

Adaptive Timeout Discovery using the Network Weather Service*

Matthew S. Allen
Computer Science Department
University of California, Santa Barbara

Rich Wolski
Computer Science Department
University of California, Santa Barbara

James S. Plank
Computer Science Department
University of Tennessee, Knoxville

In this paper, we present a novel methodology for improving the performance and dependability of application-level messaging in Grid systems. Based on the Network Weather Service, our system uses non-parametric statistical forecasts of request-response times to automatically determine message timeouts. By choosing a timeout based on predicted network performance, the methodology improves application and Grid service performance as extraneous and overly-long timeouts are avoided. We describe the technique, the additional execution and programming overhead it introduces, and demonstrate the effectiveness using a wide-area test application.

1 Introduction

The challenge for the Computational Grid is to extract reliable, consistent performance from a fluctuating resource pool, and to deliver that performance to user applications. Dynamic schedulers [12, 14, 2] have been able to “steer” resource usage automatically so that the application can take the best advantage of the performance that is available. To do so, many rely on short-term, statistical *forecasts* of resource performance generated by the Network Weather Service (NWS) [17, 16] – a Grid performance data management and forecasting service.

While application-level schedulers work well in Grid contexts, to date, they have had to rely on intrinsic “knowledge” of the application being sched-

uled. That is, they are customized software components that exploit application-specific characteristics to mitigate the negative performance effects of Grid dynamism. Tools such as the AppLeS Parameter Sweep Template [6, 5] generalize this approach to incorporate information that is common to a class of applications. Little work, however, has focused on the use of dynamic techniques to optimize the performance of *Grid services* that application schedulers and Grid users require.

One such Grid Service that is critical to application performance is communication. Grid applications must be designed to run on a wide area network, which poses a difficult performance problem. Communication times in such an environment fluctuate dramatically as routers become overloaded and drop packets, networks partition, and congestion control algorithms restrict communication. Many conditions can cause a transmission in progress to take much longer than expected. One way to deal with this problem is to time out communications that exceed some expected elapsed time. Choosing this value, however, is often little more than an educated guess.

When network performance is fluctuating, timeout determination can dramatically affect both application and service performance. Timeouts that are premature cause either needless failures or extra messaging overhead as the message is retried. Overly long timeouts result in degraded messaging performance as communication is delayed unnecessarily. Our methodology attempts to “learn” the best timeout value to use based on previously observed communication between host pairs. By combining pre-

*This work was supported, in part, NSF grants EIA-9975015, ACI-9876895, and CAREER award 0093166.

dicted message send-acknowledge times with NWS forecasting error measures, our system sets timeouts adaptively, in response to fluctuating network performance.

In this paper we present a general technique for improving communication times on the wide-area. Our system “predicts” communication times using non-parametric statistical forecasting techniques. These predictions are then used to determine an appropriate application level timeout for each send. By canceling and reissuing sends that exceed this time limit, we are able to achieve improvements in communication. This methodology’s benefits are realized in a general and application independent way.

We used a sample application to present our results. For this application, we chose a global data reduction across multiple hosts in the Grid Application Software Development (GrADS) [1] testbed. GrADS is a Grid software research project studying the problem of application development and deployment. From some of its early results [12], it is clear that the performance of the Grid services themselves is a critical component of Grid application performance. If the services respond slowly, the application using them will also suffer a performance degradation. Our choice of test application in this work is motivated by two observations. First, collective operations [7, 9, 8] are a performance critical component of many distributed scientific and engineering applications. Secondly, we observe that frequently, Grid users and schedulers must collect up-to-date information from widely distributed sites, often to pick out the “best” choice from a number of possibilities. For example, application schedulers often wish to determine the most lightly loaded resource(s) from a specified resource pool when making scheduling decisions. While it may be possible to avoid scheduling collective operations in a scientific application so that they span wide-area network connections, it is not generally possible (or desirable) to centralize Grid services in the same way.

As such, this paper will make two significant contributions. It will:

- describe an application-independent, high-performance, and simple methodology for adaptive timeout discovery using NWS forecasting tools
- demonstrate the effectiveness of this approach in a “live” Grid setting.

The rest of this paper is organized as follows. In the next section, we will discuss the network conditions we hope to improve on with this technique. In Section 3 we will describe the adaptive timeout discovery methodology. Section 4 describes the specific set of experiments we conducted. In Section 5 we discuss the results of our investigation, and Section 6 draws our final conclusions from this investigation.

2 Network Behavior

The motivation for this research is the observation that there is a high degree of variance in communication times, especially in wide-area networks. Send times have been observed to loosely follow either a heavy-tailed or Poisson distribution [10]. Although there is some debate over which accurately describes actual network traffic, either case indicates that there can be extremely long send times. Furthermore, the tail-end sends are not necessarily clustered together, but often occur among more reasonably behaved communications [13]. Thus, while the send time across a link will usually take only a couple of seconds, it may occasionally take significantly longer.

Most of these delays stem from the behavior of routers at one of the hops in the transmission. For example, a temporarily overloaded router may drop packets or accidentally direct them to the wrong destination. This is particularly a problem when a TCP transmission collides with a poorly behaved UDP stream that fills up a routing queue [3]. Although overloaded routers are likely to provide consistently poor performance, other routers before them in the communication path may detect this congestion and redirect messages to another path.

Another significant problem is that routers occasionally change their routing tables because of policies or configurations and not detection of a congested network. Sometimes this happens mid-transmission, causing packets to arrive out of order and causing other delays in the communication [11]. This is particularly evident in a phenomenon known as “fluttering” where a router rapidly switches between different paths to direct packets. Where a few packets may follow a 12 hop route, the next few may follow one with 25 before the router returns to its original route. This change can occur with every other packet in the worst cases [11].

It is also possible that the TCP/IP congestion con-

ily apparent.

To generate results appropriate for a wide area network, we use a set of hosts that is distributed across a large number of networks. We arrange these hosts so that few of the links in the tree are between hosts in the same domain. This is obviously a contrived situation, and no optimal distribution would cross domain boundaries in this way. However, we intend for this experiment to be representative of large Computational Grid programs where it is likely that hosts from many domains will be employed. The host distribution we have chosen represents the wider scattering we assume will be necessary for larger codes. The distribution of the hosts to which we had access for our experiments can be seen in *figure 1*.

Every host in the tree maintains a timeout value associated with each link to another hosts Each send call in the reduction is followed by an explicit, application-level acknowledgement. If the send does not complete or the acknowledgement is not returned before this time, the send is canceled. In the event of a cancellation, the host stops the read or write, calls *close()* on the socket, make a new *connect()* call, and resends the data. It retries the send in this manner a finite number of times, and if this number is exceeded, the reduction is abandoned.

The experiment is run using several different strategies for setting the timeout values, broadly categorized as either static or dynamic. When a static timeout is used, each host uses the same timeout value for all of its links. This situation corresponds to the current implementations of distributed applications as we know them. Our experiment runs with static timeouts of two, five, fifteen, and forty-five seconds to capture a spectrum of static values. In the wide area, where round-trip latencies can be large, a two-second timeout is an eager approach. On the other hand, forty-five seconds is a very conservative estimate. We believe that any performance-oriented application would likely wish to have a retry initiated if this time limit is exceeded.

When dynamic timeouts are used, each host keeps a distinct timeout value for each of its links. These values are determined automatically throughout the experiments execution as we described in the previous subsection. Because the square root of the mean square error is the standard deviation of the distribution, we chose to have one set of dynamic experiments use two factors of this value. Although we do not believe that the distribution of send times follows

a normal distribution, we wanted to to test the effect of timing out on the longest of the sends. We also tested it with one factor of the standard deviation for a more aggressive dynamic strategy.

Over the application’s lifetime, the NWS adaptive timeout changes as the network performance response changes. Our experimental results compare the effect of using various static timeouts to the NWS dynamic timeout discovery technique on overall application performance.

5 Results

We ran the experiment described in the previous subsection over a week period in late January of 2002. Run start times were distributed throughout the day so that the results would span potential daily cycles of network congestion. The experiment was run in 10 sets of 100 reductions each using the static and dynamic timeout values we described above. Nodes attempted to resend data 5 times before giving up on the reduction altogether. The calculations were done on buffers of 16k integers, for a send size of 64kb. For each reduction, we calculated the *reduction time* to be the max elapsed time among all nodes between sending its data to its parent and receiving the final reduction.

Figure 2 compares statistics gathered over all reductions as a function of timeout method. For each method, we show the maximum, the 95th-percentile, the average, and the minimum of all reduction times.

We note that only a couple of timeouts actually occurred during the 45 second timeout run. As such, the network connections and hosts involved rarely “failed” during the experiment and the typical TCP/IP reliability mechanisms were able to make progress. In particular, we used the TCP KEEP_ALIVE mechanisms with their default settings which will cause a timeout and a reset of the socket after some long, configuration-specific time period. In the experimental data, the KEEP_ALIVE timers never caused a reset. The effect we illustrate, then, is a performance effect in the sense that a timeout and retry of a connection changes the performance characteristics (and not the reliability) of the application. This observation surprised us since the users of these systems (primarily members of the GrADS [1] project) often report intermittent network failures. Nonetheless, the 45 second timeout measure-

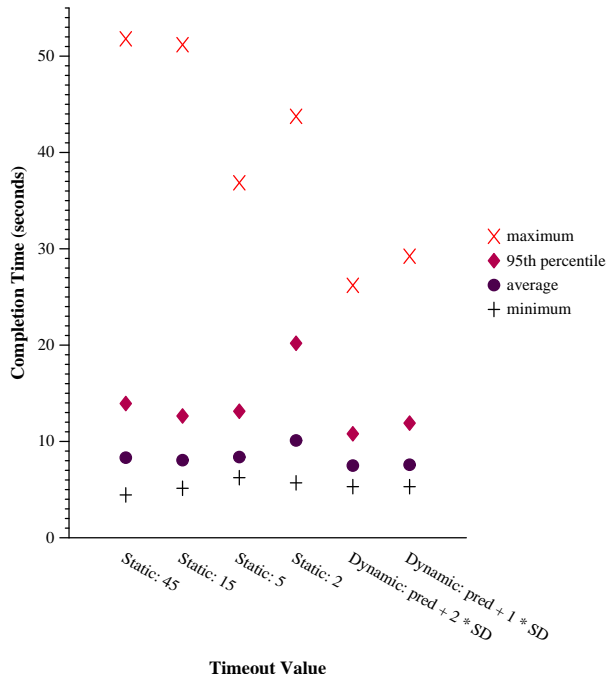


Figure 2: Statistics of reduction times

ments represent the “non-failure” case with default TCP KEEP_ALIVE timeouts in use.

Furthermore, we can see from *figure 2* the other static timeout results that the 15 second values shows the best performance improvement in all metrics the maximum time. We have not yet looked at the distribution of maximums for this data so we cannot report on the significance of particular difference. Since the other metrics are the best among the statically determined timeout values, we will consider the 15-second timeout the most performance effective of the static methods.

Lastly, we can see the dynamic predictions using two standard deviations is clearly the best of the dynamic timeout methods, and represent the best predicted timeout.

5.1 Performance Improvements

The most obvious benefit of overriding the default TCP/IP timeout mechanisms is a decrease in the average completion times. As we can see from *table 1*, the average reduction time improves slightly with even the generous 15-second timeout, and dynamic timeout prediction causes a more substantial

Table 1: Characteristics of reduction times (in seconds)

	45 Second	15 Second	Dynamic
Mean	8.317	8.056	7.495
Max	51.796	51.187	29.230
Min	4.447	5.129	5.303
95th Percentile	13.942	12.642	10.794
Variance	7.888	5.261	2.722
Standard Deviation	2.809	2.297	1.650

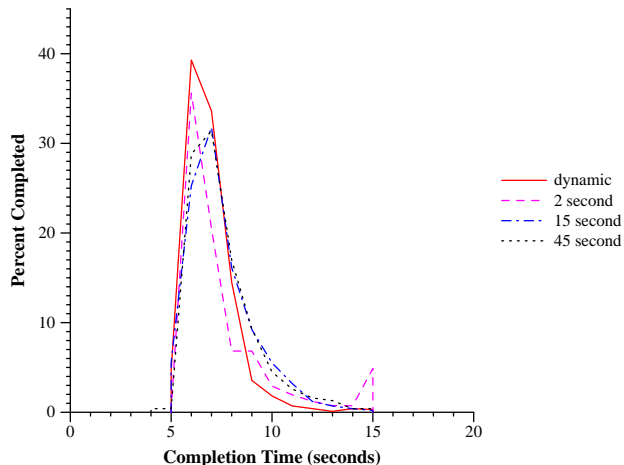


Figure 3: Distribution of reduction times

improvement. These levels of improvement suggest that the average performance with no timeouts is affected by outlier sends that take well above the average amount of time.

However, this improvement in average completion time is not overwhelming, and is not the most significant performance improvement this strategy offers. It is significant that dynamic timeouts yield an impressively lower *variance* in completion time than static ones. Not only is the variance lower (as seen in *table 1*) but the 95th-percentile and maximum values are substantially lower. Perhaps not surprisingly, then, dynamic timeout discovery based on NWS predictions yields more *reliable* and *predictable* performance than static methods.

From *figure 3* we can see the distribution of completion times for the sends completing in less than 15 seconds. Comparing the NWS dynamic timeouts with 15-second and 45-second static timeouts, it is clear that a greater fraction of the overall execution times are shorter using the NWS method than ei-

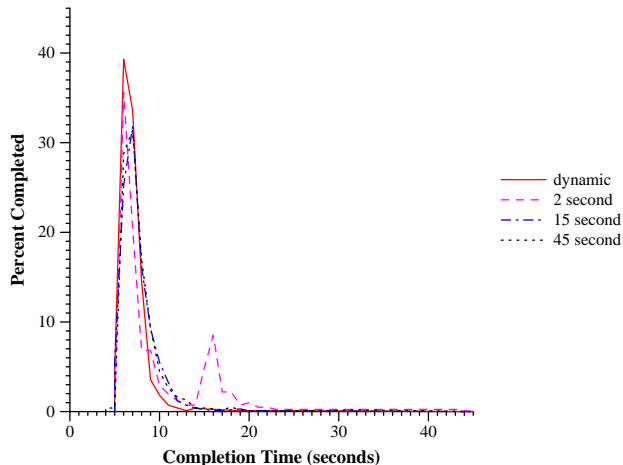


Figure 4: Extended distribution of reduction times

ther static method. The 2-second value, however, exhibits an interesting modality. A greater fraction of the execution times are shorter for 2-second timeouts, *considering only those executions that are less than 8 seconds*. However, a substantial fraction of execution times for the 2-second case are longer than 8 seconds yielding a lower aggregate performance. *Figure 4* shows the tail end of this distribution. Note the large mode in the 2-second data at approximately 16 seconds. Retrying too quickly can pay benefits, but it also induces an additional mode that is significant enough to reduce the overall performance benefit. The NWS dynamic timeout, however, generates the smoothest and most narrow range of performance values (note the NWS curve is highest indicating the largest fraction of consistently similar values). Clearly the NWS predicted timeout values yield the highest overall performance and the most predictable execution time.

5.2 Adaptability

The other significant advantage of our adaptive timeout mechanism is its ability to learn network conditions. Using this non-parametric time series analysis, we are able to choose appropriate timeout values for each connection maintained by a given node. *Table 2* shows us that although dynamic timeouts may retry connections more often than static versions, they usually only retry once. This means that the dynamic method quickly discovers an appropriate

Table 2: Timeout characteristics

	Percent Retried	Avg. # of Retries
45 Second	0.050%	1.000
15 Second	0.312%	1.380
5 Second	2.513%	1.542
2 Second	45.538%	2.638
Mean + 2SD	9.056%	1.176
Mean + 1SD	20.851%	1.241

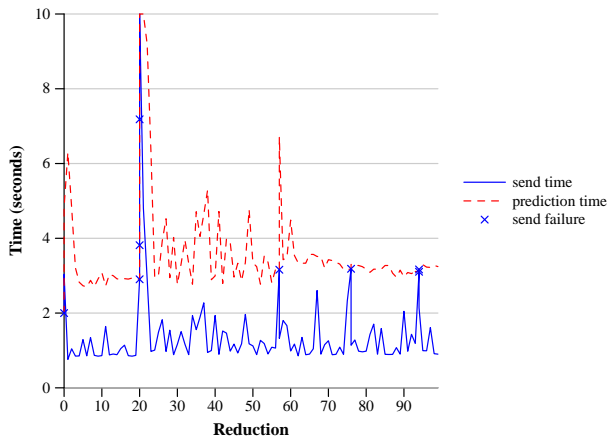


Figure 5: Send times and timeout predictions from liz.eecs.harvard.edu to torc7.cs.utk.edu

timeout value for the link.

We can see this effect more clearly from *figure 5*, which shows the actual send time and the timeouts based on NWS network performance forecasts for a run of 100 reductions across one link. Notice from this graph that early executions cause a lot of instability in the NWS forecaster as it “learns” the performance response of an individual network link. However, once a reasonable amount of data has been gathered, the forecaster is not as susceptible to delayed sends. From this point on, it only retries those sends that occur at obvious spikes in the execution time curve.

Another advantage of our dynamic timeout mechanism is the ability to automatically respond to the different network characteristics of each link. *Figure 6* presents the send times and NWS forecast-based timeouts for two of the links used by a node at dsi.ncni.net. The performance of these two links differs significantly enough that the timeout values for the slower link would result in an unnecessary number of timeouts. To accomplish this with static

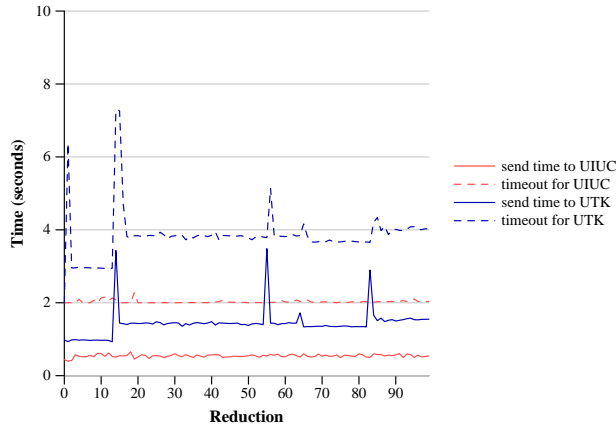


Figure 6: Send times and timeout predictions from dsi.ncni.net to cmajor.cs.uiuc.edu and torc2.cs.utk.edu

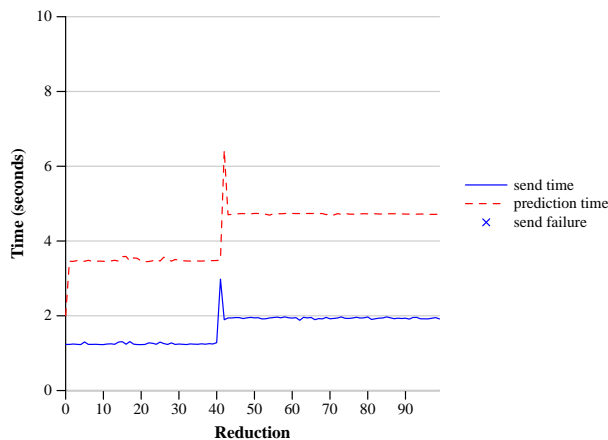


Figure 7: Send times and timeout predictions from dsi.cs.utk.edu to charcoal.cs.ucsb.edu

timeouts, we would have to manually select a timeout value appropriate to each link in each node. This is virtually impossible to do manually, but is simple when handled automatically.

One final strength of using dynamic timeouts is the ability to respond to actual changes in network conditions. Although individual anomalies will not cause the forecaster to become unstable, changes in network conditions are detected fairly quickly. In *figure 7* we can see the effects of a change in network behavior on the predicted timeout values. By noticing such changes and choosing an appropriate new timeout value, we retain the advantages of manually setting timeout without the danger of hurting our ap-

plication performance by timing out too soon.

6 Conclusion

In a widely distributed environment like the Computational Grid, network performance is an important aspect of the overall application performance. This is especially true in the wide area, where network behavior can fluctuate heavily over an application’s run time. Grid services must be aware of this problem, and respond to both improve communication speed and make send times more dependable.

In this paper, we present a simple and lightweight method to accomplish this in a generic way. We contrived a situation to test its ability to improve performance on a network with the characterizing of a Computational Grid. Our application took the form of a collective operation; a common and expensive service of distributed applications. We were able to both increase the average completion times of the reductions, as well as control the variance in the send times so that reductions finished in a more predictable time quantum. Furthermore, the overhead required to do this was minor, and did not offset the improvements we realized.

Of course, there is much work left to be done on this topic. One area that warrants further examination is the effects of this methodology on different send sizes. It is likely we will not achieve any performance improvement when sending sufficiently large or small amounts of data. There is likely also a send size that takes the greatest advantage of the adaptive timeouts. If this is the case, then the possibility that breaking large sends into small send-acknowledgement pairs will increase performance should be investigated.

Furthermore, the algorithm that we used to determine timeouts was chosen primarily for its simplicity and speed. Also, in the course of the experiment we realized that over fast connections this formula produced timeouts so low as to timeout frequently and hurt performance. To prevent this problem, we manually chose a static minimum timeout value to use. This formula worked very well for the experiment, but it was fine tuned manually based on problems we observed. This suggests that there may be an algorithm that is able to generally predict a more appropriate timeouts and possibly produce a more pronounced improvement.

Finally, it would be good to see how this methodology affects the performance of different distributed applications. We chose to test this with a binary reduction because its heavy dependence on successive communications made it easy to test our technique. However, this operation is not indicative of all of the operations common to applications in this domain. Whether other operations could benefit from adaptive timeouts needs to be more thorough explored.

References

- [1] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, L. J. Dennis Gannon, K. Kennedy, C. Kesselman, D. Reed, L. Torczon, , and R. Wolski. The GrADS project: Software support for high-level grid application development. Technical Report Rice COMPTR00-355, Rice University, February 2000.
- [2] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.
- [3] S. Floyd and K. R. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.
- [4] M. P. I. Forum. Mpi: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, 1994.
- [5] D. Z. H. Casanova, A. Legrand and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing workshop*, pages 349–363, 2000.
- [6] F. B. H. Casanova, G. Obertelli and R. wolski. The apples parameter sweep template: User-level middleware for the grid. In *Proceedings of Supercomputing 2000*, 2000.
- [7] T. Kielmann, H. Bal, S. Gorlatch, K. Verstoep, and R. Hofman. Network performance-aware collective communication for clustered wide area systems. *Parallel Computing*, 27:1431–1456, 2001.
- [8] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. Magpie: Mpi’s collective communication operations for clustered wide area systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’99)*, pages 131–140, 1999.
- [9] B. Lowekamp and A. Beguelin. Eco: Efficient collective operations for communication on heterogeneous networks. In *10th International Parallel Processing Symposium*, 1996.
- [10] V. Paxson and S. Floyd. Why we don’t know how to simulate the internet, 1997.
- [11] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997.
- [12] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical libraries and the grid. In *(to appear) Proc. of SC01*, November 2001.
- [13] J. Plank, R. Wolski, and M. Allen. The effect of timeout prediction and selection on wide area collective operations. In *The IEEE International Symposium on Network Computing and Applications*, February 2002.
- [14] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing 1998*, July 1998.
- [15] A. Veres and M. Boda. The chaotic nature of TCP congestion control. In *INFOCOM (3)*, pages 1715–1723, 2000.
- [16] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1:119–132, 1998. also available from <http://www.cs.ucsb.edu/~rich/publications/nws-tr.ps.gz>.
- [17] 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, October 1999. <http://www.cs.ucsb.edu/~rich/publications/nws-arch.ps.gz>.