the full causality. For example, assume that the locality hierarchy has depth of five and the fanout is 10 at each node. Such an architecture can accommodate $100,000$ application processes, yet each process only tracks either six or twelve processes (depending on whether it is an application process or a proxy respectively). With SCML, on the other hand, each application process would maintain a dependency matrix with $10^5 \times 10^5 = 10^{10}$ entries.

However, the tradeoff is that HCML will over-estimate the causality as compared to SCML and and more often needlessly piggyback determinants to processes that are not dependent on them. In addition, HCML will send more messages than SCML because all non-local messages are relayed through the proxy hierarchy. This, however, is offset by the fact that the proxies act as local caches for determinants. This caching of determinants reduces the overall message overhead by over 50% percent. More importantly, HCML reduces the message overhead over slower communication channels and reduces the effective message communication latency.

In this section, we first describe our application, the proxy hierarchy, and the scheduling of processes within the hierarchy. We then discuss the performance results.

4.1 Effect of the Hierarchy

In order to gauge the effect of the hierarchy on both the message overhead and the message cost for HCML and SCML, we analyzed the performance of an application of 256 processes where, on average, each process communicates with four other processes selected randomly. The application proceeds in rounds. At each round, each process sends a message to its neighbors and delivers the messages sent in the previous round. The run ends after approximately 5,000 messages have been delivered.

An execution completely defines a run, but the performance of the run using HCML depends on the structure of the hierarchy and on how the processes are scheduled in the hierarchy. We then considered proxy hierarchies of different depths:

1. A depth-one hierarchy consisting of one locale containing all 256 application processes and no proxies. As stated earlier, this is identical to SCML.
2. A depth-two hierarchy with four locales (hence four proxies), each containing 64 application processes.
3. A depth-three hierarchy with sixteen application processes per lowest level locale. Their proxies have three siblings each, and so there are 20 proxies total.
4. A depth-four hierarchy that divides each of the application process locales of the previous hierarchy by four. Thus, there are four application processes per lowest level locale, and there are 84 proxies total.

The application processes are placed into locales independently of the communication patterns that they exhibit.

We used the Network Weather Service [16] to measure the available bandwidth for processes communicating in different locales. The values we measured ranged from over 200 MB/s for communication within the locale of a simple workstation to less than 0.4 MB/s for the wide area locale containing San Diego and western Europe. Thus, we set $BW_1 = 1MB/s$ (intercontinental communications), $BW_2 = 10MB/s$ (intra-stub domain communications), $BW_3 = 100MB/s$ (local area network communications), and $BW_4 = 1,000MB/s$ (intra-high performance multiprocessor communications). [1] Figure 1 shows the total message size for the run using both HCML and SCML as a function of depth. Because SCML does not take advantage of the locale hierarchy, its performance is constant with respect to the depth. HCML, on the other hand, relays non-local messages through the hierarchy and therefore sends more messages overall. Hence, one might expect that HCML would have a higher total message overhead. As the figure shows, the caching of the determinants actually improves the message overhead of HCML over SCML by as much as 50%. As the hierarchy gets deeper, the net effect of the caches is reduced. For a depth of four, for example, the locales at depth 3 have only four processes in them each and so the opportunity to benefit from caching is low.

To see how the caches reduce the communication costs, consider the example from the last section once again. After m_2 is finally delivered to process u, the determinant for message m_1 is stored at the intermediate nodes p and r as well as the application process u. Consider what happens if a third message m_3 is sent from process t to process v. In SCML t simply piggybacks $\#m_1$ on m_3 which gets sent from the locale of p to the locale of r. Using HCML, however, m_3 is redirected to node p which knows that r already has a copy of $\#m_1$. Therefore p does not need to piggybacked $\#m_1$ again. Process r does not know whether v has stored $\#m_1$, and hence piggybacks the determinant to v.

A secondary effect of the proxies is that more of the communication occurs lower in the hierarchy, where there is more bandwidth available. Figure 1 also shows the total transit overhead for SCML and HCML. In the case of depth 3 hierarchy, HCML reduces the total transit overhead by 97%. It should be noted that this metric does not include any added latency arising from the processing time of proxies.

We have found similar results for different application with different communication properties. In most cases, HCML is able to leverage the locality and produce a net reduction in both the total message overhead and the total message transit time. In addition, HCML performs better when the communication pattern the application processes use biases communication to be mostly within the higher bandwidth locales. Hence, we believe that HCML can only benefit from the the careful scheduling of grid-based applications.

---

[1] For hierarchies of depth less than three, we assigned bandwidths starting with BW_1. While doing so is unrealistic—for example, one would not expect a program to consist of 256 processes, each running in its own stub domain—it is at least well defined and no less arbitrary than any other choice. Furthermore, doing so does not affect the relative total transmission overheads for a fixed depth.

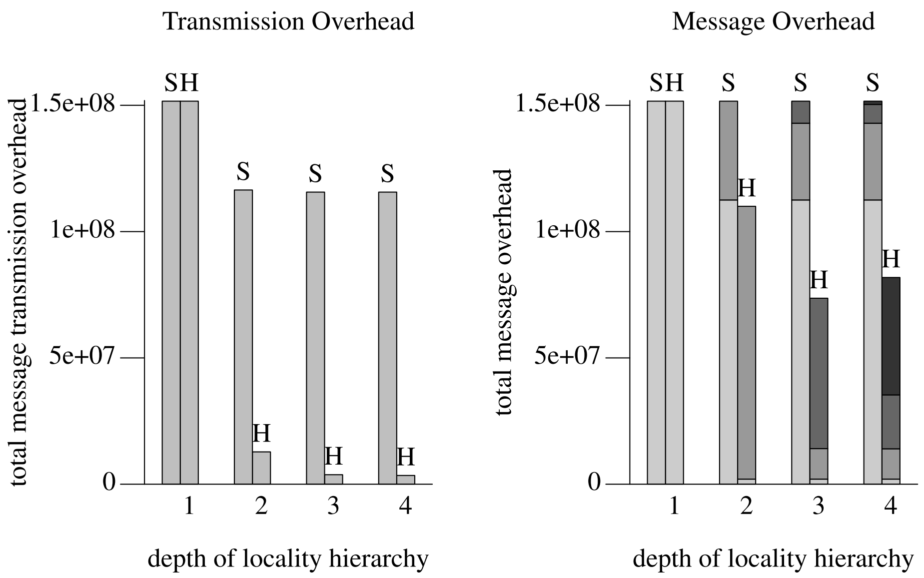

Fig. 1. HCML (H) and SCML (S) performance. Darker areas represent bandwidth overhead sent between locales lower in the hierarchy, and hence locales with greater available bandwidth.

5 Conclusions

We have developed a scalable version of causal message logging. Our preliminary measurements indicate that it can easily scale to the largest grid-based computing environments that are being envisioned. Not only are the data structures that are maintained by each application process reduced by an exponential amount, but a caching effect reduces the message overhead as well when compared to traditional causal message logging. To attain these benefits, one sets up a hierarchy of proxies, each serving both as a router of causal message logging communication and as a cache of recovery information. Indeed, an interesting open question is if the routing of fault-tolerant information could be implemented as part of the underlying network routing function.

The protocol as described here is very simple, and appears to be amenable to refinement. For example, each proxy p manages an instance $\mathrm{CML}^{\hat{p}}$ of a causal message logging protocol. It seems straightforward to allow $\mathrm{CML}^{\hat{p}}$ to be replaced with a pessimistic message logging protocol to limit the spread of recovery information to be below p in the locale hierarchy. Another refinement we are developing would allow one to give specific failure model information about locales, thereby allowing one to replicate recovery information more prudently.

One spreads recovery information for the purpose of recovery, which is not discussed in any detail in this paper. In fact, we have designed HCML to allow us to experiment with recovery in the face of partitions. HCML does not appear to be hard to extend to support dynamic replication of a process (or an object) when a partition makes it inaccessible to a set of clients that require its service. The approach we are developing has some similarities with other dynamic replication services and with wide-area group programming techniques [10, 15].

References

1. M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
2. L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd Fault Tolerant Computing Symposium*, pages 145–154, June 1993.
3. L. Alvisi and K. Marzullo. Trade-offs in implementing causal message logging protocols. In *Proceedings of the 15th ACM Annual Symposium on the Priciples of Distributed Computing*, pages 58–67, Jan 1996.
4. L. Alvisi and K. Marzullo. Message logging: pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, Feb. 1998.
5. D. Anderson and D. Werthimer. The seti@home project. http://setiathome.ssl.berkeley.edu/.
6. K. Bhatia, K. Marzullo, and L. Alvisi. Scalable causal message logging for wide-area environments. Technical Report CS2001-0671, University of California–San Diego, May 2001.
7. E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback recovery protocols in message passing systems. Technical Report CMU-CS-99-148, CMU, June 1999.
8. E. Elnozahy and W. Zwaenepoel. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.
9. I. Foster and C. Kesselman. The Globus project: a status report. In *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 4–18, 1998.
10. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 1992.
11. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
12. H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. The top 500 supercomputers list. http://www.top500.org/.
13. F. Previato, M. Ogg, and A. Ricciardi. Experience with distributed replicated objects: the Nile project. *Theory and Practice of Object Systems*, 4(2):107–115, 1998.
14. R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proceedings of the Eighteenth Annual International Symposium on Fault-Tolerant Computing*, pages 44–49, June 1988.
15. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 172–183, Dec. 1995.
16. R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, Oct. 1999.