Robust Detection & Recovery from Service Disruptions in Distributed Systems

Hassan Rom

Carnegie Mellon University

May 4, 2007 23:54

Abstract

Distributed systems are complex to design, build and debug. Components crash due to software bugs such as unchecked array bounds, logic errors, and unchecked return codes. Components hang due to deadlocks and resource leaks. Instead of relying on failure-proofing the services which is rarely feasible, we focus on detecting and restarting failed components as an approach to improve system availability.

With the concepts and benefits of restarting in mind, I will present a *watchdog service* for detecting and recovering from failed components in a distributed system. As a case study, I will describe the design and implementation of a watchdog service for a distributed storage system called Ursa Minor. Experiences and novel extensions will be highlighted.

1 Introduction

Studies have shown that a large percentage of failures in systems are due to software faults [5]. Most software failures are caused by intermittent or transient bugs [4] that are hard to debug, especially if the bugs appear very infrequently. It is even harder to debug in large distributed systems, where different components have different states and may crash when those states mismatch. Worse, a crash of one component may result in cascading crashes of other components and even the whole system. Other failures such as due to natural disasters and hardware faults are unavoidable. Failures caused by human error, such as misconfiguration of systems, are claimed to be underreported [4].

Failure-proofing of each individual components is impractical if not impossible. When a new component is written or new hardware is installed, a rigourous robustness test of the distributed system as a whole needs to be rerun. A failure in one component may require the whole system to be restarted, which if not designed propery may require long hours for consistency checks. For commercial systems, this may result in loss of millions of dollars; for critical systems, this may result in loss of life. Training system operators can be expensive. As distributed systems grows in size, so does the number of system operators; resulting in higher likeliness of human error due to lack of coordination.

Studies have also shown the benefits of restarting as a mechanism for recovery from failure [6]. Restarting systems can also yield better performance due to reclaimation of leaked resources. Further, by restarting, components default to a clean state which also happens to be the most tested state, thus are less likely to fail.

In this paper, we choose to focus on the benefits of restarting over the impracticality of failure-proofing components. We present a clean abstraction for distributed system developers to focus on application code rather than worrying about reliability.

The remaining sections of the paper are organized as follows: In the next section, we will discuss related work on techniques to achieve high-availability in distributed systems as well as failure detection of components. In section 3, we will describe the architecture of the watchdog service. In section 4, we describe an implementation of the watchdog service in a distributed storage system called Ursa-minor. In section 5, we present an evaluation of our implementation. In section 6, we discuss our experience and limitations of our architecture. We conclude in section 6.

2 Related Work

Much work has been done that focuses on achieving high-availability via redundancy and fail-over mechanism. Although redundancy does offer performance benefits, it is often expensive. Most systems which use redundancy also require system operator intervention when bringing back up the failed component. In our architecture, recovery is automated.

Recent studies show that high-availability can also be achieved by recovery. For example, microrebooting [3] have shown that by isolating failures of components results in faster recovery time when compared to restarting of the whole distributed system. Our work is similar to that of microreboot and shadow drivers [7] but applied to a distributed storage system although our architecture is still applicable to distributed systems in general. Also, while microrebooting focuses on recovering from software-faults, our architecture could also recover from hardware failures.

Ironically, in some systems, cleanly shutting down components and reinitializing is slower than crashing and recovering [2], although in other systems such the earlier UNIX file systems, unexpected restarts can result in state inconsistencies. FSCK will need to be run on the file system in order to bring it back to a consistent state.

3 Architecture

The main goal of this current work is to provide a clean abstraction for distributed system developers who want to focus on application code and not worry about reliability code. With this in mind, we have identified four main properties that our architecture needs to hold.

1. *Reliable failure detection.* The architecture should reliably detect failed components.

2. *Recovery.* The architecture should support automatic restarts after a component failure has been detected.

3. *Isolation.* Failed components should not result in crashes of other components.

4. *Invisible.* The architecture should provide a simple abstraction for the components for realiable communication. Components should not need worry about retransmission and reestablishment of messages.

One major assumption of the architecture is that the components are crash-only. A crash-only software is defined to be software that only has one way of stopping it - by crashing it - and only one way to bring it up - by initiating recovery [2].

There are five main components in our architecture:

Messaging Layer :

As with Nooks [8], the messaging layer serves as a reliability layer for communication between components. The messaging layer also provides isolation of component failures.

Directory Service :

The messaging layer depends on the directory service for routing information lookup when sending a message. Since routing information is abstracted out by the directory service, components are free to restart on a new machine.

Resource Manager :

The resource manager is reponsible for managing shared resources across the distributed system. Since our assumption is the components of the system may crash at any point of its execution, shared resources and locks are leased in order to prevent from a crash components holding shared resources and locks for infinite amount of time.

Watchdog Service :

The watchdog service is responsible for detecting and restarting failed components.

Crash-only components :

All components in the system as well as our architecture specific components needs to be crash-only. This is a strict requirement for all components because the watchdog service may decide to restart any of the components of the system at anytime.

3.0.1 Messaging Layer

The messaging layer serves as a reliability layer in the distributed system. The purpose of the messaging layer is to provide an efficient and reliable means of communication between components. Network and machine connections, transports and failures are abstracted into a simple, unified semantic; the semantic being clients of the messaging layer are guaranteed to get a response from the messaging layer be it a success or a communication error. Thus, messaging layer is also responsible for reestablishing network communication betwen components after a restart. In the case of a communication error, the clients need not worry about retransmission since the messaging layer has already attempted that, so the only reasonable action is to propagate the error up to the caller. Since our main assumption is that the components are crash-only components, the messages meant for a crashed component will eventually reach the component after a restart. If this doesn't happen, then there's a bug in the recovery code of the component, which needs be fixed by the programmer.

The timeout value should be set to the maximum recovery time of the server plus the maximum request time. Timeout values can be dynamically set by analyzing workload patterns on the server. Timeouts should be invisible to client code.

Messages between components should also be entirely self-describing [2]. That way, on a component restart, the component can continue where it left off before it crashed. Messages should also carry information on its idempotency and time-to-live. Recovering from an idempotent message after a crash requires only reissuing of the message. For non-idempotent messages however, the component might need to roll-back or simply tolerate the inconsistency resulting from the replay.

To avoid retransmits and faster response time, the messaging layer also logs pending requests. On a restart, the messaging layer replays the logs.

Progress counters on the server side are also maintained in the messaging layer. The progress counter values are appended to each heartbeat(see below), which allows the watchdog service to decide whether or not the component is making progress. If the watchdog service determines that a component is not making progress, perhaps because it is hung, then the watchdog service restarts the component. The progress counters should also be invisible to components.

One possible measurement of progress is the number of request messages a component has received vs. the number of request complete messages the component has sent. If the watchdog service sees that the number of requests the component has finished remains the same while there are still pending requests for a certain period of time, then the watchdog service may decide to restart the component. While this statistic is easy to maintain, the watchdog service is not always able to identify whether or not a particular request is making progress since the thread that is handling that particular request might be hung while the other threads are servicing other requests just fine.

Another possible measurement of progress is the maximum in-flight request time. While the watchdog service is able to decide on a smaller granuality whether or not any particular request is making progress, obtaining the maximum in-flight requests time requires more work. Since the messaging layer is one of the most commonly executed pieces of code in the distributed system, adding more code to support this feature may decrease the performance of the distributed system as a whole by a significant amount.

3.0.2 Directory Service

The directory service provides a naming service for getting routing information that the messaging layer uses to send messages to a component. On startup of a component, the component reports its routing information to the directory service. If a component wants to send a message to another component, then the messaging layer of the sender will do a routing lookup of the receiver on the directory service. Routing information is periodically resent by the messaging layer of all components. This allows a component to be restarted on another machine if the machine it previously ran on experiences hardware problems.

The directory service is a component in the distributed system itself, hence is also monitored by the watchdog.

3.0.3 Resource Manager

All shared resources and locks in the distributed system are handled by the resource manager. Resources and locks are leased rather than held for unbounded periods.

Resources and locks that are held by a crashed component will eventually be reclaimed by the resource manager.

The resource manager is also a component in the distributed system and hence, is also monitored by the watchdog.

3.0.4 Watchdog Services

The primary purpose of the watchdog service is to detect service disruptions of components, whether the component has crashed or the component has hung. The watchdog service receives heartbeats and progress counters from all components in the distributed system. From the heartbeat and progress counters, the watchdog decides whether or not to restart the component.

Watchdog service is a component in the distributed system itself and could be a single point of failure. If the watchdog service crashes, the system is no longer reliable. As a remedy, a secondary watchdog service is needed to monitor the primary watchdog and vice versa. This secondary watchdog service could then be the next pooint of failure, but we address this issue by having the primary watchdog service monitor it; so, they watch and restart each other, as long as they don't both fail simultaneously.

3.0.5 Watchdog Clients Library

A watchdog client library is statically linked in each component. On initialization, the watchdog client library spawns a thread which periodically sends a heartbeat with progress counters from the messaging layer to the watchdog service.

Notice that the watchdog service need not know prior knowledge of the distributed system. A new component can simply be started and will integrate seamlessly with the current system without additional configuration.

The frequency of heartbeats presents an interesting availability-network utilization tradeoff. The less frequent heartbeats are sent, the higher the network utilization will be for inter-component communication. The more frequent heartbeats are sent, the lower network utilization will be for inter-component communication. For network sensitive applications, we suggest that the distributed system designer take this into account, although we doubt that the heartbeats will degrade network performance too much.

3.0.6 Crash-only components

All components in the system should assume that they may be restarted by the watchdog service at any point of its execution. It is reasonable to put this condition on the components since we are uncertain when the components might crash anyways, so the components might as well deal with it.

If the crash-only condition is met by the components, it can also be used towards its advantage, first for reliability and second potentially for performance. Components may experience resource leaks after running for long periods of time. As a result, performance may suffer. If the watchdog notices the performance

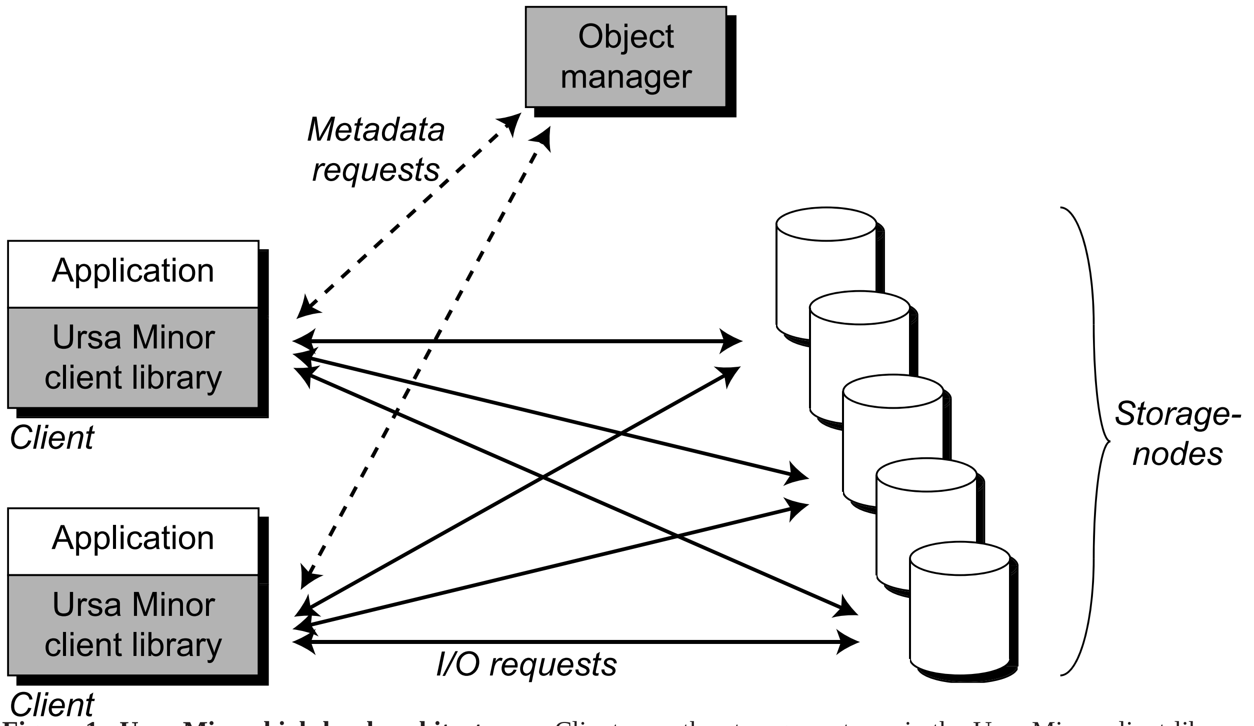

Figure 1: Ursa Minor high-level architecture. Clients use the storage system via the Ursa Minor client library. The metadata needed to access objects is retrieved from the object manager. Requests for data are then sent directly to storage-nodes.

degradation, then the watchdog could simply restart the component putting it into a clean state.

Non-volatile states of a crash-only application must be managed by a dedicated state store in order to make the component as stateless as possible [2], strictly leaving only the program logic in the component. This results in a much simpler and shorter recovery after a crash or a force restart by the watchdog service thus minimizing the downtime of the component.

On a restart, a component must restore its state from the dedicated stores. The components need not worry about reestablishing network communications, which is the responsibility of the messaging layer.

4 Implementation

Ursa-minor [1] is a distributed storage system that aims to be self-administrating, self-managing, ... Our initial motivation was to extend Ursa-minor to support automated service disruption detection and recovery, although our proposed architecture can be applied to any other distributed system.

Ursa-minor already implements its own directory service and messaging layer. The current implementation of the messaging layer in Ursa-minor handles reestablishment of network connection after a components restart although it lacks dynamic timeouts. Timeouts are exposed to the components. Ideally, timeouts should be hidden from the components in order to provide a nice abstraction of reliable communication between crash-only components. Currently, timeouts are set to a large number. Ursa-minor's current implementation also lacks logging of pending messages. Since this feature is a mere optimization over retransmission of messages, for the purposes of our current work, we safely ignored this feature.

Apart from the messaging layer and the directory service, Ursa-minor consists of 3 other core components; the metadata service, workers and the NFS servers.

Metadata service:

The metadata server is responsible for maintaining information about the backing of data which includes the location and encoding for all objects in the storage system.

Workers:

The workers are where all persistant data is stored.

NFS server:

The NFS server provides an interface for NFS clients to the storage system exported by ursa-minor. Communication between NFS clients and the NFS server doesn't go thru the messaging layer.

We made a minor modification to the messaging layer to add progress counters. Since the NFS server doesn't serve its client via the messaging layer, progress counters were added manually to the NFS server.

We added a watchdog service component to Ursa-minor, although we didn't implement a resource manager since there are no shared resources on locks between the components in ursa-minor. Also, a watchdog client library is embedded in all the components of Ursa-minor.

A considerable amount of work was done by the authors of the Ursa-minor components to make the components of Ursa-minor crash-only.

5 Evaluation and early experiences

We evaluate our work on three things. First on the robustness of the watchdog service detecting and restarting failed components. Second, on the crash-only-ness of the components and finally we evaluate on our experience making the components crash-only. At the end of this section, we discuss the limitations of our architecture.

5.1 Robustness

We used the postmark benchmark to evaluate robustness of our watchdog service. There are three phases of postmark; first the creation of files, second the reading and writing of files, and third, the deletion of files as

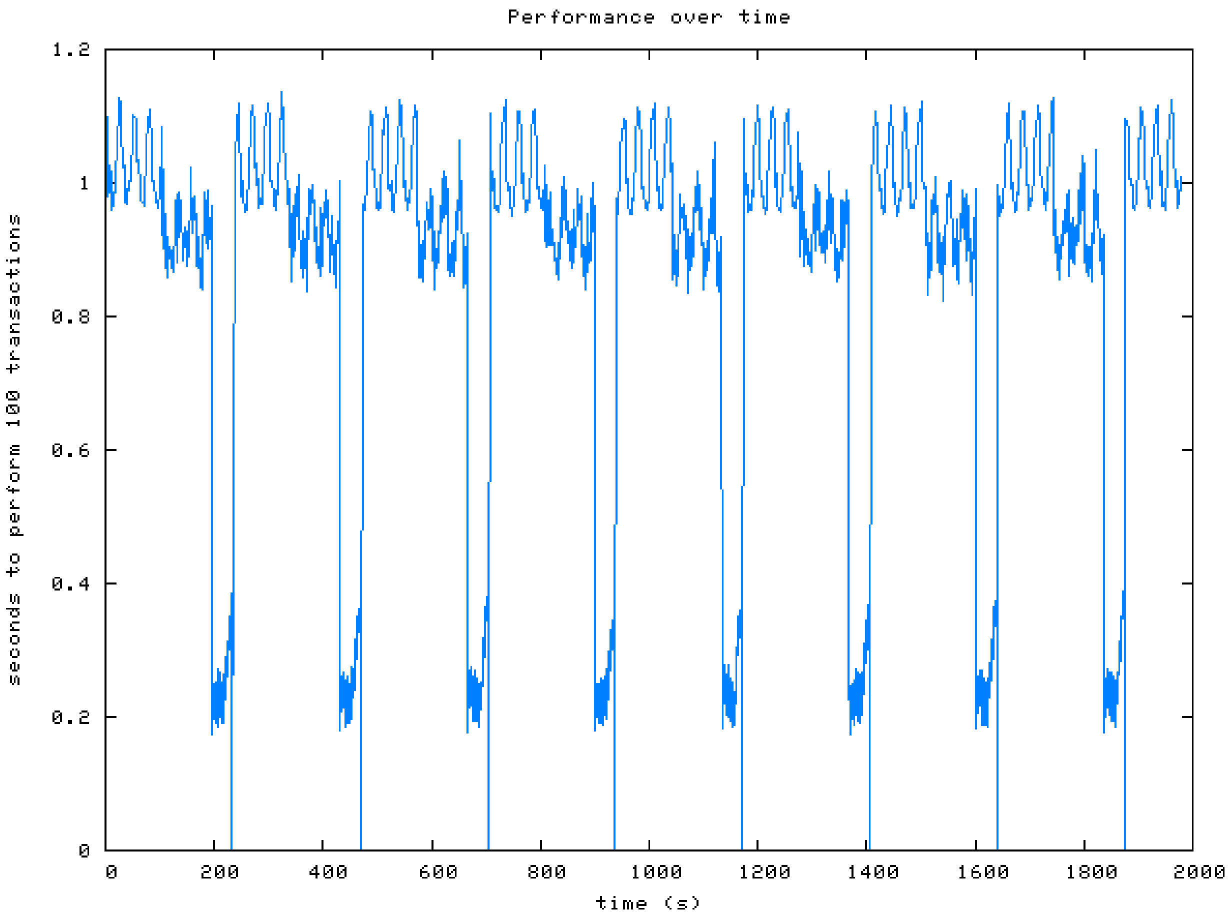

Figure 2: Few Iterations of Postmark Benchmark. The figure shows a graph of performance over time for eight and a half iterations of the postmark benchmark. Each iteration has a noticable three phases. For example, in the first iteration of postmark which is from time 0 to time 240s, the first phase of the benchmark is file creation, the second phase is reading and writing and the third phase is deletion of files.

shown in Figure 2.

We ran the postmark benchmark in a loop while randomly killing the components in Ursa-minor and observing whether or not the watchdog has successfully detected a component failure and restarting it. We also observed the impact of performance in postmark benchmark on a restart.

We conclude that the watchdog service was able to realiably detect component crashes and restarting them. In between a crash and a restart, we observed a small pause in the postmark benchmark.

5.2 Crash-only-ness

We evaluate the crash-only-ness of the components in our system by running postmark in a loop without injecting any faults. The result is shown in Figure 4.

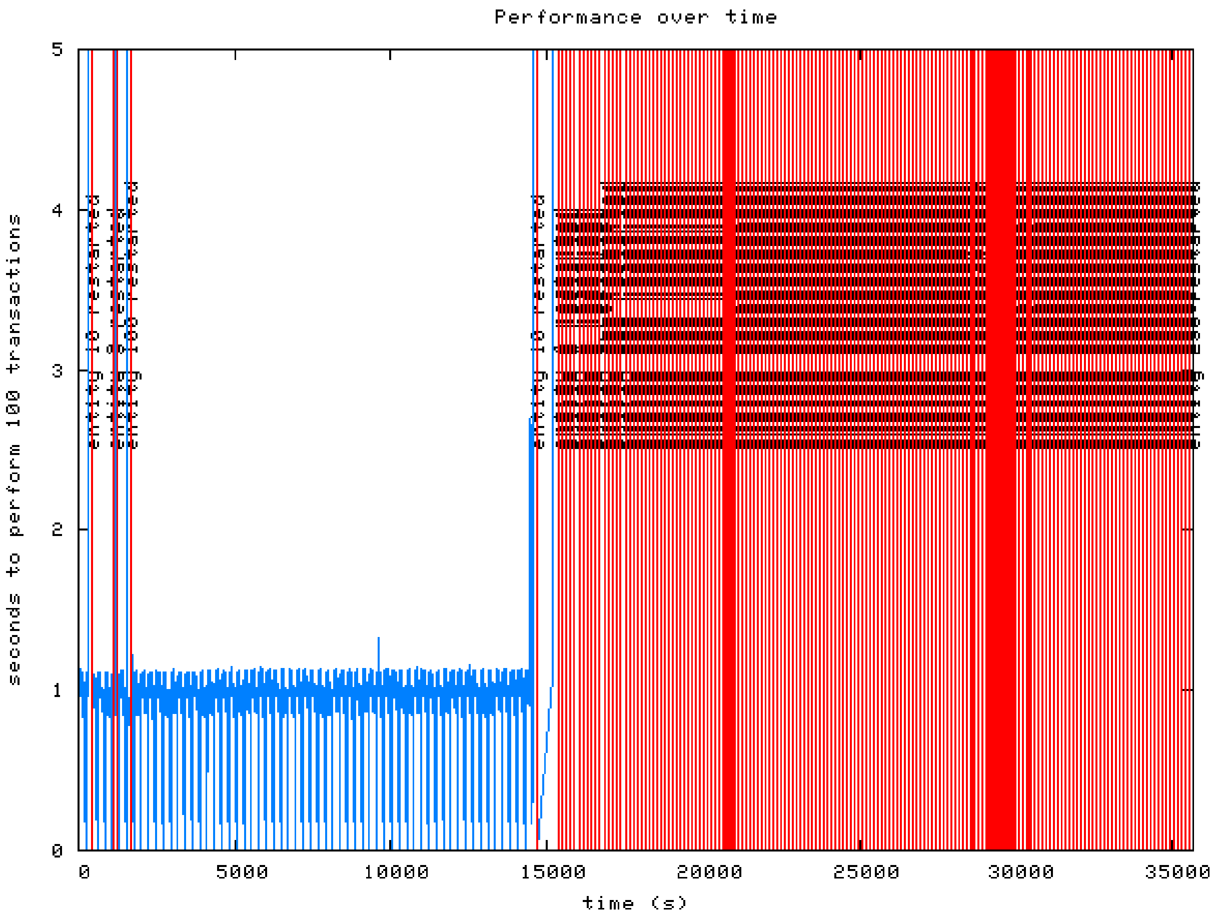

Figure 3: Failure detection robustness test.

This figure shows a manual test of injecting crashes on the components in Ursa-minor simply by sending a KILL -9 command on the components. The red vertical lines represents a restart of a component. The watchdog service succeeded on detecting all component crashes and the components are able to successfully restart after the manual killing of the components as shown by the first four red vertical lines.

In this test, we conclude that the MDS is not crash-only.

5.3 Experiences with achieving crash-only

In our experience of making existing non crash-only components into crash-only components, we conclude that it is not a trivial task. This is probably mainly due to software authors tending to put more focus on performance over restartability. We believe that to most software developers, improving performance is probably more intuitive to think about than restartability. But, we also believe that it would be much easier if the component author keeps the crash-only requirement from the very beginning of development. We also believe that it is easier to tweak a crash-only component to achieve better performance than to make components that are tweaked for better performance to be crash-only.

Contrary to software development principles, we believe that it is sometims fine to have bugs in code, since this results in more testing of recovery code which should be the primary focus of systems that focuses on robustness over performance. If the code reaches to a state where a component is not able to recover from a crash, then the author of the component should put more priority to fixing the recovery code than the bug that caused the crash of the component in the first place.

5.4 Limitations

Our architecture imposes two hard conditions on all the components. First, all components are required to be crash-only and second, all components need to communicate via the messaging layer.

6 Future Work

Since making existing components crash-only is hard, it would be interesting to explore on the possibility of using static and dynamic analysis tools to identify non-volatile states in components. Once identified, programmers can use this information to help them make their components crash-only.

Writing crash-only components is a non-trivial task. It would be interesting to explore how compilers can be used to abstract non-volatile state to ease the develoment of crash-only components. Compilers are also more knowledgable about the architecture level of the machine which makes them a better candidate for managing performance optimization.

It would also be interesting to see how we can apply techniques from machine learning to predict when a component would fail. The watchdog service can also be smarter about when to restart a component depending on the workload on the component. If the watchdog is predicting a component failure, by looking at the workload of the component, the watchdog might want to restart the component when the component is relatively not busy. This is beneficial if we want to deliver better user experience to end users of the distributed system.

7 Summary

In summary, for distributed systems in general, we believe that it time that robustness and reliability be the first priorities of current systems.

Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, and Symantec) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0326453 and #CCF-0621508, and by the Army Research Office, under agreement number DAAD19–02–1–0389.

References

[1] Michael Abd-El-Malek, Garth R. Goodson, Gregory R. Ganger, Michael K. Reiter, and Jay J. Wylie. Lazy verification in fault-tolerant distributed storage systems. *Symposium on Reliable Distributed Systems* (Orlando, FL, 26–28 October 2005). IEEE, 2005.

[2] George Candea and Armando Fox. Crash only software. *Hot Topics in Operating Systems* (Lihue, HI, 18–21 May 2003), pages 61–66. USENIX Association, 2003.

[3] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot—a technique for cheap recovery. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 06–08 December 2004), pages 31–44. USENIX Association, 2004.

[4] Jim Gray. *Why do computers stop and what can be done about it?* Tandem Technical report 85.7. June 1985.

[5] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, **24**(9):39–48, September 1991.

[6] Mark Sullivan and Michael Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems, June 1991.

[7] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 06–08 December 2004), pages 1–16. USENIX Association, 2004.

[8] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Symposium on Operating System Principles* (Lake George, NY, 19–22 October 2003), pages 207–222. ACM, 2003.

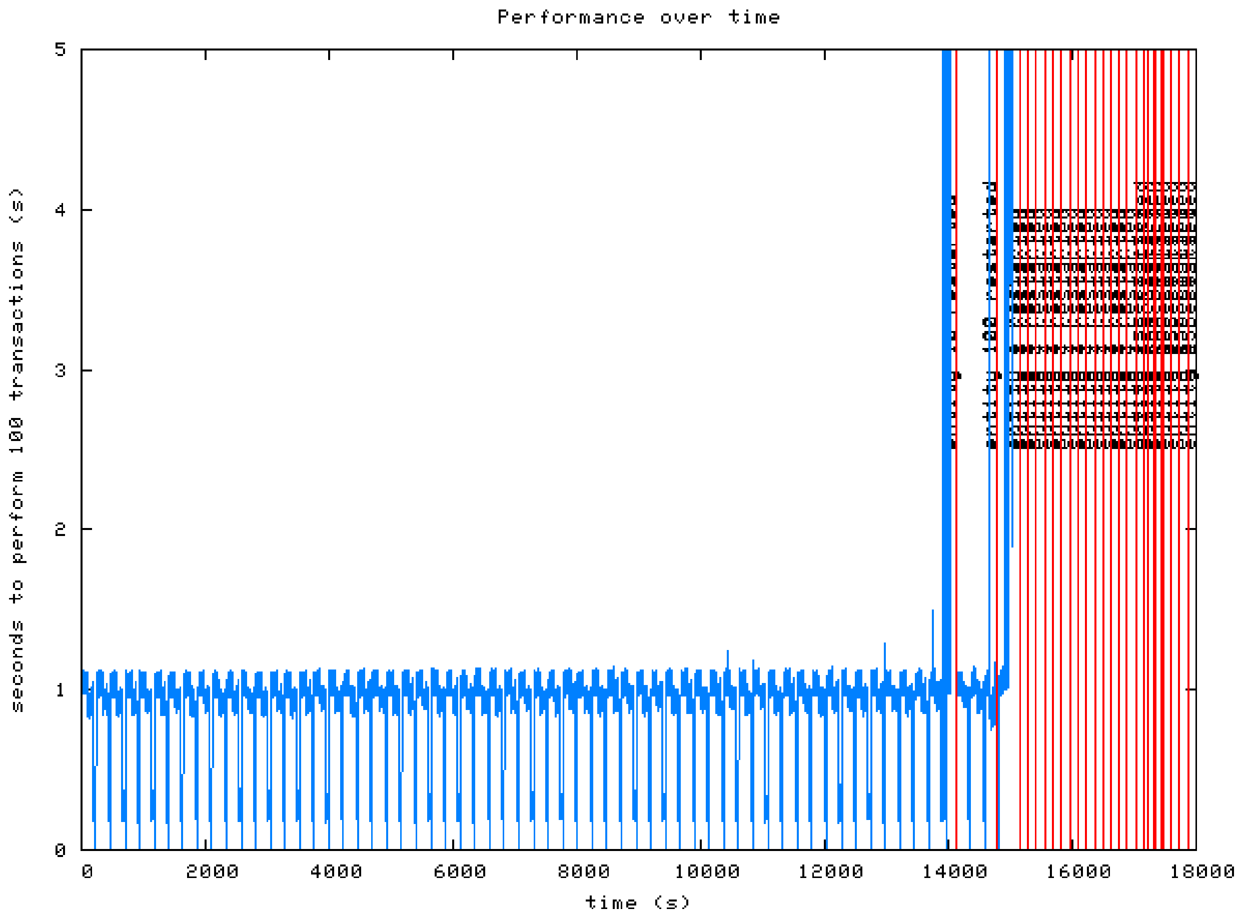

Figure 4: Crash-only-ness test.

In this test, we ran postmark without any fault injection. In this instance, the first component to crash was the NFS server. The watchdog service sucessfully detected the failure and successfully restarted the NFS server. The next component to crash was the worker. Again, the watchdog service successfully detected the failure and successfully restarted the worker. Unfortunately, the next component to crash was the MDS server, but wasn't able to restart successfully. The watchdog repeatedly tried to restart the MDS but to no avail. We conclude that the MDS is not crash-only.