A Fault Detection Service for Wide Area Distributed Computations

Paul Stelling[1] Ian Foster[2] Carl Kesselman[3] Craig Lee[1] Gregor von Laszewski [2]

[1] The Aerospace Corporation
El Segundo, CA 90245-4691

[2] Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439

[3] Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292

Abstract

The potential for faults in distributed computing systems is a significant complicating factor for application developers. While a variety of techniques exist for detecting and correcting faults, the implementation of these techniques in a particular context can be difficult. Hence, we propose a fault detection service designed to be incorporated, in a modular fashion, into distributed computing systems, tools, or applications. This service uses well-known techniques based on unreliable fault detectors to detect and report component failure, while allowing the user to tradeoff timeliness of reporting against false positive rates. We describe the architecture of this service, report on experimental results that quantify its cost and accuracy, and describe its use in two applications, monitoring the status of system components of the GUSTO computational grid testbed and as part of the NetSolve network-enabled numerical solver.

1 Introduction

A major difference between distributed and sequential computing, as they are usually practiced, is that in a distributed computation, individual components may fail without the entire computation being terminated. Indeed, components may fail without the rest of the computation being aware that failure has occurred. These phenomena represent both an opportunity and a challenge. The opportunity is that a computation can, in principle, continue to operate despite failure of individual components. The challenge is that new techniques are required for detecting and responding to component failures. Many years of research on these topics have yielded a considerable body of theoretical and practical knowledge of fault detection, handling, and recovery techniques.

In our work, we approach these issues from the perspective of the user of what are termed *computational grids* [9], that is, networks of computing resources, often high-performance computers, intended to be used in an integrated fashion for such problems as collaborative engineering, computational steering, and distributed supercomputing. Grid programmers often want to adapt existing computational models, tools, or applications for distributed execution. They require services that simplify this task by encapsulating complex aspects of distributed computing environments. In previous work, we have developed and demonstrated the utility of services for resource location, resource allocation, information, communication, security, and data access, collectively termed the Globus toolkit [8]. These services use simple local mechanisms to support a variety of global policies. For example, the Globus resource management service deploys just a simple local manager at each managed resource, but supports a variety of management policies via resource brokers and co-allocators [6].

In this paper, we propose a low-level service to support fault handling strategies in grid applications. These applications may want to respond to component failure in a variety of ways. For example, they may

- terminate the entire application (i.e., fail-stop);
- ignore the failure and continue execution;
- allocate a new resource and restart the failed application component [13, 4]; or

- use replication and reliable group communication primitives to continue execution [1, 15].

Each of these behaviors has costs and benefits associated with it and the appropriate behavior will be application-dependent.

To date, grid applications have either ignored failure issues or have implemented failure detection and response behavior completely within the application. This approach places an undue burden on the application developer and complicates the design and development of grid applications. The situation could be improved considerably if the underlying grid infrastructure provided basic services that support the implementation of application specific failure behaviors, such as those described above.

In this paper, we consider the nature of these grid services and propose a specific service based on fault detectors, which detect when a system component has failed and notify the application of that fact. Factors such as the highly variable communication latency and best-effort service provided by today's wide area networks, and the need to construct a scalable service impact the design of the fault-detector. These pragmatic issues dictate that we consider an *unreliable fault-detection service* which may sometimes report a resource to have failed, only to retract that report at a later time. We have designed and implemented such a service and have demonstrated that a range of application-specific fault behaviors can be implemented on top of this service.

In Section 2, we discuss the nature of faults, review some basic results in distributed systems, and show why fault detection is an appropriate basic service to provide in the grid. In Section 3, we define a system model which we use to define an fault detection architecture. In Section 4, we present a design for a specific implementation of a fault detection service and discuss how this service has been used to construct fault-tolerant grid applications (Section 6). In Section 5, we provide results that quantify the accuracy of our fault detector in a realistic wide-area environment. We then compare our approach with other approaches to constructing fault-tolerant grid applications and conclude with a discussion of future work.

In summary, the contributions of the work described in this paper are:

- We propose the unreliable fault detector as a basic grid service and propose a specific fault detection architecture and implementation for wide-area computational environments.

- Using experimental data, we demonstrate that our approach to fault detection can be implemented efficiently and accurately.

- We demonstrate that the proposed service can be used to implement useful behaviors in distributed computing systems and applications.

2 Faults and Distributed Systems

Components of a distributed system can fail in different ways [18]. In the simplest case of *crash failure*, a component simply ceases to function, for example due to an operating system crash or the severing of a network connection. A special case of crash failure is *fail-stop failure*, in which a crash results in the component transitioning permanently to a state that allows other components to detect that it has failed (e.g., by ceasing to send periodic "i-am-alive" messages). More complex failure modes are also possible, for example when a component fails by not functioning "correctly," such as when a memory chip returns an incorrect value, a packet is corrupted during transit over the network, or (in the extreme, so-called Byzantine case) because a component operates in a malicious fashion, perhaps causing a general failure or obscuring the real source of the failure.

In this paper, we focus on the problem of detecting fail-stop crash failures. While in some situations, system components can detect and correct even Byzantine failures through the use of mechanisms such as redundancy and retransmission, this is difficult and may require detailed knowledge about the components in question. Furthermore, many such failure modes are masked at the component level.

We are interested in the question of what basic services should be provided as part of a distributed computational infrastructure to support fault recovery. To gain insight into this problem, it is interesting to consider what is the least amount of information needed to implement some basic reliable distributed algorithms, such as Consensus. In the Consensus algorithm, all functioning processes must propose and unanimously agree on a value, in spite of the fact that any of the participating processes may fail during the execution of the algorithm. Consensus has interesting practical uses as it can be used to implement essential fault-tolerant functions such as leader election and voting.

A fundamental result from distributed systems is that Consensus can not be implemented in a distributed system subject to crash failures if the system is asynchronous, i.e., if no timing assumptions can be made, such as the amount of time it takes for two processes to communicate or the amount of time it takes for an operation to complete [7]. Note that this asynchronous distributed system model corresponds well with the best-effort service provided over current wide area networks.

One solution to this problem is to augment the distributed system with additional information, such as knowledge about which system components have failed. Given this information, it is possible to implement Consensus in the presence of crash failure. Less formal failure behaviors, such as restart, can also be defined with respect to failure detection. Failure detection provides a sound foundation on which to build a range of failure behaviors and as such, we conclude that it should be provided as a basic service in distributed computing environments.

It is interesting to consider what are the weakest properties that a failure detector can have and still be useful. In [5], it is shown that Consensus in asynchronous distributed systems can be solved with an *unreliable failure detector*: a failure detector that can erroneously indicate that a component has failed only to correct this error at a later time. Furthermore, an unreliable failure detector can be distributed, with each component of the system having access to its own detector and each detector potentially producing a different account of which system components have failed. This result holds as long as the failure detector meets some minimal requirements for completeness and accuracy. In particular:

- all failed components are eventually discovered and permanently identified as such, and
- at least one functioning component is known to be functioning by all functioning components in the system after some point in time.

From the perspective of producing a practical service, unreliable failure detectors have several advantages over reliable detectors. Because each component can have access to its own failure detector, and detectors do not have to agree about what components of the system have failed, the service does not have to be centralized, nor do we need to provide a globally consistent state across detectors. Furthermore, unreliable communication protocols can be used to implement an unreliable failure detector. These protocols have the advantage of lower overheads, lower latency and better scalablity. For these reasons, an unreliable failure detector will be more scalable, simpler, and more efficient to implement then a reliable detector.

An unreliable failure detection service is not without limitations. Provably correct algorithms guarantee termination by a combination of iteration and the fact that the failure detector will *eventually* identify all failed components and at least one functioning component. In real systems, this unbounded wait is unacceptable as it can be the case that the cost of waiting for an absolutely correct determination may exceed the cost that would be incurred if we simply assumed that the failure detector was correct and took action based on this assumption.

Ultimately, the decision as to when the information provided by a failure detector is to believed must be the responsibility of an application; the failure detector cannot interpret its results. An application must use the information provided by the failure detector to make a decision based on the probability of a failure report being in error, the application-specific cost of performing some action if the report was false, and the application-specific cost of not performing that action if the report was in fact true. Clearly, information about the reliability of the failure collector is necessary. In Section 5, we show that the probability of an erroneous report is generally low, and decreases the longer that one waits.

In summary, we propose that a distributed computing environment should provide unreliable failure detection as a basic service, providing notification when system components *might* have failed and leaving it to the application to interpret this information based on a characterization of the fault service, future information provided by the service and application requirements.

3 Design of a Fault Detection Service

We now consider issues that arise in designing the proposed fault detection service. We discuss the entities for which we wish to detect failure, the design goals for our fault detection service, and the overall architecture of this service.

3.1 System Model

We first define a model for the system being monitored by the fault detector. This model identifies the visible components of the system and hence determines what types of entities the fault detector needs to monitor. In principle, system components could be entire sites, specific computers, processors within a computer, processes, threads, network interfaces, network connections, or any number of other low-level system abstractions. For reasons of complexity, utility and overhead, we have chosen to model the system as consisting of processes and computers.

Our ultimate goal in constructing the fault detector is to enable the construction of robust applications, not to diagnose the causes of system or application failure. Considering a computer as a single unit prohibits the detection of some types of failure, such as a specific disk going off-line. However, it is often the case that failures of a component of a computer are detected by underlying system mechanisms and cause the entire computer to fail. A similar argument can be made for not considering low-level software abstractions such as threads. In both cases, we considered the cost of having the fault detector deal with such low-level abstractions as outweighing any potential benefits.

Note that we do not include networks as components to be monitored. Detection of network failure tends to be difficult because it is hard to discriminate between host failure and network failure without the existence of a second, independent path. Furthermore, the identification of such paths when they exist requires both detailed knowledge of network topology and coordination among distributed monitors. For reasons of simplicity and generality, we limit monitoring to processes and hosts. Note that monitoring processes and hosts does serve to monitor indirectly the network connections between these monitored objects.

3.2 Design Goals

Given this system model, we consider the requirements which the fault detector must satisfy. The main concerns that should be addressed in the design of a fault detector for grid environments are:

- **Scalability**. The design of the fault detector must be capable of scaling to large numbers of processes and computers.

- **Accuracy and completeness**. The fault detector must identify faults accurately, with both false positives and false negatives being rare.

- **Timeliness**. Problems must be identified in a timely fashion, so that responses and corrective actions can be taken as soon as possible.

- **Low overhead**. Monitoring should not have a significant impact on the performance of application processes, computers, or networks.

- **Flexibility**. We want to support a range of application-specific fault detection policies and usage models. For example, applications may wish to control which entities are monitored, how often they are monitored, the criteria used to report failure, and where failures are reported.

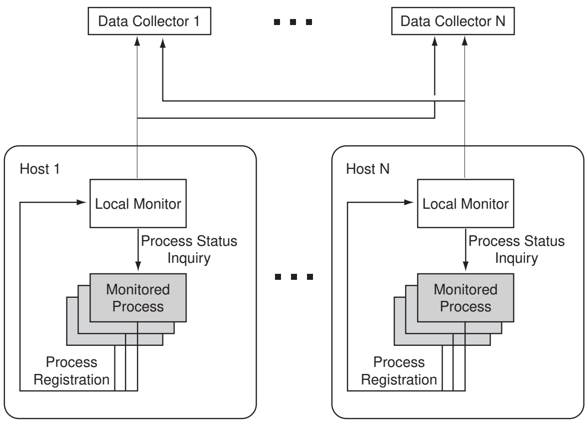

Figure 1. Architecture of a fault detector for monitoring computers and processes

3.3 A Fault Detection Service Architecture

As illustrated in Figure 1, our fault detection service architecture is defined in terms of two types of entities:

1. A *local monitor* is responsible for observing the state of both the computer on which it is located and any monitored processes on that computer. It generates periodic "i-am-alive" messages or *heartbeats*, summarizing this status information.

2. A *data collector* receives heartbeat messages generated by local monitors and actually identifies failed components based on missing heartbeats.

As we shall see in the next section, this separation of local monitor and data collector functions provides considerable flexibility in how we handle faults.

Scalability and performance concerns require that the heartbeats communicated by the local monitors be transmitted via a connectionless, typically unreliable protocol. Hence, the data collector must take into account network delays and possible packet loss when interpreting a lack of heartbeat from a particular local monitor. We discuss these issues below.

4 A Fault Detection Service

We now discuss the implementation of a specific fault detector service, namely the Globus Heartbeat Monitor (HBM), which provides a fault detection service for applications developed with the Globus toolkit. HBM comprises three components:

- A local monitor, responsible for monitoring the computer on which it runs, as well as selected processes on that computer.
- A client registration API, which an application uses to specify the processes to be monitored by the local monitor, and to whom heartbeats are sent.
- A data collector API, which enables an application to be notified about relevant events concerning monitored processes.

In brief, an application that wishes to use the HBM service needs to do just two things:

1. Register the processes for which failure detection is required, either by calling the registration API directly, or by having the registration function called externally on behalf of the application.
2. Use the data collector API to construct a data collector that implements the desired application-specific fault behavior, whether this is global termination, rescheduling of a failed component, further testing to verify that failure has occurred, etc.

We describe these aspects of the HBM service in turn.

4.1 Client Registration API

Globus-based grid systems maintains a HBM local monitor for each Globus-managed resource. Hence, the Globus user need not be concerned with creating or maintaining these processes. The local monitor is typically run on the resource itself, to simplify the task of monitoring the status of the resource and of processes running on the resource. On workstations and shared memory computers, such as the Convex Exemplar, the local monitor runs directly on the machine in question. On distributed memory computers such as the Cray T3E, the local monitor runs on a tightly coupled front end or service node.

A process must be explicitly registered with the local monitor for its status to be reported. A client registration API is provided for this purpose. This API may be called either from within the application program or externally by a separate process. The registration process provides the local monitor with the identity of the process to be monitored, the identity of the data collector(s) to which process heartbeats are to be sent, and a heartbeat interval. On termination, processes use an unregister function provided by the client API to disconnect from the local monitor, preventing them from being reported as failed.

The local monitor can use a range of methods to determine the status of registered processes. We currently use the standard UNIX `ps` command to report on monitored processes. The `/etc/proc` mechanism found on many UNIX platforms could also be used.

The local monitor reports the status of each monitored process to the appropriate data collectors at the time of process registration and unregistration, and at fixed specified intervals in between. A separate message of size 70-90 bytes is sent for each monitored process. This message includes data identifying the monitored process and its current status. In addition to process heartbeats, the local monitor also generates a heartbeat for itself, allowing an application to detect a resource failure even if there are no monitored processes running on that resource.

Heartbeat data is sent to the data collectors using an unreliable datagram service: specifically, the UDP protocol. We chose this protocol over the reliable TCP protocol for several reasons. First, TCP is connection-oriented and consumes resources on both the sender and receiver. The overhead associated with UDP is less, making this solution more scalable than if TCP had been used. Second, the fact that TCP is a reliable protocol tends to introduce additional latency into communication operations. Given that heartbeats are time sensitive, introduction of additional latency in the delivery of heartbeat data is ill advised. Finally, when available, we wanted to have the option to use multicast to send data from a local monitor to an arbitrary and dynamic set of data collectors, and TCP cannot be used in conjunction with multicast.

4.2 The Data Collection API

The HBM data collection API allows for the construction of application-specific data collectors. The API is callback-based, allowing an application to register a function to be called when an event of interest occurs. When making a call to the data collection API, an application provides a callback function along with an event mask to indicate the events the callback should be called on, such as a late heartbeat or a heartbeat received.

The API implementation keeps track of all registered processes and records whenever a heartbeat arrives. Since the data collector knows the frequency at which heartbeats are being generated by registered processes, it can infer missing heartbeats. The API can generate callbacks for missing heartbeats for individual processes, or for the host itself. Callbacks can also be issued when other events of interest occur, such as when a new process is registered, or when a process unregisters or is reported as having failed.

Note that the function of the data collection API is limited to keeping track of heartbeats and invoking callbacks into the application. An application-specific data collector must provide a set of callback functions that implement the desired responses in response to the HBM callbacks. Again, it is the responsibility of the application to make the determination as to component failure based on how late the heartbeat is, the requirements of the application, and the type of fault recovery being implemented.

We note that the structure of the data collection API offers us a great deal of flexibility not only in how a data collector is implemented, but where it is implemented as well. Data collection functions can be integrated into the basic algorithms of an application, provided by specialized modules started as part of the application, or by separate, stand-alone programs. This flexibility further promotes the use of the HBM to implement a wide range of fault behaviors.

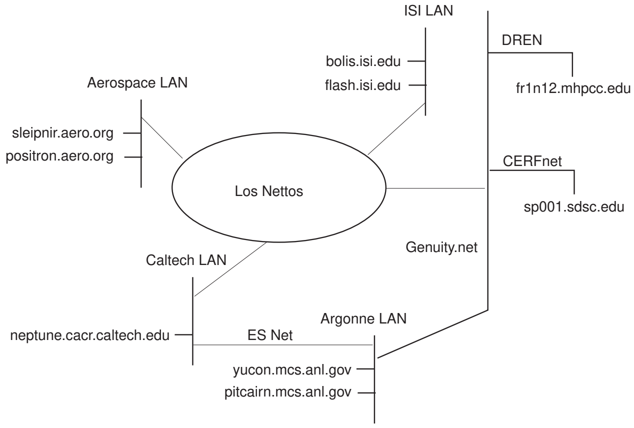

Figure 2. The nine hosts used in the HBM experiments, showing their connectivity, which included local area networks, a metropolitan area network (Los Nettos, a 100 Mb/s network in southern California), and the Internet.

5 Experimental Results

We are concerned with two aspects of HBM performance: first, the costs associated with monitoring (at hosts, network, and data collectors), particularly as the number of monitored hosts increases; and the accuracy of the HBM reports: that is, how quickly a failure is reported, and how frequently such reports turn out to be incorrect. We discuss these two issues in this section and report on experimental results that provide some insights into these questions.

As discussed above, HBM monitors send heartbeat messages to data collector(s) at regular intervals. In addition to recording and handling failures reported by the local monitors, the HBM data collector must diagnose potential component failure when no heartbeat message has been received from that component for a specified amount of time. The data collector is not guaranteed to receive all such messages, as heartbeats may be lost or delayed for a variety of reasons, including network congestion, scheduling delays at the local monitor or data collector, and network failure. Hence, there is always the possibility that the data collector may diagnose a component as having failed when it has not. For this reason, any discussion of HBM accuracy involves a tradeoff between the amount of time we are prepared to wait before concluding that a component has failed, and the number of false reports that we are prepared to deal with.

Our goal is to minimize some function of reporting delay, false positive rates, and system overheads. The parameters that we can control are system costs (by which we mean primarily heartbeat frequency, although the priority given to HBM processes and to HBM network traffic can also be cost issues) and definition of failure. Note that these variables can be varied on a per-component basis.

Previous research [2, 16, 3, 17] provides some relevant data. For example, in a 1992 study in which 32 byte packets were sent over a 128 Kb/s transAtlantic link at regular intervals for extended periods, Bolot observes high loss rates (9%) but notes that losses are essentially independent as long as probe traffic accounts for less than 10% of available bandwidth. In a 1997 study, Borella et al. [3] analyzed traffic between three pairs of sites located variously within a metropolitan area and on a wide area network. They sent an 80 byte packet every 30 msec and observed packet loss rates of 0.36%, 0.61% and 3.54% for the three pairs. Losses were seen to be bursty, but the mean loss burst size of 6.9 (around 200 msec) suggests that loss rates for packets sent at 10 sec intervals would be essentially independent. Simulation studies show similar results.

In order to obtain more detailed data on loss rates we conducted our own studies. We studied HBM performance on an experimental system comprising the nine hosts shown in Figure 2. A local monitor at each host sent heartbeats to data collectors at every host (including itself) at 10 second intervals for several days, during which time a total of 3,835,905 messages were sent, of which 93.6% were received. The (modified) data collectors logged timestamped heartbeats for subsequent analysis.

We use the heartbeats received at each host to com-

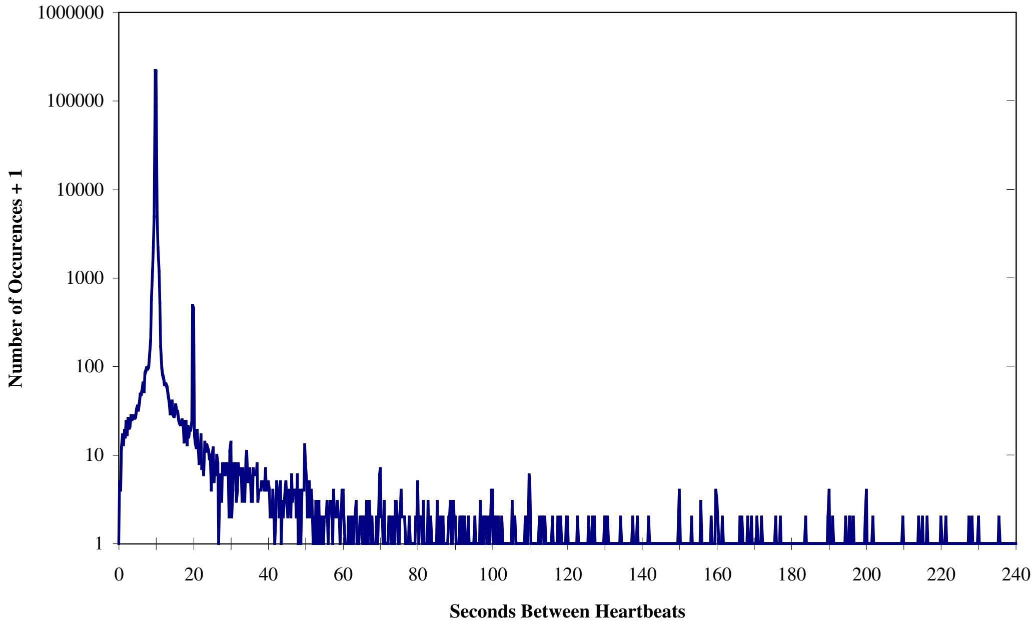

Figure 3. Histogram representation of the interarrival time data observed at bolas, one of the nine hosts in our experimental testbed, showing the distribution of interarrival times. Each of the peaks on the far right represents a single heartbeat.

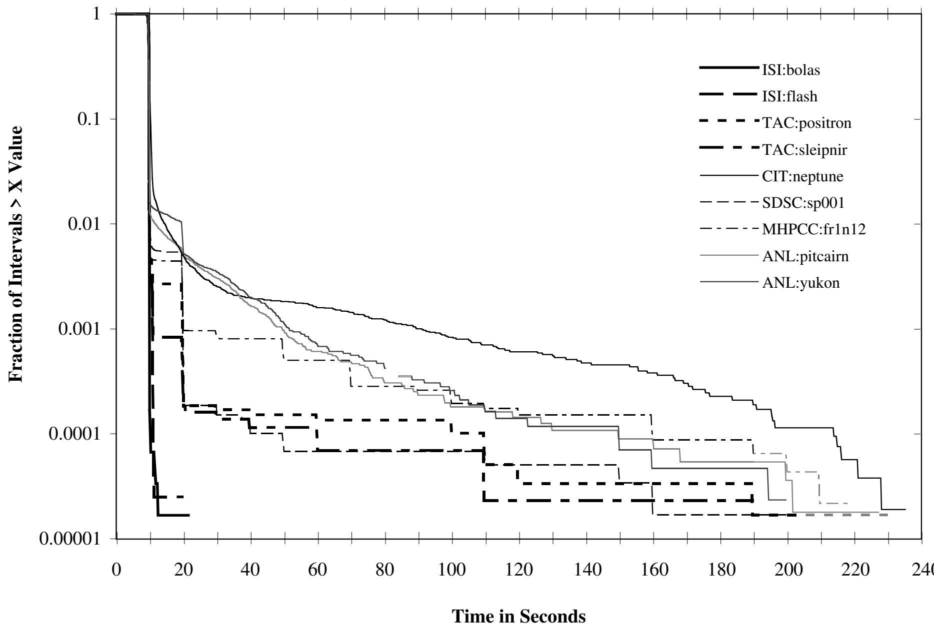

Figure 4. Interarrival times observed at bolas for heartbeats arriving from each of the nine hosts, expressed in terms of proportion of heartbeats with interarrival time greater than X.

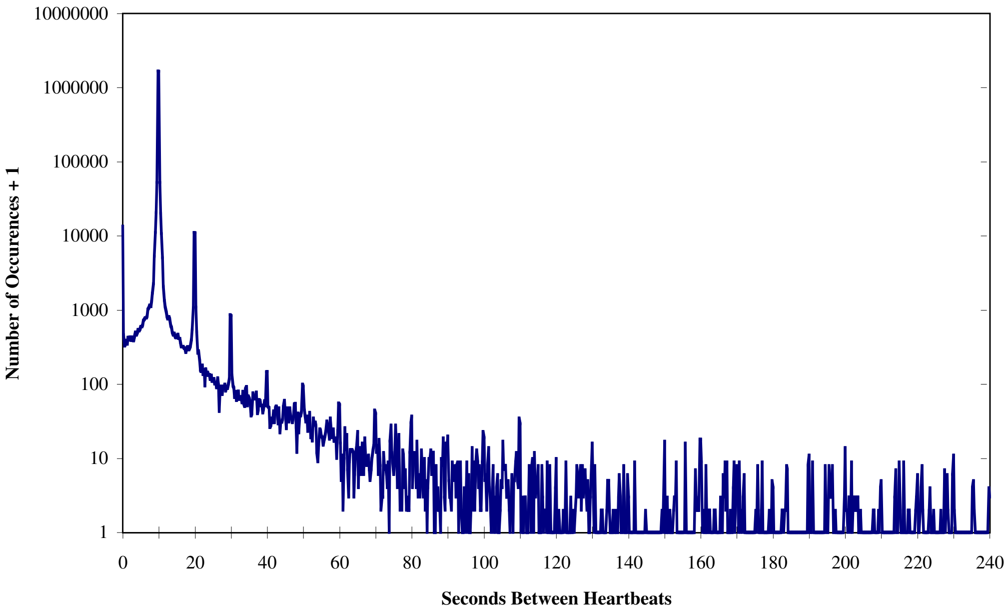

Figure 5. Histogram representation of all interarrival times observed at all nine hosts.

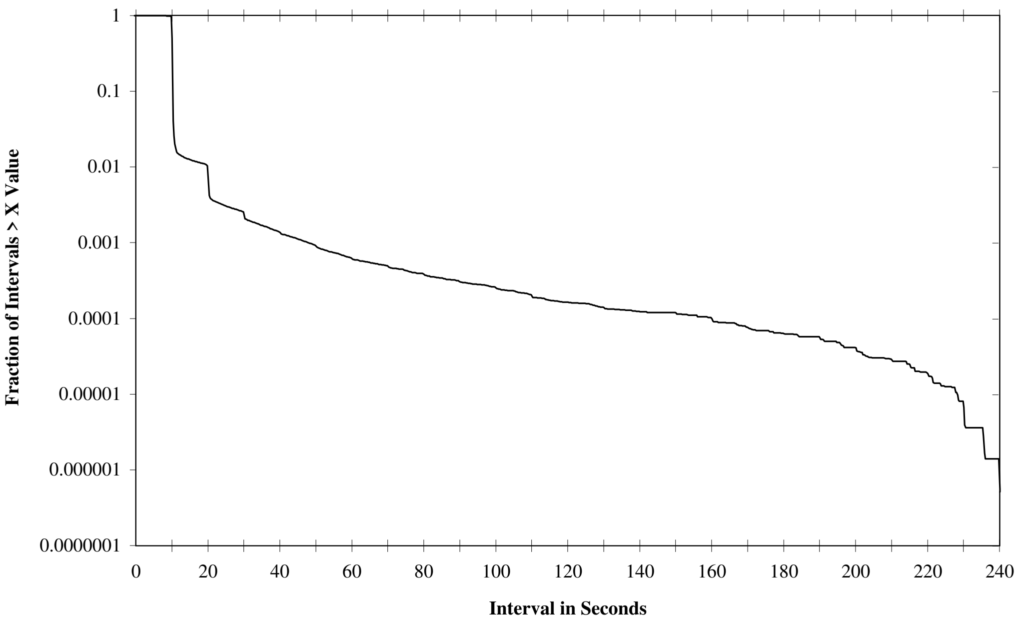

Figure 6. Interarrival times across all nine hosts, expressed in terms of proportion of heartbeats with interarrival time greater than X.

pute interarrival times. The results are summarized in four figures. Figures 3 and 4 present results from the point of view of a single host, bolas at ISI; Figures 5 and 6 provide results for all heartbeats received, summed across all hosts.

Figures 3 and 5 express results as a histogram, showing the frequency with which various interarrival times are observed. Because heartbeat packets are generated every 10 seconds, and messages losses are independent, we expect a large peak at 10 seconds (corresponding to packets that arrive with no intervening lost packet); a second step at an interarrival time of 20 seconds corresponding to one lost packet; and so on. This is more or less what we see, although because packets can be delayed both in the network (to a small extent: typical router queue delays are small) and by scheduling delays at the generating and receiving processors (especially if the processor is heavily loaded), the graph is not quite so regular.

Figures 4 and 6 express results in terms of the number of observed interarrival times greater than the time on the X axis. (Notice that these graphs hence plot the false positive rate that we can expect if we accept that X-axis value as our definition of "failure.") Figure 4 distinguishes the heartbeats received at bolas according to the originating host. Notice the wide variation in loss rates and in the extent to which network and scheduling delays skew the interarrival times. Skewing is particularly noticeable for data from the Convex Exemplar at Caltech ("CIT:neptune"). Examination of the interarrival times observed by the local data collector showed that this skew was due to delay in scheduling the local monitor, caused by high loads on the processor running the local manager.

Our results confirm the general trends observed in previous studies. The vast majority of interarrival times correspond to a packet that arrives with no intervening lost packet. If we define "failure" as corresponding to no heartbeat seen for 240 seconds, then the false positive rate is less than 1 in a 100,000. Furthermore, it appears that some of these "false positives" are due not to an unfortunate series of dropped packets, but to temporary losses in network connectivity that lasted several minutes: that is, events that we might well want to see signalled as "failures." More rapid failure detection is possible, with some increase in false positive rates: for example, at 35 seconds, the false positive rate is 1 in a few hundred. More frequent heartbeats would be required to reduce false positive rates at low interarrival times.

We also measured overheads associated with monitoring. The local monitors at the different hosts showed CPU utilization averaging under 1.3%, with minimum and maximum of 0.05% and 1.5%, respectively. Most of this utilization was due to children of the local monitors, which executed the system `ps` program to get the client status. The data collectors showed even lower utilization, averaging under 0.14%, with minimum and maximum of 0.06% and 0.29%, respectively.

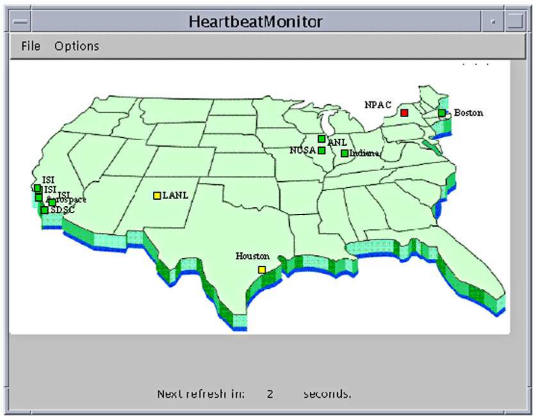

Figure 7. An interactive display tool for GUSTO fault data

6. HBM Applications

The HBM service has been used to monitor the health and status of critical components in GUSTO, a grid testbed that currently spans over 20 institutions [8]; and to detect server failure in a distributed computing system called NetSolve [4].

HBM is used in GUSTO by system administrators to identify problems in the testbed and to determine which testbed sites are functional. We are currently using a single data collector which can report its data through a web interface (e.g., see Figure 7) or via an email notification system. This application revealed that determining the status of arbitrary processes can be expensive on some large parallel computers. For example, a `ps` command without arguments on a loaded 512-processor Cray T3E can take 17 seconds of real time. This time can be decreased significantly by providing `ps` with the list of processes whose status is of interest, a change that has been made in the current version of HBM.

HBM is used within NetSolve to construct a application-specific restart-based fault recovery mechanism. NetSolve is a library that provides remote access to mathematical solvers and libraries such as ScaLAPACK. Client calls to a NetSolve-enabled library are forwarded to a NetSolve agent, which transparently dispatches the call to the most capable server that has the library installed. If either the server fails or the server or network becomes too slow, the NetSolve agent redispatches the request to another server.

NetSolve initially relied on the underlying network transport layer to notify it of broken connections. However, in many situations, this resulted in the NetSolve agent hanging. This behavior has been eliminated by modifying NetSolve to use HBM to detect server failure.

Only NetSolve servers register with HBM, while NetSolve agents act as data collectors, monitoring the status of the servers to which they have dispatched requests. Clients are not monitored at all. Currently, NetSolve shifts a computation over to a new server if a single heartbeat is missed. However, since the overhead of moving a problem to a new server is known to the agent, it would be possible to wait for additional missed heartbeats if the cost of resubmitting the request is high.

7 Related Work

Numerous distributed computing systems incorporate fault detection and recovery mechanisms. We review some of this work here, contrasting it with the approach described in this paper. Our work is distinguished primarily by its focus on providing a basic, flexible service that can be used to construct a range of application-specific fault behaviors.

Distributed systems, such as ISIS [1] and Horus [19] provide high reliability via replication and ordered group communication protocols. An ISIS-specific fault detection system, similar to HBM, is used to determine when a process or computer leaves a computation. However, this service is not made available to the application, nor is the application able to choose how faults are handled. Ordered group communication operations enable the construction of robust applications, but the use of replication as the only means of providing fault recovery makes ISIS unsuitable for some grid applications, especially extremely large computations requiring the use of multiple supercomputing resources.

Network batch queuing systems operate in distributed, networked environments and often use checkpoint and restart techniques to provide fault tolerance. Checkpointing is provided in systems such as CODINE [11] and in high-throughput systems such as Condor [13]. The fault behaviors for these systems are limited to checkpoint/restart with no support for more robust design such as replication.

In the Legion wide-area computing system [12], a hierarchy of "phoenix" demons is used to monitor system processes and restart them as needed. Application fault tolerance is provided using the Mentat macro dataflow model, in which objects can be replicated, with only one replica active and the remaining dormant. The first dormant replica is scheduled by the Legion system only if it does not receive a reply to a status query (ping) from the active replica within a prescribed number of seconds. While this approach suffices for the Mentat programming model, it assumes that system components function perfectly and that communication links do not fail.

In PVM [10] and the PVM-based SNIPE system [14], fault detection is performed by the PVM daemon which is located on each host, based on the receipt of responses (or not) from the daemons on other hosts. Application-level fault detection must be explicitly included in the application program code.

Weissman has compared different fault tolerance techniques for wide-area metacomputing [20]. In the Gallop system, each site runs a Scheduling Manager that actively discovers the status of local machines by "pinging" them. This monitoring function has been used to notify replication and checkpointing fault tolerance mechanisms, which were also compared. This approach detects some host and network failures, but does not detect the failure of hosts on which ping responses are generated independently by the network card. Furthermore, it does not detect either system or application process failures, and the scheduling managers are single points of failures for their sites.

8 Conclusions and Future Work

We have described the design and implementation of a fault detection service for high-performance distributed computing systems. By basing the service on unreliable fault detection and providing a clear separation between monitoring, detection, and response, we believe that we have succeeded in providing a flexible and efficient core service that supports a wide range of application requirements.

In our work to date, we have demonstrated that this service can be used by applications with minimal perturbation to application structure. We have also shown that this service has minimal impact on the load of the machine implementing the service, typically less then 1.5% CPU utilization. Furthermore, we have shown that the use of an unreliable fault detector does not prohibit an application from making sensible decisions, as the characteristics of the underlying transport mechanism ensure that the probability of error decreases with time.

While the HBM does well detecting process failure, discriminating between host and network failure is fundamentally difficult for any remote monitoring approach. If absolutely no information about a host is obtainable (e.g., a heartbeat) then it is impossible to decide whether it is due to host or network failure. However, we do not regard this as a significant problem for our applications, for which the knowledge that a host is unavailable is more important than knowing why.

We are working with various application groups and tool developers to investigate further the utility of this fault detection service. We expect to see a variety of application-specific fault recovery mechanisms implemented, including the use of replication and voting protocols. We also hope to explore the use of more rigorous fault tolerant behaviors such as reliable group communication primitives, and to investigate the relationship between heartbeat frequency, system overheads, failure criteria, and false positive rates. We are particularly interested in understanding the utility of adaptive techniques that modify either heartbeat rates or failure criteria in response to observed heartbeat loss behavior and system overheads.

Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; and by the National Science Foundation.

References

[1] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[2] J.-C. Bolot. Characterizing end-to-end packet delay and loss in the internet. *Journal of High-Speed Networks*, 2(3):305–323, 1993.

[3] M. S. Borella, D. Swider, S. Uludag, and G. Brewster. Analysis of end-to-end internet packet loss: Dependence and asymmetry. Technical Report 3Com Advanced Technologies Technical Report AT031798, 3Com Corporation, 1998.

[4] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical Report CS-95-313, University of Tennessee, Nov. 1995.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), Mar. 1996.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), Apr. 1982.

[8] I. Foster and C. Kesselman. The Globus project: A progress report. In *Proceedings of the Heterogeneous Computing Workshop*, 1998. to appear.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[11] G. S. GmbH. CODINE: Computing in distributed networked environments, 1995. http://www.genias.de/genias/english/codine.html.

[12] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-wide computing: Results using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, 1995.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[14] K. Moore, G. Fagg, A. Geist, and J. Dongarra. Scalable networked information processing environment (SNIPE). In *Proceedings of Supercomputing '97*, 1997.

[15] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), 1996.

[16] A. Mukherjee. On the dynamics and significance of low-frequency components of network load. *Internetworking: Research and Experience*, 5:163–205, 1994.

[17] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, U.C. Berkeley, 1997.

[18] S. Mullender (ed.). *Distributed Systems*. ACM Press, 1989.

[19] R. van Renesse, T. Hickey, and K. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, Cornell University, 1994.

[20] J. Weissman. Gallop: The benefits of wide-area computing for parallel processing. Technical report, University of Texas at San Antonio, 1997.