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@cs.utk.edu

‡ Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

Abstract

Failure identification is a fundamental operation concerning exceptional conditions that network programs must be able to perform. In this paper, we explore the use of timeouts to perform failure identification at the application level. We evaluate the use of static timeouts, and of dynamic timeouts based on forecasts using the Network Weather Service. For this evaluation, we perform experiments on a wide-area collection of 31 machines distributed in eight institions. Though the conclusions are limited to the collection of machines used, we observe that a single static timeout is not reasonable, even for a collection of similar machines over time. Dynamic timeouts perform roughly as well as the best static timeouts, and more importantly, they provide a single methodology for timeout determination that should be effective for wide-area applications.

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 present a discussion of typical timeout management strategies that are currently employed by most distributed computing systems. We then propose timeout determination methods based on both standard static values and online monitoring and prediction via the Network Weather Service [16]. We assess the effectiveness of the various timeout determinations using a collection of 30 machines distributed throughout the United States and one in Europe. Although the dynamic timeout determinations do not perform better than the best static methods, they do provide a uniform methodology for timeout determination that should be applicable in a variety of network computing settings.

2 The Problem of Failure Identification

One key difficulty associated with using the Internet to support computation is the presence of frequent communication failures. The typical communication abstraction used

by most applications today is the Unix socket abstraction. 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 send a packet (typically a packet with a previously acknowledged sequence number) to its communication partner. The TCP response to the arrival of a previously acknowledged packet over an existing connection is to send an acknowledgement packet with the correct sequence number to the other end of the connection. If, however, the connection is not recognized by the receiver as valid (e.g. the receiver has rebooted since the last packet exchange) a reset packet will be sent in response indicating that the connection should be terminated. Similarly, if the receiver simply does not respond within a given short period of time, the keep-alive sender assumes that the connection has failed and that the socket should be terminated.

The frequency with which keep-alive packets are sent determines the resolution with which a failure can be detected. A *timeout* occurs when the keep-alive mechanism detects a failed connection. Unfortunately, for network-based computing, this default keep-alive mechanism is not of great practical use. It was developed, primarily, to allow remote login facilities to terminate terminal connections when the remote machine reboots after a long period of idle time. The timeout values are typically 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. Nowadays, 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 (driven by a user-settable timer) 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. This paper goes under the assumption that the application employs the latter strategy. 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.

We propose that for better performance, an application must be able to control the timeouts for failure identification. This may be done either by a configurable keep-alive timeout, or by implementing the same functionality

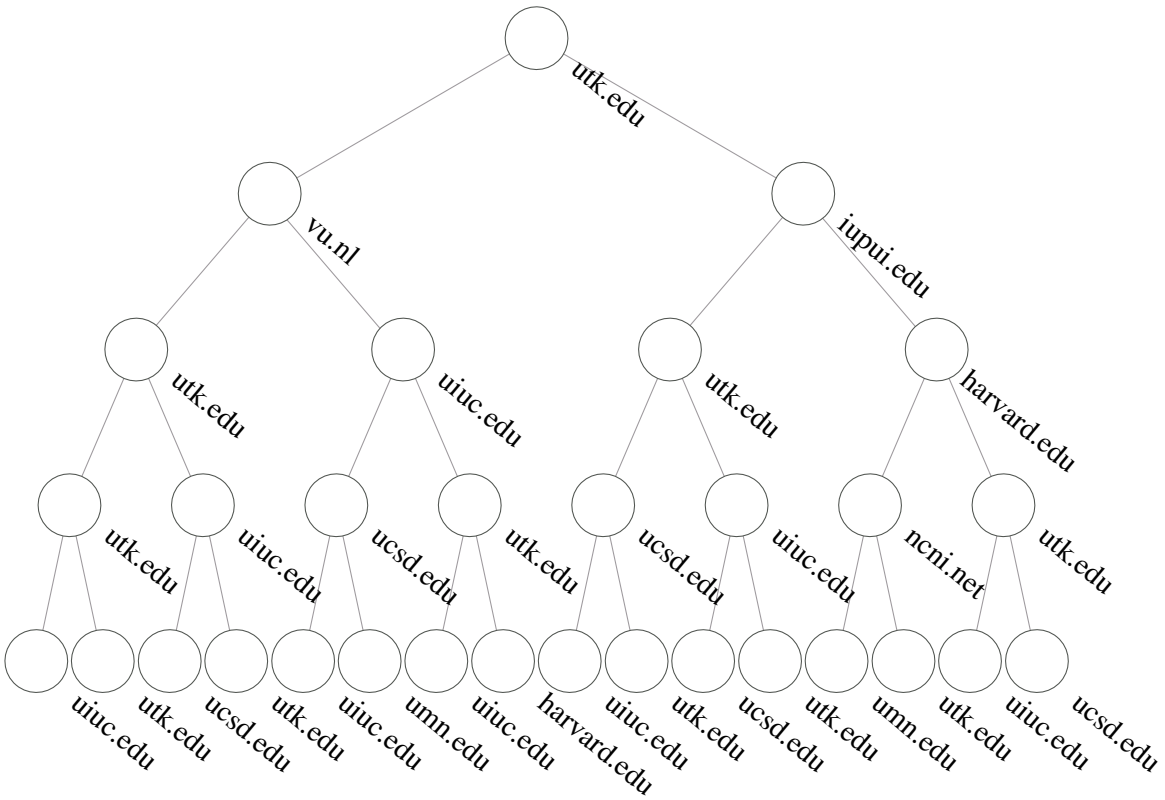

Figure 1: **Topology for a 31-node tree-based reduction.**

at the user level by interrupting **read()/write()** system calls with an alarm signal, or by using variants of **read()/write()** that support timeouts. A large body of literature (see recent ACM SIGCOMM proceedings) addresses the issue of timeouts in networking. In our approach, we are taking the networking implementation as a given, and addressing strategies that the network computing application may employ to deal with failure identification.

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. The details of each model are beyond the scope of this paper and are discussed in [16]. However, what is important is how the NWS blends the various forecasting models. 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 characterize 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 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]. NWS is supported on every architecture – the reader is referred to `http://nws.cs.utk.edu`

Location	# of Machines		
	Collection 1	Collection 2	Collection 3
University of Tennessee in Knoxville	11	11	10
University of California, San Diego	5	7	8
University of Illinois at Urbana-Champaign	8	8	8
Harvard University	2	2	1
University of Minnesota	2	2	2
Vrije University, the Netherlands	1	1	1
Internet2 Distributed Storage Machine, Chapel Hill, North Carolina	1	0	1
Internet2 Distributed Storage Machine, Indianapolis, Indiana	1	0	0

Table 1: **Composition of the three collections of 31 nodes**

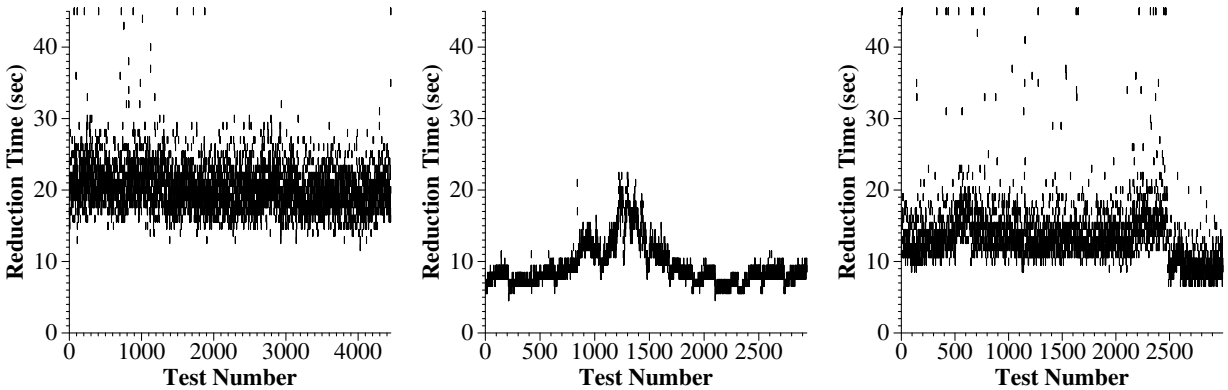

Figure 2: **Scatter plot of reduction times as a function of test number for both 31-node collections.**

for the most recent NWS distribution.

We use the NWS to determine a dynamic 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 (i.e. avoids a greater number of false timeouts).

In the sections below, we use a very simple application on a wide-area network to explore the effects of automatically setting network timeouts based on dynamically perceived response delay.

4 Experimental Validation

We choose a simple application to test the effects of timeout determination. The application finds the maximum value amongst a set of values that is partitioned across processors and then broadcasts it so that all processors have a valid copy of the maximum. Specifically, we have n computing nodes, each of which holds a set of m values. Let the j-th value of node i be a_j^i. When the reduction is complete, a_j^i is the same for any fixed value of j, and is equal to the maximum value of a_j^i 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_j^r. 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_j^i.

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 three collections of thirty-one machines, whose compositions are listed in Table 1.

Note, these are collections of thirty machines from the continental United States and one from Europe. We arranged the machines into a five-level tree as exemplified in Figure 1, which depicts Collection 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.

We performed 4453 reductions of 16K integers (64 Kbytes) on Collection 1, 2939 reductions on Collection 2 and 2997 on Collection 3. 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 Collections 1 and 3. This 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; the tests of Collection 2 were done over a one-day period starting March 18, 2001, and the tests of Collection 3 where done over an eight day period starting August 8.

4.1 Static Timeouts

We wrote our reduction so that it identified an application failure if any node took more than 45 seconds to perform the reduction. Given this metric, there were ten timeouts in Collection 1, none in Collection 2, and 25 in Collection 3. If we assume that a 45-second timeout correctly identifies failures, then Figure 4 displays how shorter timeouts fare in correctly identifying failures. In this and subsequent experiments in Section 4, a timeout is defined to occur when any node determines that the reduction is taking longer than the timeout value.

Note that in Figure 4, 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 three 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 (depicted by the thick dotted line in Figure 4). Then Collection 1 would employ a 27-second timeout, Collection 2 would employ a 17-second timeout, and Collection 3 would employ a 40-second timeout. Note, in contrast, that the mean reduction time is 20.4 seconds for Collection 1, 9.40 seconds in Collection 2 and 13.8 seconds for Collection 3.

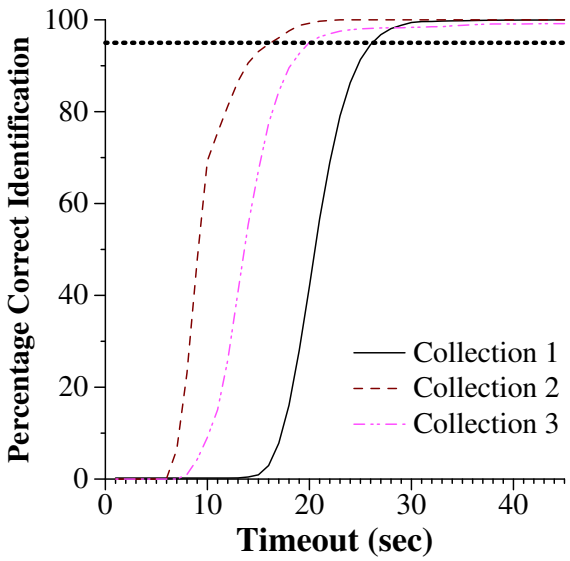

Figure 4: **Percentage of correct failure identifications by static timeouts.**

While percentage of correct timeout identifications is a valid metric for selecting a timeout, we must also assess the implications of the timeout selection on the application. This is hard to do in practice, since network conditions are continuously changing, which makes it difficult to run controlled experiments. In an attempt to assess the implications, Figure 3 uses the data from the three collections to approximate the average running time of the reductions when timeouts are determined statically. For each timeout value in the graph, we attempt to simulate the running time of each reduction in the data sets as follows. If the actual reduction time is less than or equal to the simulated timeout, than that is used as the simulated reduction time. If the actual reduction time is greater than the simulated timeout, then we assume that the reduction is attempted

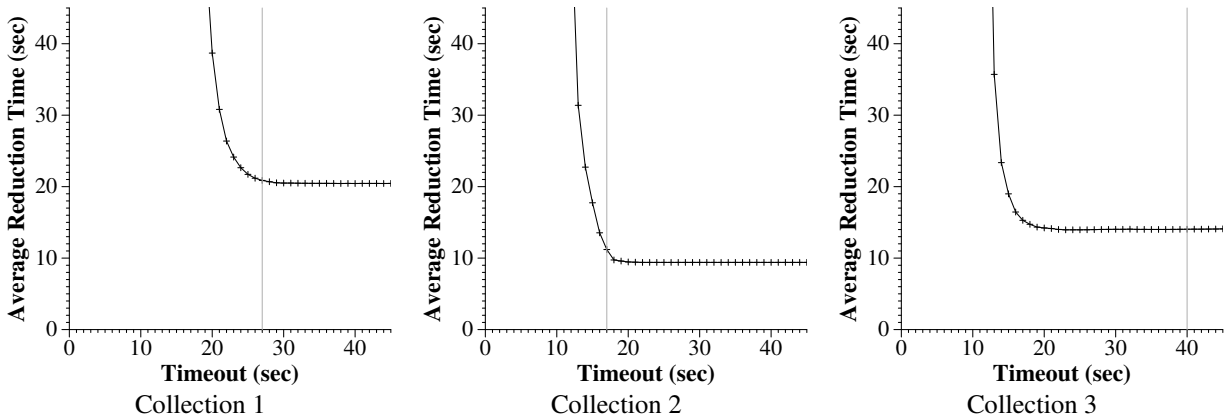

Figure 3: **Simulated average running times for static timeout values**

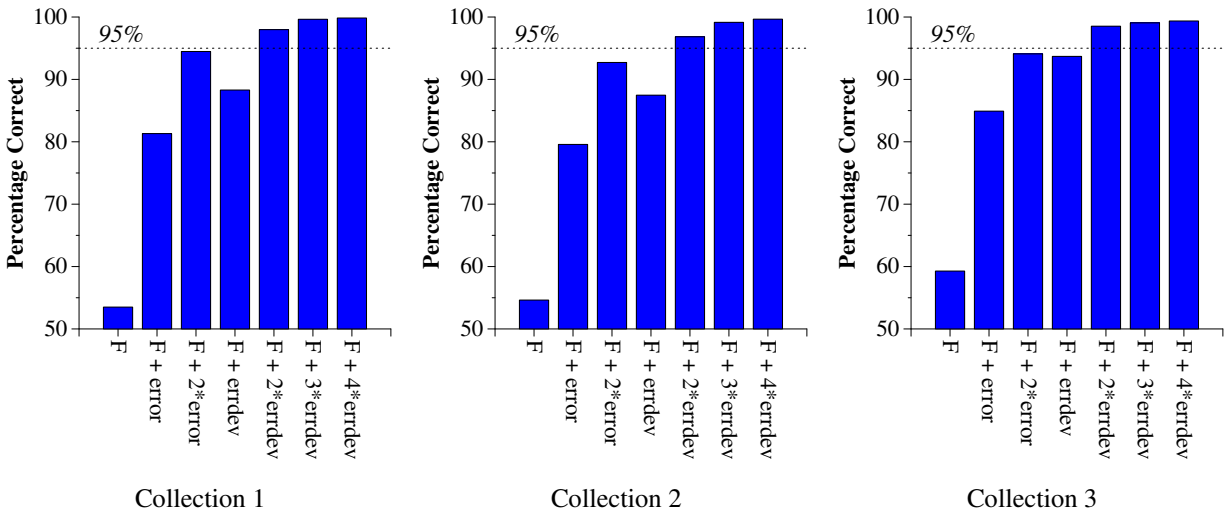

Figure 5: **Percentage of correct failure identifications for the seven different dynamic timeout selection methods.**

anew. The timeout is added to a total reduction time, and the next reduction in the data set is employed as the time for the retried reduction. We continue in this manner for each reduction in each data set, and average the simulated reductions.

As the graphs show, for each collection, there is a minimum timeout value beyond which higher timeout values yield approximately similar average reduction times. Horeover, the 95-percent timeout value, as depicted by the vertical gray line in each graph, is *not* a good indicator of where this minimum timeout value lies. It should be noted that all curves in Figure 3 are not strictly decreasing. Collection 1 reaches its minimum at a timeout of 38 seconds and Collection 3 at 23 seconds. Collection 2 reaches its minimum at at 22 seconds, and stays there for larger timeout values.

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:

- **F**: the forecast itself
- **F + i*error**: the forecast plus i times times the mean error. $i = 1, 2$.

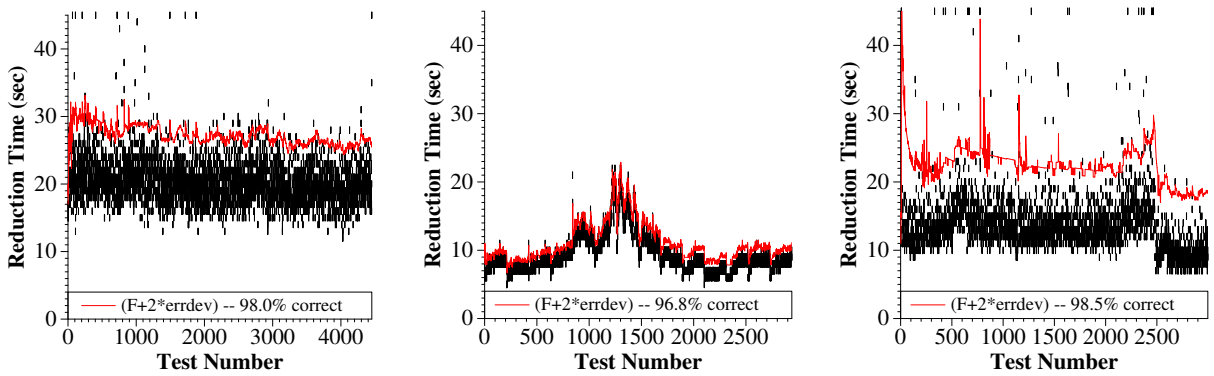

Figure 6: **Reduction times along with dynamic timeout determinations using Network Weather Service forecasts plus twice the error deviation.**

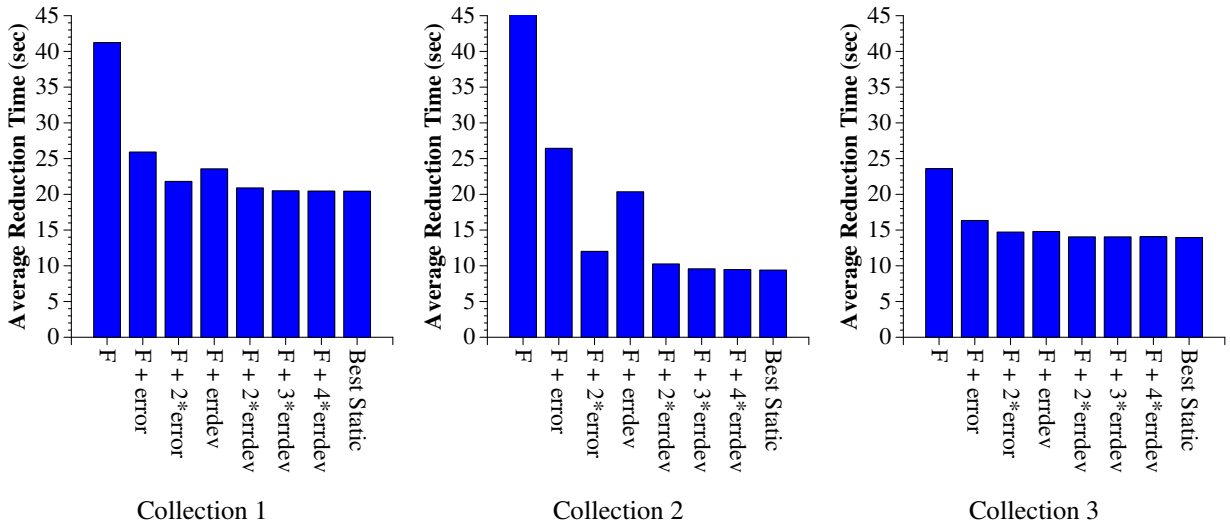

Figure 7: **Simulated average running times for dynamic timeout values**

- **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 all three collections in Figure 5. 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 5. 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 6, where the **F + 2*error** predictions are plotted along with the reduction times. Note that in all three collections, the timeout values follow the trends in the data, reacting to the changing network conditions accordingly.

Next, we performed simulations of using the collection data to calculate average reduction times as in Figure 3. However, instead of using static timeouts, we used forecasted values. The results are in Figure 7. Note that the best static value is plotted along with the dynamic values. Although the dynamic forecasts appear to follow the data, they do not yield lower average reduction times — in fact the best static and dynamic values appear to to roughly equal. This will be discussed in Section 5 below.

4.3 Experiments with Static and Dynamic Link Timeouts

The above determination has focused on whole reduction timeouts. In other words, each node calculates a timeout for the entire reduction, and the entire application is flagged with a failure if any of the processing nodes senses a timeout. This experiment is reasonable, since it is an implementation strategy that would be suitable to a wide-area application where the application has little control over the kernel of the nodes, or perhaps even the message-passing substrate (for example a MPI or PVM application).

As a further exploration, we instrumented our reduction so that each link can set its own timeout, and if that timeout expires, it triggers a resend of the data, but does not shut down the application. The application is shut down when the reduction has not been completed in 90 seconds. We then performed roughly 1000 iterations on the machines of Collection 3, where each iteration performs six reductions:

- Static timeout of 1.5 seconds.
- Static timeout of 5 seconds.
- Static timeout of 15 seconds.
- Static timeout of 45 seconds.
- Dynamic timeouts using all previous reduction data, where the timeout is equal to the forecast plus the error deviation.
- Dynamic timeouts using all previous reduction data, where the timeout is equal to the forecast plus two times the error deviation.

The results are tabulated below, and plotted in Figure 8.

Timeout Determination Method	Average Reduction Time (sec)	Average Retries Per Node	Reduction Shut Downs
Static, $t = 1.5$	16.96	0.199	15
Static, $t = 5$	13.56	0.002	8
Static, $t = 15$	13.47	0.000	5
Static, $t = 45$	13.71	0.000	8
Dyn., **F + errdev**	13.69	0.021	24
Dyn., **F + 2errdev**	14.31	0.062	19

There is little appreciable difference between the dynamic timeout determinations, and the static timeouts greater than five seconds per link.

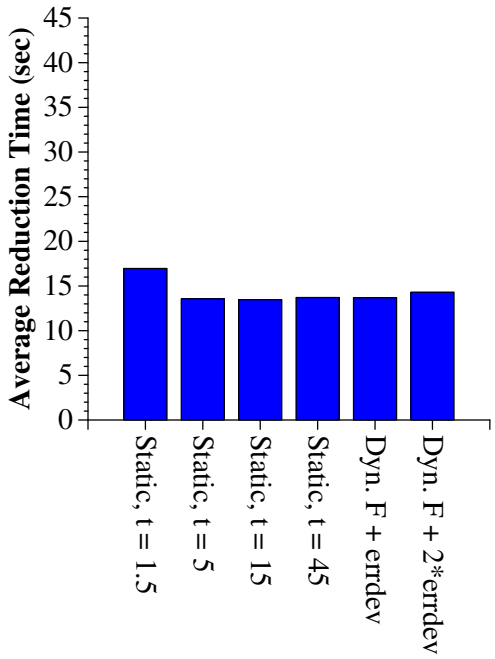

Figure 8: **Average running times with different link timeouts**

5 Discussion

In the above experiments, we have compared the effectiveness of static and dynamic timeout determinations in terms of percentage of correct timeout determinations, simulated reduction times using reduction trace data, and performing live experiments with link timeouts built in. We believe the following significant observations may be made concerning these experiments:

1. In terms of the smallest effective static timeout value, there is no one system-wide value. This is a direct observation from the three graphs in Figure 3. Although the composition of the three collections does indeed differ, the collections are similar enough to each other to think that a single static timeout value may exist. Both Figures 3 and 4 imply that this is not the case.

2. The Network Weather Service forecasts can generate timeouts that follow the trends in the data. In all three collections, using forecasts plus two times the error deviation generated timeouts that both followed the changing network conditions (as demonstrated in Figure 6), and yielded timeout determinations that were correct in over 95 percent of the cases.

3. Although the dynamic timeout determinations did not outperform the static timeouts in either the simulations or the live experiments, the two methodologies produced similar results in their best cases. This is significant, since a single optimal static timeout cannot be determined for all cases. However, the NWS forecasts do present a single *methodology* that can yield good timeout

performance in all cases.

4. More work needs to be done in this area. Since this work deals with long-term averages and exceptional conditions, it is hard to draw far-reaching conclusions without an enormous amount of data. We look forward to performing more work on larger collections of machines, differing applications, and longer time durations.

6 Conclusion

In order for high performance applications to run effectively across the wide area, failure identification must be performed by either the application or 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 environments. Additionally, we show that dynamic timeout prediction via the Network Weather Service can be effective, and is advantageous 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 to be a central first step toward the effective development of this library, and we believe that further exploration into the nature of effective failure identification will reap benefits for future distributed and network programming environments.

7 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. We also thank our friends at various institutions — Norman Ramsey, Thilo Kielmann, Graziano Obertelli, Micah Beck, and the folks from NPACI — who let us use their machines for this research.

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.