The Effect of Timeout Prediction and Selection on Wide Area Collective Operations

James S. Plank Rich Wolski Matthew Allen

Department of Computer Science
University of Tennessee
Knoxville, TN 37996-3450
[plank,rich,allen]@cs.utk.edu

Technical Report CS-01-457
University of Tennessee
Department of Computer Science
March 27, 2001

For the the publication status of this paper and papers derived from it, see:
http://www.cs.utk.edu/~plank/plank/papers/CS-01-457.html

1 Introduction

The current trend in high performance computing is to extend computing platforms across wider and wider areas. This trend is manifested by the national efforts in Computational Grid computing [2, 6, 7, 8, 12], and commercial endeavors such as Entropia [5], Parabon [11], and United Devices [13]. As computational processes attempt to communicate over wider areas, the need to identify and tolerate failures greatens, and most communication software packages, initially developed for tightly coupled computing environments (a notable example is MPI [9]), do not deal with exceptional or faulty conditions smoothly.

Most wide-area communication libraries base their message-passing on TCP/IP sockets. TCP/IP has been designed to keep most networking applications functioning smoothly in the face of link and node failures, and message congestion. However, its failure semantics as presented to a message-passing library are limited, and high-performance applications often have a difficult time running robustly in the presence of failures.

In this paper, we address one important exceptional condition: identification of network failures using timeouts. We first discuss why this is an important problem, and the typical *ad hoc* solutions to it. We then propose a timeout identification method that uses online monitoring and prediction to set timeout values and use them for failure identification. We then show results of a primitive wide-area application that explores the implications of setting timeouts in both simple ways and using an adaptive technique based on the Network Weather Service [16].

2 The Problem of Failure Identification

One fact of computing on the Internet is that failures do occur. The failure with which we will concern ourselves is one where a socket connection is no longer valid. This appears in a running process as one of the two following scenarios:

- The process is performing a **read()** operation on the socket, and the read will never complete because either the writer is gone, or the corresponding **write()** operation has been partitioned away from the reader.
- The process is performing a **write()** operation on the socket, and the bytes will never get to the reader.

For a wide-area application to be successful, these two scenarios must result in the identification of a failure so that the application may deal with it appropriately. The methods of dealing with the failure are beyond the scope of this paper. They may involve aborting the program and starting anew, attempting a reconnection of the socket to retry the communication, or perhaps performing rollback recovery to a saved state so that the loss of work due to the failure is minimized [1, 3, 4]. No method of dealing with the failure will be successful, however, unless the failure is properly identified.

The default failure identification method in TCP/IP sockets is a method of probing called “keep-alive.” At regular intervals, if a socket connection is idle, the operating system of one side of the socket attempts to bounce a packet (typically a packet with a previously acknowledged sequence number) off the other side. If an acknowledgement response is not forthcoming within a preset threshhold time, then it is assumed that there has been a failure, and that the connection should be broken. Reads from or writes to that connection now fail, and the application may take appropriate steps.

We term this threshhold time a *timeout*. Unfortunately, this default keep-alive mechanism is not of great practical use. The main reason is that the timeout values are typcially not user-settable. They differ from machine to machine or operating system vendor to operating system vendor, and they tend to be arbitrarily chosen. Thirty seconds is common, although IETF RFC 1122 specifies only that the keep-alive time be less than 120 minutes (two hours) by default [10]. Keep-alive timing is typically set when the operating system is configured. Its primary use to to allow server processes to reclaim used network state if clients quietly disconnect. For performance-oriented applications that use TCP sockets for interprocess communication, the standard keep-alive values can cause serious performance degradations.

It is possible to turn off keep-alive probing on a socket. Indeed, by default on most Unix systems, sockets are not conditioned to use keep-alive. When keep-alive is disabled, the first scenario above (a read not being satisfied) is never identified by the operating system, since no probing is performed to see if the writer is alive. The second scenario is only identified when the reader shuts down the connection and its operating system explicitly refuses to accept packets from the writer. If the reader’s machine becomes unreachable, the writer simply attempts retransmission of its message until the kernel socket buffer fills or some unspecified retry limit is reached. The typical remedy is for the application itself to choose a timeout value for each connection, and to use the Unix signal mechanism to interrupt an indefinitely-blocked **read()** or **write()** call.

Thus, applications must choose one of two methods to perform failure identification – either use the default keep-alive probing with its limitations, or turn off keep-alive probing, and use some sort of heuristic to perform its own failure identification, typically by setting its own timeout values. Regardless, there are important tradeoffs to the selection of the timeout value. Large timeouts impose less stress on the network, but they greaten the latency of failure detection, which in turn reduces the performance of the applications that rely on accurate failure detection to proceed efficiently. Small timeouts impose more stress on the network, and additionally, they may be too aggressive in labeling a sluggish network as failed. For example, suppose an application processes for ten minutes between communication. If the network happens to be unavailable for a majority of that ten minute interval, it does not affect the application so long as the network is restored when the application needs to communicate. In such a situation, a small timeout value will impede application performance, as it will force the application to recover from a failure that is not preventing the application from proceeding.

3 Static vs. Dynamic Timeouts

There are two ways that an application may select timeouts – statically or dynamically. Obviously, static timeouts are simple to implement. However, they have two limitations. First, a static timeout value, especially if it is arbitrarily chosen, may not by ideal for an application and networking environment. Second, even if a static timeout starts out ideal, a changing network or application may result in its being not ideal. For this reason, we explore dynamic timeouts.

Dynamic timeouts require more complexity in implementation and also need to be chosen to have effective values. We propose to use monitoring and prediction to select timeouts. That is, a connection monitors its past communication parameters and predicts a timeout value that is most likely to result in a correct identification of a failure. We base our prediction on the Network Weather Service (NWS) [16].

The NWS uses an adaptive time-series forecasting methodology to choose among a suite of forecasting models based on past accuracy. Each model in the suite is used to predict a measurement that has been previously recorded. Since both the past values and the predictions of them are available, it is possible to chacaterize each forecasting method according to the accuracy of its forecasts in the past. At the time a forecast for an unknown value is required, the forecasting method

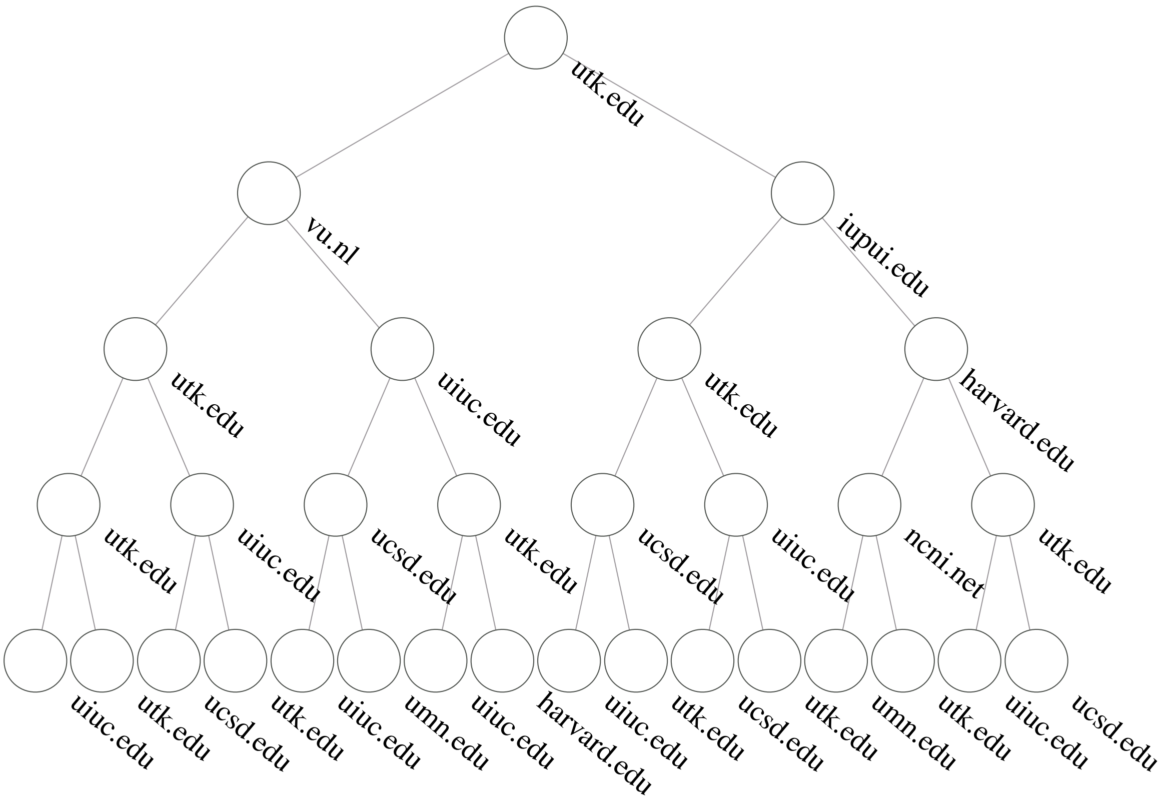


Figure 1: Topology for a 31-node tree-based reduction.

having the lowest aggregate error so far is selected. Both the forecast, and the aggregate forecasting error (represented as mean-square error) are presented to the client of the NWS forecasting API [14].

As such we can use the NWS to determine a timeout for each network connection based on previous response time readings. The NWS can forecast the time required to send a message and receive a response. The forecasting error provides a measure of how accurate this estimate is likely to be. By adding the error to the estimate, we have a "worst-case" (in terms of accuracy) estimate for the response time.

For example, if the NWS predicts that a message response will occur in 20 seconds, and the mean-square forecasting error is 16 $seconds^2$, then one can use the error deviation (the square-root of the mean-square forecasting error) as an estimate of how much "slack" to build into the prediction. The estimate plus two deviations (8 seconds in this example) would set the timeout at 28 seconds.

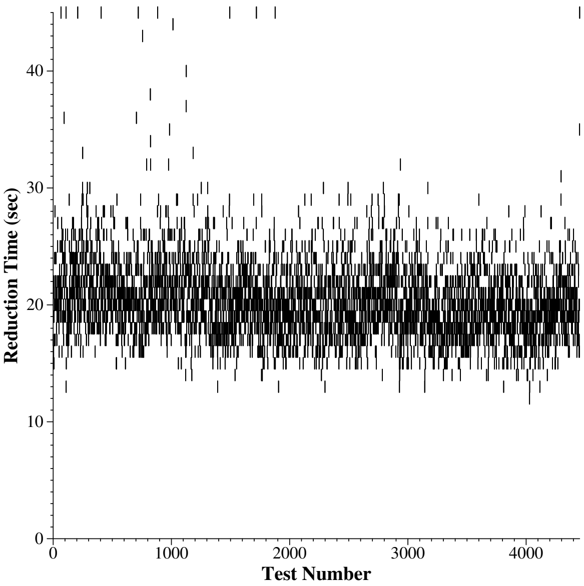
In order for this methodology to be successful, the timeout prediction must improve upon a static timeout selection in one of two ways:

- It correctly identifies a failure in a shorter time period than the static timeout.
- It correctly identifies a non-failure state by waiting longer than the static timeout.

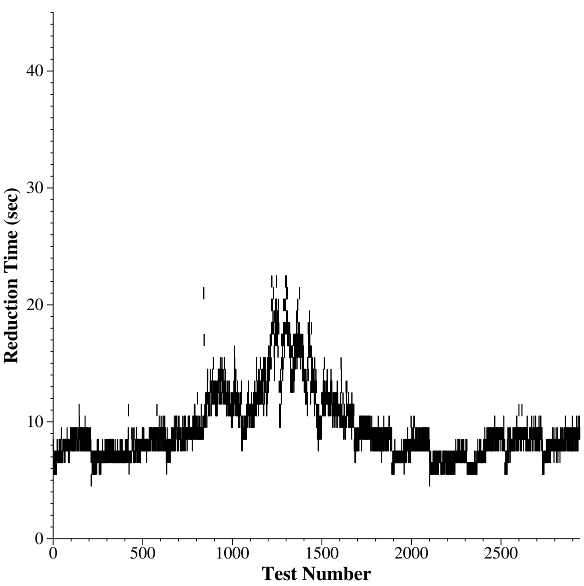
In the sections below, we use a very simple application on a wide-area network to explore this methodology.

4 Experimental Validation

We choose a simple application to test the effects of timeout determination. This is performing a maximum reduction on a set of values. Specifically, we have n computing nodes, each of which holds a set of m values. Let the j-th value of



Collection 1



Collection 2

Figure 2: Scatter plotp of reduction times as a function of test number for both 31-node collections.

node i be a^i_j. When the reduction is complete, a^i_j is the same for any fixed value of j, and is equal to the maximum value of a^i_j for all i.

We perform this reduction by arranging the nodes into a logical tree, an example of which is depicted in Figure 1. The reduction proceeds in two phases. In the first phase, values are passed up the tree from children to parents, calculating maximum values as they are passed. At the end of this phase, the root node r holds the proper values of a^r_j. The second phase is a broadcast of all the a_j down the tree. When the second phase is complete, each node i holds correct values of a^i_j.

While not all distributed applications use a methodology such as this, reductions (and similarly barriers) are common in parallel programming. For instance, many communication libraries (e.g. MPI) contain basic primitives for reductions. We do not claim, however, that the method of reduction we to validate this work is optimal. Rather, we use it to provide some insight into the importance of timeout tuning for distributed applications that include reductions.

To test the effect of timeout determination on the wide area, we used two collections of thirty-one machines. The first had the following composition:

- 11 machines from the University of Tennessee in Knoxville (labelled `utk.edu`).
- 8 from the University of Illinois at Urbana-Champaign (`uiuc.edu`).
- 5 from the University of California at San Diego (`ucsd.edu`).
- 2 from Harvard University (`harvard.edu`).
- 2 from the University of Minnesota (`umn.edu`).
- 1 from Vrije Univeristy in the Netherlands (`vu.nl`).
- 1 from the Internet2 Distributed Storage Project in Hawaii (`iupui.edu`).
- 1 from the Internet2 Distributed Storage Project in Chapel Hill, North Carolina (`ncni.net`).

Note, this is a collection of twenty-nine machines from the continental United States, one from Hawaii, and one from Europe. We arranged the machines into a five-level tree as depicted in Figure 1. We did not try to cluster machines from a single location, so that we more closely approximate a random collection of widely-dispersed machines.

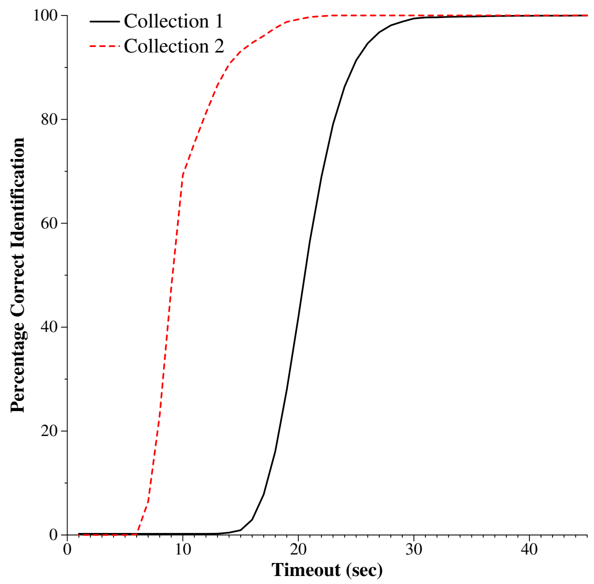


Figure 3: Percentage of correct failure identifications by static timeouts.

The second collection substituted two `ucsd.edu` machines for the Internet2 machines, as the Internet2 machines were down for maintenance.

We performed 4453 reductions of 16K integers (64 Kbytes) on Collection 1, and 2939 reductions on Collection 2. Scatter plots of the maximum reduction times for each reduction are plotted in Figure 2. The connectivity to the Internet2 machines was the limiting link in Collection 1, which is responsible for the larger reduction times. When these machines were removed from the tests, the reduction times were lower. Moreover, they show more variability as a result of network traffic (The tests of Collection 1 were done over a three day period starting March 9, 2001, and the tests of Collection 2 were done over a one-day period starting March 18, 2001. We will collect more results over a longer period for the final paper).

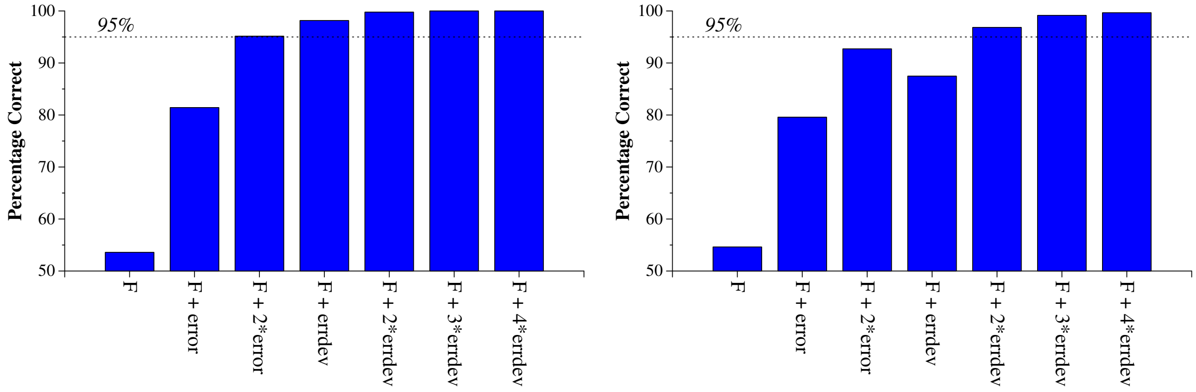
4.1 Static Timeouts

We wrote our reduction so that it identified a failure if any node took more than 45 seconds to perform the reduction. Given this metric, there were ten timeouts in Collection 1, and none in Collection 2. If we assume that a 45-second timeout correctly identifies failures, then Figure 3 displays how shorter timeouts fare in correctly identifying failures. Note that in this case, the true failures are indeed identified by shorter timeouts, but the shorter timeouts also incorrectly identify slowness as failure. The interesting feature of this graph is that the two collections, though similar in composition, have vastly different characteristics, and depending on the definition of “optimal,” require different timeout values for optimality. For example, suppose we define optimal to be the shortest timeout that correctly identifies at least 95 percent of the failures. Then Collection 1 would employ a 27-second timeout, and Collection 2 would employ a 17-second timeout.

In the final paper, we will also explore different timeouts for different nodes, since the location of a node on the tree affects its timeout value.

4.2 Dynamic Timeouts

To assess the effectiveness of dynamic timeout determination, we fed the reduction times into the Network Weather Service forecasting mechanism. For each reduction, the NWS comes up with a forecast of the next reduction time, plus two error metrics – a mean forecasting error to this point, and a mean square forecasting error. We use these values to assess seven different timeout determination strategies:



Collection 1 Collection 2

Figure 4: Percentage of correct failure identifications for the seven different dynamic timeout selection methods.

- **F**: the forecast itself
- **F +** i***error**: the forecast plus i times times the mean error. $i = 1$, 2.
- **F +** i***errdev**: the forecast plus i times the error deviation (square root of the mean square error). $i = 1$, 2, 3, 4.

We plot the effectiveness of these strategies for both collections in Figure 4. Since the forecast is an estimate of the next reduction time, setting the timeout to the forecast is obviously a bad strategy, because any reduction that takes slightly longer will fail prematurely. This is reflected in Figure 4. However, when the error metrics are added in, the dynamic timeout determinations become much better. In both collections, adding two times the error deviation brings the correct timeout determination above 95 percent.

To get an idea of what the predicted timeouts are using this method, see Figure 5, where the predicted timeouts are plotted along with the reduction times. Note that in both collections, the timeout values follow the trends in the data, reacting to the changing network conditions accordingly.

As before, in the final paper we will analyze the effect of having each node perform prediction rather than calculating a single timeout for each reduction.

4.3 Conclusion

In order for high performance applications to run effectively across the wide area, failure identification must be performed by the message-passing substrate. In this paper we have assessed the effect of static and dynamic timeout determination on a wide-area collective operation. Though the data is limited, it shows that a single system-wide timeout is certainly not effective for multiple computing enviroments. Additionally, we show that dynamic timeout prediction via the Network Weather Service can be extremely effective, even when the network conditions change over time.

Our long-term goal with this research project is to develop a message-passing library suited to the development of high performance applications that can execute on the wide area. This library will be based on the communication primitives of EveryWare, a project that has sustained high performance in an extremely wide-area, heterogeneous processing environment [15]. We view the issue of failure identification a central first step toward the effective development of this library.

5 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants ACI-9701333, ACI-9876895, EIA-9975015 and CAREER grants 0093166 and 9703390. Additional support comes from the NASA Information Power Grid Project, and the University of Tennessee Center for Information Technology Research.

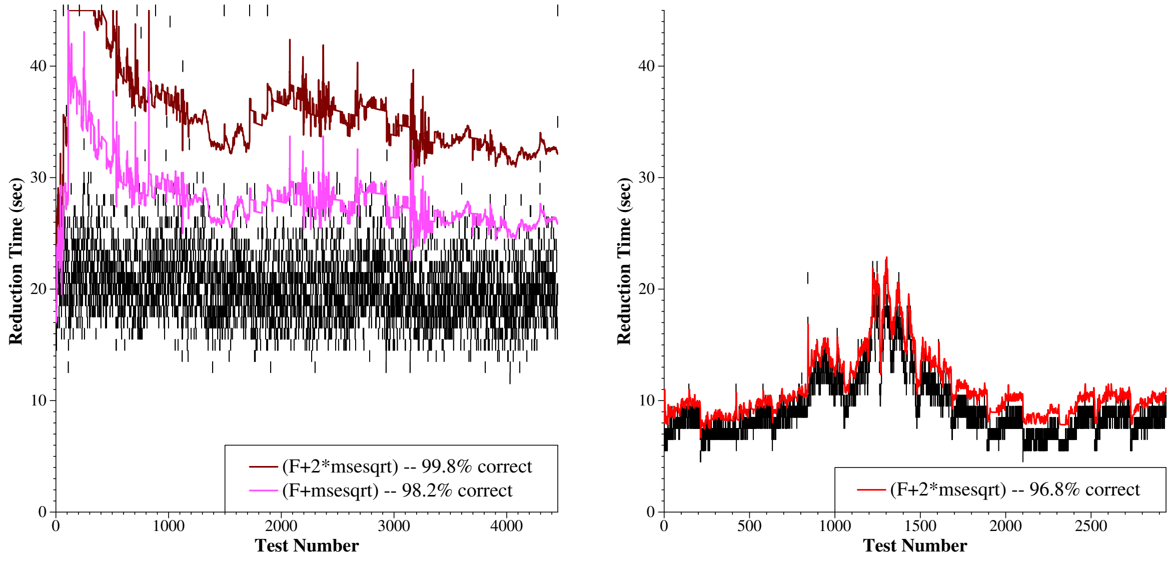


Figure 5: Reduction times along with dynamic timeout determinations using Network Weather Service forecasts plus mean square error metrics.

References

[1] A. Agbaria and J. S. Plank. Design, implementation, and performance of checkpointing in NetSolve. In *International Conference on Dependable Systems and Networks (FTCS-30 & DCCA-8)*, pages 49–54, June 2000.

[2] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[3] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing*, San Jose, November 1997.

[4] E. N. Elnozahy, L. Alvisi, Y-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[5] Entropia web page at: `http://www.entropia.com/`.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, Summer 1998.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, July 1998.

[8] A. S. Grimshaw, W. A. Wulf, and The Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[9] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[10] Network Working Group, R. Braden, ed. Requirements for Internet hosts – Communication layers. IETF RFC 1122 (`http://www.ietf.org/rfc/rfc1122.txt`), October 1989.

[11] Parabon Computation, web page at: `http://www.entropia.com/`.

[12] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobb's Journal*, #227:40–48, February 1995.

[13] United Devices, web page at: `http://www.ud.com/`.

[14] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. `http://www.cs.utk.edu/~rich/publications/nws-tr.ps.gz`.

[15] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the computational grid. In *SC99 Conference on High-performance Computing*, November 1999.

[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):757–768, 1999. `http://www.cs.utk.edu/~rich/publications/nws-arch.ps.gz`.