Agile Store: Experience with Quorum-Based Data Replication Techniques for Adaptive Byzantine Fault Tolerance*

Lei Kong[†], Deepak J. Manohar[†], Arun Subbiah[‡], Michael Sun[‡], Mustaque Ahamad[†], Douglas M. Blough[‡]
[†] College of Computing and [‡] School of Electrical and Computer Engineering
Georgia Institute of Technology; Atlanta, GA 30332 USA
{konglei,mjdeepak,mustaq}@cc.gatech.edu,{arun,msun,dblough}@ece.gatech.edu

Abstract

Quorum protocols offer several benefits when used to maintain replicated data but techniques for reducing overheads associated with them have not been explored in detail. It is desirable that a system be able to adapt its operation so that fault tolerance related overheads are only incurred when the protocol execution actually encounters faults. There are a number of issues that need to be carefully examined to achieve such agility of quorum based systems. We make use of a file system prototype, developed in our Agile Store project, to experimentally evaluate several techniques that are important for efficient implementation of Byzantine fault-tolerant quorum protocols. We present an optimistic quorum collection scheme and a probabilistic hashing scheme for determining the response to a quorum request, and show that they lead to significant performance improvements. The Agile Store also makes use of reconfigurable quorum techniques to allow system size and fault threshold to be dynamically varied when, for example, faulty servers are removed, new servers are added, or the threat level is changed. We quantify the performance gains made possible by such reconfiguration of quorum parameters. We also show how performance scales with different system parameters and how it is affected by design choices such as whether to use proxies. We believe that the results in the paper provide important insights into how to implement quorum protocols to provide good performance while achieving Byzantine fault tolerance.

1. Introduction

As computing and communication devices become pervasive, new information rich applications will be deployed in environments that range from home and the community to enterprises. For example, the Aware Home project [2] at Georgia Tech, which is exploring future applications in the home environment, deploys a variety of sensors and computational devices to capture and manipulate information about the home's residents and their activities. Such information is essential for enabling applications that assist elderly residents in their daily activities and in case of emergencies. Clearly, a service that is entrusted to store data created in the Aware Home will have to provide confidentiality for personal information. At the same time, high availability and timely access are essential for critical information that may be needed in emergency situations. We believe that future applications will create, access and manipulate information that has a variety of dependability and consistency requirements, while at the same time demanding high performance. Our research project focuses on the design, implementation and evaluation of *Agile Store*, a storage service that can meet such diverse needs.

This *agile* approach is explored in the context of file system services in this paper. Our file system prototype is available to the user with an NFS interface, but the back end is a distributed set of servers that provide scalable and dependable data storage services. Thus, the *agile* file system service is provided transparently to user applications. The performance of the file system can be measured using two metrics - the time taken to read and write files (latency), and the throughput. Not surprisingly, these performance metrics are in direct conflict with the dependability of the system, measured by the number of nodes that can fail or be compromised before the service guarantees are violated. Our approach is to acknowledge the trade off between performance and dependability and exploit it under the spectrum of operating conditions experienced by a given system.

Most works assume that not more than b servers fail or are compromised during the entire operational life time of a system. We instead use a fault threshold only to mask faults for a certain period of time, but not as an upper bound on the total number of faults or compromises that can take

*This research was supported by the National Science Foundation under Grant CCR-0208655.

place in the system over its lifetime. Our solution incorporates a fault detection algorithm that detects faulty servers and reconfigures the system to exclude them without any disruption to the storage service [14]. Through timely detection and removal of faulty servers, the requirement that not more than b compromised servers are present in the system can be maintained as long as possible. Intrusion detection mechanisms help us in estimating a safe fault threshold setting based on perceived threat conditions. In order to take advantage of this approach, the read and write protocols of the Agile Store adapt themselves based on the current fault threshold and the current system size [14].

The overhead of Byzantine quorum protocols increases rapidly with the number of servers and the fault threshold. We focus in this paper on several optimizations that can significantly improve their performance. Note that maintaining a low fault threshold, as is enabled by our basic reconfigurable Byzantine quorum approach [14], is important for achieving good performance. However, performance measurements from our file system prototype have shown that this, by itself, is not sufficient. The first optimization we present is *optimistic quorum collection*. Since fault thresholds must account for worst-case situations, they are inevitably pessimistic under normal operating conditions. In our optimistic approach, the read protocol initially attempts to complete a read operation using a subset of a quorum under an optimistic assumption about the current number of faults. Using quorum properties, it is possible to detect when the optimistic assumption is violated in the initial read attempt and, in this case, a second round of the read is performed to finish contacting a full quorum. This allows performance to degrade gracefully with the *actual* number of faults present, instead of being dependent on worst-case limits or a pessimistic view of the current number of faults. A second optimization we consider is *adaptive probabilistic hashing*. In this approach, data servers probabilistically return either a complete data block or a hash of the data block. By adapting the probability with which servers return a hash, system performance can be tuned to the current operating environment.

In order to evaluate the performance of our basic approach and its optimizations, Emulab [23] facilities were used to gather extensive performance measurements from our agile file system prototype. We also used this experimental set-up to investigate the performance of Byzantine quorum protocols more generally, considering their scalability, the impact of design choices such as proxies, and how well they compare against the non-fault-tolerant NFS v3 file system. Results show that, with our optimizations, single Byzantine fault tolerance can be achieved with 5 servers and an overhead of less than 10%, as compared to NFS v3.

2. Related Work

A number of projects [3, 6, 7, 8, 19] have addressed dependable storage services that rely on data replication. Quorum based protocols [10] represent one popular replication technique used by several such systems [12, 17, 19]. In quorum systems, reads and writes are performed on a quorum or a subset of all servers. By requiring that any two quorums overlap, at least safe variable semantics [16] can be provided. Safe variable semantics provides single-register consistency guarantees as long as read operations to a data object (variable) are not concurrent with write operations on the object. To tolerate Byzantine server faults and server compromises, Byzantine quorum systems [18] were developed. Here, the overlap between any two quorums is increased sufficiently to mask arbitrary actions by faulty servers. While there are different types of Byzantine quorum constructions, threshold masking quorums, which we consider herein, are widely studied because of their simplicity and high availability.

To mask faults, fault tolerant protocols need to execute expensive operations, despite the fact that such faults are not present most of the time. In dynamic quorum systems [4], varying the fault threshold (the maximum number of faults that can be masked) was proposed to reduce overheads associated with such protocols. The idea is to increase the fault threshold only when the situation requires it, and to operate at a lower fault threshold at other times. In reconfigurable Byzantine quorum systems [14], we added the capability to remove faulty servers while the system is in operation, which helps to keep the fault threshold low. A framework that allows reconfiguration of arbitrary Byzantine quorums was reported in [20].

Besides dynamic threshold adjustment, we use an optimistic quorum protocol for reads, where a subset of a read quorum is first probed, and only if required, a second round is executed wherein the remaining servers in a full quorum are queried. To the best of our knowledge, this is the first quorum collection technique that is optimized for the common case where few faults are present and adapts to the actual fault number without changing the fault threshold. Other kinds of optimistic behavior have been studied in the context of storage systems. For example, S. Frolund, et al., have a two step read that tolerates crash failures in [9]. It is optimistically assumed that there are no partial writes and a light weight read is attempted at first. A more expensive recovery process is executed when the assumption does not hold. In dynamic Byzantine quorums [4], write quorums change dynamically. A read operation first reads from a small read quorum, optimistically assuming that it has sufficient intersection with the latest write quorum of the object being read, and if this is not true, it reads from a larger quorum.

Many works, e.g. [7], have attempted to reduce the network communication overhead as a means to achieve efficiency. This is typically achieved by communicating the hash of the data instead of the data itself, which is typically much larger. In [22], this has been explored in the context of Byzantine-tolerant coordination protocols which use Byzantine quorum systems, but lacks a detailed performance evaluation. In this paper, we provide a detailed performance evaluation to demonstrate the benefits of hashing in the context of quorum protocols in local area and wide area network settings.

In [11], optimistic techniques and cryptographic hashes are used in the context of storing erasure-coded data using versioning and quorum systems. In this paper, we investigate optimistic quorum collection techniques and *probabilistic* hashing for improving the efficiency of the original read and write protocols for Byzantine quorum systems given in [18]. The techniques presented are sufficiently general to be used in other quorum read-write protocols. In [11] however, the optimistic techniques and cryptographic hashes are used for easy detection and repair of malformed writes due to faulty clients and servers, and to achieve linearizability and wait-freedom of read-write operations when versioning is used along with erasure-coding. In contrast, the read-write protocols given in [18] consider only replicated data and do not use versioning. Only safe-variable semantics are provided. We do not consider the possibility of partial and inconsistent writes by malicious clients, as these can be overcome by incorporating reliable broadcast protocols into writes.

3. The Agile Store Prototype

3.1. System Model

Broadly speaking, the system consists of three main entities: *servers* that provide the file system service, *clients* that use the file system, and a *fault detection service* which is responsible for reconfiguring the system as part of providing agility. A Byzantine fault model is assumed for the servers. It is assumed that not more than a threshold number of servers are Byzantine faulty at any given time. Some measures are taken to deal with faulty clients, but as in prior work on Byzantine quorum systems, our primary focus is on server faults. Our read/write protocols and fault detection and reconfiguration algorithms operate under the assumption of an asynchronous network. However, the access control mechanism in our file system prototype requires loose clock synchronization among the servers. Since Byzantine-tolerant access control is not the focus of the project, we achieved clock synchronization in our prototype using the Network Time Protocol (NTP) [1]. Throughout the paper, we use the term "system size" to refer to the number of data servers in the system.

3.2. Architecture of File System Prototype

Figure 1 illustrates the main architectural components of the file system prototype. The client agent is a user-space application that exports an NFS file server interface to the client machine. The client agent fulfills NFS requests by interacting with the metadata service and the data servers. A small subset of NFS functionality (symbolic links, hard links, dynamic file system information) is not implemented. File system consistency mimics that of NFS V3, a stateless protocol with loosely enforced open-to-close file semantics.

The metadata service consists of a distributed set of servers designated to handle the bulk of the file system's metadata needs. Metadata belongs to a class of information that is usually modified from its current state, but is rarely overwritten with data that is independent of its earlier contents. Since strict ordering requirements must be enforced on metadata operations, the metadata service is implemented as a Byzantine-Fault Tolerant (BFT) state machine [21]. Our prototype uses the Castro–Liskov protocol [7] to achieve this. It is responsible for managing the mapping of the file system directory structure to the flat directory namespace used at the data servers, as well as directory attributes and access control information.

Certain types of metadata such as file sizes, file access times, file modification times, and attribute modification times are best managed at the data servers because they do not require strict ordering and are modified frequently (on every file block read/write request). Read requests also implicitly require a data server to perform certain file attribute modifications, i.e. updates on access times. Reads and writes occur on file blocks, which are 8 KB in size.

The data servers are comprised of a distributed set of servers from which file data and attributes are read and to which they are written. A client agent randomly chooses a data server as its proxy server, and the proxy server forwards its request to a quorum of data servers. Responses from the data servers are channeled back to the proxy and then sent back to the client agent. The quorum and the proxy server are randomly chosen by the client agent for *every* request. Note that a proxy server is a data server and, according to our system model, it can suffer from a Byzantine fault. In order to protect the integrity of reads and writes despite the possibility of a faulty proxy, message authentication codes (MACs) are used on requests and responses. Thus, individual data servers verify a client-generated MAC before processing a request and the client verifies a server-generated MAC before processing the server's response. A proxy server can drop requests and/or responses but can not undetectably modify them. A client that does not receive enough verified responses simply retries the operation us-

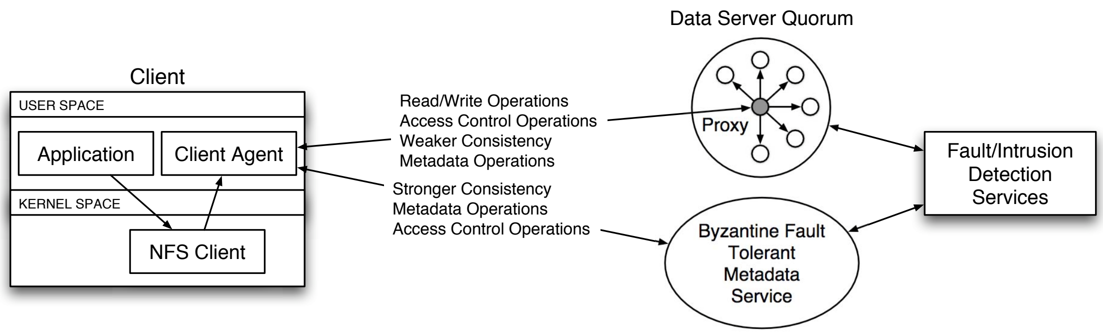

Figure 1. Schematic overview of the file system prototype

ing a different proxy.

Upon opening a file for read access, a client agent obtains the file handle from the metadata service. The client agent proceeds to read file blocks from the data servers by means of quorum read protocols. The file size, read time, access time, and attribute creation time are also read from the data servers. A write session occurs in an analogous manner with quorum write protocols governing the interaction between the client agent and data servers.

The *agile* system continuously adapts its operating point so that a balance between performance and dependability is achieved. This operating point is varied using a system parameter called the *fault threshold*. If the fault threshold is set conservatively to a large value, system performance will suffer. If it is too small, the system will suffer in its robustness to faults. The fault and intrusion detection services include feedback mechanisms from which the fault threshold is set and the system reconfigured. Section 3.3.1 further discusses these feedback mechanisms.

3.3. Algorithms and Protocols

3.3.1. Reconfigurable Byzantine Quorum Systems

In dynamic Byzantine quorum systems [4], the fault threshold for masking quorum systems is allowed to change dynamically based on information about the current fault environment. In [14], we introduced the technique of reconfigurable Byzantine quorum systems, where both the system size and fault threshold can be adjusted dynamically. Read and write operations on data objects are executed according to the current system parameter values. Reconfigurable Byzantine quorum systems include read and write protocols that adapt to changes in system parameters, and fault detection protocols that enable the system to remove data servers that are identified as faulty or compromised. The read and write protocols are similar to those of dynamic Byzantine quorum systems but extended to deal with varying system size. Full details of the protocols can be found in [14].

The fault detection algorithm described in [14] is used to detect compromised data servers. The use of proxy servers (see Section 3.2) in the Agile Store architecture enables servers to monitor other servers' responses to read requests. Each individual server monitors sequences of responses from other servers and uses a statistical approach to identify servers that are likely to have failed or been compromised. This statistical approach takes into account the likelihood that servers might respond incorrectly because they were not part of the most recent write quorum and the fact that clients choose quorums at random in our approach. The detection algorithm has a high probability of detection with low false alarm rate even with a moderate percentage of concurrent write operations. A fault detection service, which connects directly to servers and does not otherwise accept external connections, gathers detection results from individual servers and executes a voting algorithm to make the final determination on when to remove a data server. The fault detection service is responsible for removing the data server and updating system parameters accordingly. The fault detection service can be executed as a Byzantine fault-tolerant state machine [7, 21]. Feedback is also provided by a specification-based intrusion detection service (IDS). The IDS has been designed to focus on detecting abnormal usage of the file system and adapting the fault threshold accordingly. A central IDS detects system-wide attacks on the system while IDS sensors at each server detect attacks in individual servers. A full description of the IDS service and its implementation is beyond the scope of this paper.

3.3.2. Optimistic Quorum Collection

In the Agile Store, agility is not only expressed in its ability to dynamically adjust the fault threshold, but also in its ability to tune system performance based on the actual number of faults present [15]. In optimistic quorum collection, we achieve this by optimistically assuming a smaller number of faulty servers is present among a read quorum than the actual fault threshold. We use this optimistic assumption to initially query fewer servers than in the case of a full read quorum.

In threshold masking Byzantine quorum systems, a read quorum intersects with any write quorum in at least $2b + 1$ servers. When there are no faulty servers, these $2b + 1$ servers all return the latest value of the object being read (assuming there is no concurrent write operation on the object). Since $b+1$ servers are enough to guarantee that the returned value was produced by a past write operation, querying b extra servers is a waste in a fault free scenario. A simple version of our optimistic approach is realized in what we call "blind" optimistic quorum collection. During a read operation, the read quorum size minus b servers are first queried. When fewer than $b+1$ servers return the latest value, which can be caused by faults among queried servers or by concurrent writes, b more servers are queried to complete a full read quorum collection.

The above "blind" optimistic quorum collection finishes in one round with b fewer servers queried, when no faults are encountered. The risk taken is that a read quorum might have to be collected in two rounds. Although this does not lead to higher server workload or lower throughput, read latency might be higher than that of a single round full quorum collection.

A more general optimistic collection scheme utilizes an estimate of the current number of actual faults to derive an estimate of the number of faults that are expected in the servers contacted during the initial read attempt. This value is used as the optimistic "threshold" for read operations. When there are fewer faults than estimated, the optimistic assumption holds true and the reads still terminate faster and server workload is lower due to the use of fewer servers than in the case of a full quorum read. We can detect during the initial phase of the read if the optimistic assumption is violated, and if so, query additional servers to bring the total number of responses to a full read quorum. In this way, the correctness of the approach is not compromised even when the optimistic assumption does not hold.

With this generalization, the optimistic quorum collection approach becomes similar in certain respects to dynamic Byzantine quorums [4], but *does not rely on accuracy of the current fault estimate for correctness*. Only performance is penalized when the estimation is not accurate. In Section 4.3, we demonstrate that optimistic quorum collection achieves similar performance to dynamic Byzantine quorum systems without the need for an accurate fault estimate assumption for correct operation.

Figure 2 shows the pseudo-code for an optimistic read quorum collection strategy within a b-masking Byzantine quorum system [18]. δ refers to an estimate of the number of faulty servers among a read quorum, q_r is the read quorum size, and b is the fault threshold. $\delta = 0$ represents the blind optimistic quorum collection scheme. If an estimate of the number of faulty servers in the system is available, denoted by f, we set $\delta = \lceil \frac{f(q_r-b)}{n-f} \rceil$, which represents the expected number of faults in a read set of size $q_r - b + \delta$. Another strategy would be to dynamically adapt the value of δ so that a target percentage of read operations in which the optimistic assumption is met is achieved. Our proxy-based fault detection [14] or statistical methods reported in [5] can be used to estimate the number of faults present, but more efficient methods might also be possible since we do not rely on the estimate for correctness.

A faulty server can respond with corrupted data or simply refuse to respond. The latter case will cause a client side timeout. When a timeout occurs, a client must query more servers in order to collect the required number of responses. To simplify the protocol description, we omit the details of this in the pseudo-code.

Theorem 1. *Applied to any b-masking Byzantine quorum system [18], optimistic quorum collection with $0 \leq \delta \leq b$ maintains safe variable semantics [16], i.e. when there are no concurrent write operations on the data object being read, a read operation returns the value of the most recently completed write in some serialization of preceding writes.*

Proof. First, assume there are no write operations concurrent with a given read operation. Suppose a data object A is being read, and when A was written most recently, the write quorum was Q_w and $< v, t >$ is the value and timestamp pair written. In optimistic quorum collection, suppose responses collected in Round 1 are from a server set S_1. Then $|S_1| = q_r - b + \delta$, where q_r is the read quorum size. Given this is a b-masking Byzantine quorum system, $|Q_w| + q_r - n \geq 2b + 1$. It follows that, $|Q_w| + (q_r - b + \delta) - n \geq b + 1 + \delta$. This means that at least $b + 1 + \delta$ responses from S_1 were also part of Q_w. Assuming there are no concurrent write operations on object A, then the client will get at least $\delta + 1$ copies of $< v, t >$ in Round 1.

When the number of faults in S_1 does not exceed δ, the client will get at least $b + 1$ copies of $< v, t >$ in Round 1. Any response with a timestamp higher than t is obviously faulty and can not be returned by more than δ servers in this case. Therefore, the client will return $< v, t >$ as the result in Round 1. Legitimate but older values of A cannot be returned as the read result in Round 1, because at least $\delta + 1$ copies of $< v, t >$ are collected.

The protocol will execute Round 2 if it does not return $< v, t >$ in Round 1. In Round 2, $b-\delta$ additional responses are collected, which completes a full read quorum together with the $q_r - b + \delta$ responses collected in Round 1. In this case, $< v, t >$ will be returned as the result, as guaranteed by the

```
Round 1: {Fast Path}
Query $q_r - b + \delta$ servers for the object O.
If there is a highest timestamp winner then
  Return highest timestamp winner {Fault free case}
Else If there is a winner then
      If more than $\delta$ responses have higher timestamp
        than the winner then
        Perform Fault Handling Round {Faults present}
      Else Return winner {Faults present but masked}
      EndIf
    Else Perform Fault Handling Round {Faults present}
    EndIf
EndIf

Round 2: {Fault Handling Round or Slow path}
Query $b - \delta$ more servers for the same object O
If there is a winner among responses from
both Rounds 1 and 2 then
  Return winner
Else Return Null {Concurrent write}
EndIf
```

Response: A <value, timestamp> of the data object being read, which is returned by a server.

Candidate: A candidate is a response returned by at least $b + 1$ different servers in a given read operation.

Winner: A winner is defined as the candidate that has the highest timestamp among all candidates that have been received in a particular read attempt.

Highest timestamp winner: A winner is also a highest timestamp winner if the winner has the highest timestamp among all responses in a particular read attempt.

Figure 2. Optimistic Read Protocol

properties of b-masking Byzantine quorum systems. □

The optimistic approach not only performs better in a system with faults, but performance degrades gracefully as the number of actual faults increases up to the fault threshold. This is due to the fact that quorums are chosen at random and, therefore, the probability that the optimistic assumption is violated increases monotonically as the actual number of faults increases beyond the assumed value. In Section 4.3, we provide detailed performance results demonstrating the benefits of the optimistic approach.

3.3.3. Adaptive Probabilistic Hashing

Fault tolerance in replicated systems comes at a price, i.e. multiple copies of a data object need to be received by a client in order to compute a safe result. This communication overhead can be reduced by making use of hashes, assuming that the data object size is much larger than the hash size. Cryptographically strong hashes can be treated equivalently as their data objects for verification purposes in read result voting. In [7], a client picks one server to return the full data object requested and all others return a hash of the data object. This approach does not work well with Byzantine quorum systems because of their inherent data inconsistency. Writes happen to quorums, not to all servers. If the server picked to return the full data object does not have the latest version, then the read will fail, even when the chosen server is not faulty.

To address this problem particular to quorum systems, we adopt a probabilistic approach. Each server in a read quorum returns the hash of the requested data object with some probability p_h and it returns the data object itself with probability $1 - p_h$. Intuitively, the larger p_h is, the less data is transferred. However, a large p_h leads to a high probability of failure, i.e. that no server returns the latest version of the data object. If this happens, the client will execute a second round where it requests the full data object from one of the servers that returned the correct hash value previously. Figure 3 gives the pseudo code executed by clients.

During execution of the system, the hash probability can be adapted in order to optimize different performance metrics, e.g. client latency or system throughput. This can be done based on analytical derivations of these quantities given measurable parameters such as round-trip time and available bandwidth, or on hill-climbing techniques within a feedback control loop where changes are made to the hash probability and the performance metrics are directly measured, or on some combination of these approaches. Adaptation of the hash probability in this manner allows the system to maintain best performance as system conditions change. In Section 4.4, we present some initial results showing how this can be done when the goal is to minimize latency.

4. Experiments and Results

In this section, we present a detailed performance evaluation of the Agile Store prototype. Micro-benchmarks and

```
Query servers in a quorum Q for the object O
and await either hashes or data
values or a combination of the two.

Case a: Only data values received

If there is a winner (defined in Figure 2) then
   Return winner
Else Return Null {Concurrent write}
EndIf

Case b: Mixture of hashes and data objects received
{This pseudocode is optimized
for clarity rather than performance}
1. Compute C(O), hash of each object O returned
2. Compare each response and of those H or C(O)
   that were returned by at least $b + 1$ servers
   add those H or C(O) with the highest TS to set HH.
3. If there is a C(O) in HH then
      Return the data object O corresponding to C(O)
   Else
      For each server that has its H in HH
         a. Query the whole data object O from
            the server
         b. Compute the hash of the returned
            object O in C(O)
         c. If C(O) matches with H then
               Return the data object O
               Exit
            EndIf
      EndFor
   EndIf
4. If no value returned then
      Return Null {Concurrent write}
   EndIf
```

Figure 3. Probabilistic Hash Read Protocol

the widely-used Andrew benchmark are used to evaluate the prototype and its realization of the techniques discussed in Section 3.3. Our evaluations focus on failure-free performance (as is customary in the field), except where presence of faulty servers is essential to the experiment.

All experiments were performed in Emulab [23]. Unless otherwise explicitly noted, experiments were performed in a 100 Mbps switched Ethernet environment. Our machine testbed consisted of Intel PIII 600 MHz processors with 256 MB RAM and 13 GB 7200 RPM IDE hard disks, and Intel PIII 850 MHz processors with 512 MB RAM and 40 GB 7200 RPM IDE hard disks. All machines ran RedHat Linux 7.3. 1 MB files were used to measure read and write latencies. A block size of 8 KB was used in all experiments—in our prototype and in standard NFS implementations when comparisons were made.

4.1. Agile Store Prototype Base Version

The base version of the Agile Store prototype refers to our implementation of a reconfigurable Byzantine quorum system without optimistic quorum collection and adaptive probabilistic hashing. Figure 4 shows the performance of the prototype's base version—measured in terms of latency when reading and writing a 1 MB file—as a function of system size. A minimum Byzantine quorum configuration, which can tolerate one Byzantine fault, requires 5 data servers. Reading or writing a 1 MB file with this minimum configuration of our prototype's base version takes approximately 0.6 seconds. Read/write latency of the base version increases linearly with the number of data servers, albeit at a fairly steep slope. This baseline performance is tolerable (about one third of the speed of NFS), though later we will show that our optimizations improve it considerably.

Figure 5 shows the base version performance as the fault threshold, b, is increased. In this figure, n is set to 21, which is the minimum necessary to tolerate up to 5 faulty or compromised servers. As b increases from 1 to 5, write throughput decreases by over 20%, read throughput decreases by over 25%, and read latency increases by over 40%. Thus, to optimize performance for a given system size, it is important to keep b as low as possible. We note that smaller b also allows the system size to be minimized, further improving performance (compare Figure 4 latencies for $n \leq 10$ to those of Figure 5 with $n = 21$).

One of our mechanisms for maintaining a low fault threshold is the fault detection service, which diagnoses faulty or compromised data servers and removes them from the system by suitably changing the quorum variables. The integration of this process with Byzantine quorum systems is known as reconfigurable Byzantine quorum systems [14]. The fault detection algorithm we described in [14] is used in our prototype. Figure 6 shows the variation of the number of data servers in the system and the fault threshold in one sample run where data servers became faulty over time. The variations in n and b for reconfigurable quorum systems are compared against the case where the fault threshold is preset to a conservatively high value (static Byzantine quorums [18]), and the case where the fault threshold is increased to mask new faults but no faulty servers are removed (dynamic Byzantine quorums [4]).

The experiment was run with 35 data servers, fed with a continuous stream of read operations. The fault threshold b was allowed to vary between $b_{\min} = 1$ and $b_{\max} = 5$

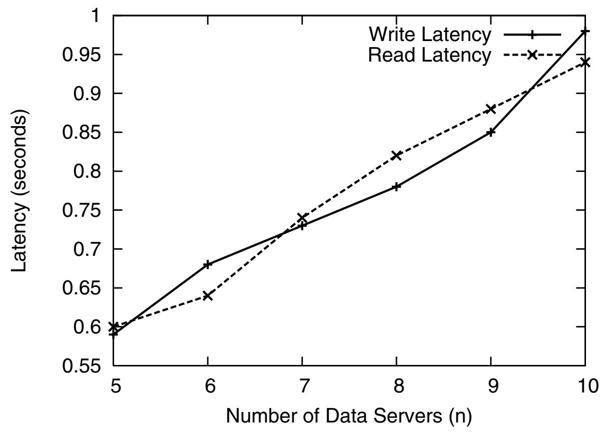

Figure 4. Agile Store base version: latency vs. system size (b = 1).

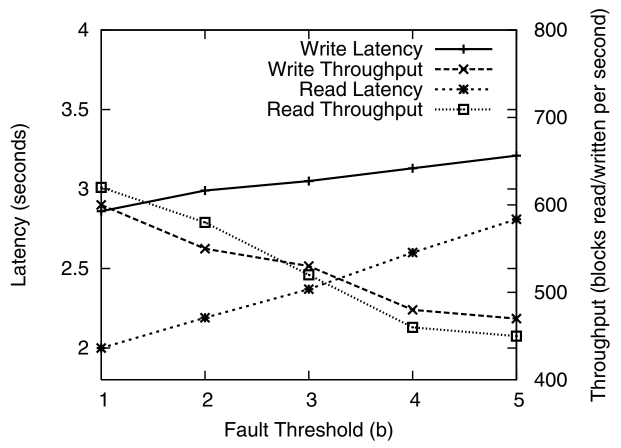

Figure 5. Agile Store base version: latency and throughput vs. fault threshold (n = 21).

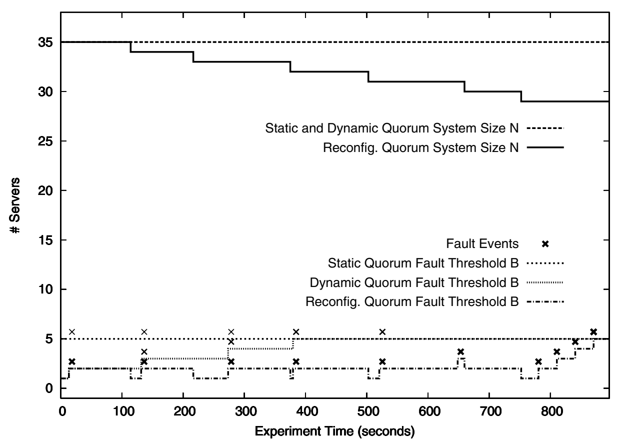

Figure 6. The variation of the number of data servers (n) and the fault threshold (b) when data servers become faulty over time.

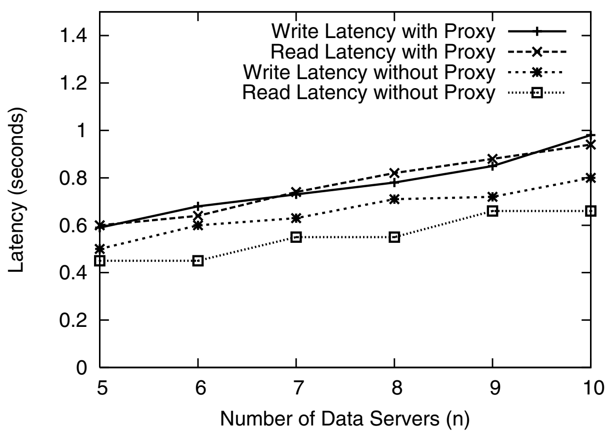

Figure 7. Latency with and without proxies

servers. At start, there were no faulty data servers in the system. Servers became faulty at random times, but the likelihood of additional servers being faulty was increased each time a new server became faulty. A randomly chosen parameter for each faulty server controlled how closely its behavior matched that of a correct server. This controlled how easy or difficult the faulty server was to detect. Since the fault threshold b can increase only up to $b_{\max} = 5$ servers, only five faults can be injected into dynamic and static quorum systems. By contrast, a total of 11 faults were introduced in the case of reconfigurable quorums due to the timely detection and removal of faulty servers, which keeps b in check. All of these faults were eventually detected by our fault detection service despite several of them exhibiting behaviors very close to that of a correct server. Furthermore, the lower value of b maintained for reconfigurable quorums resulted in significantly lower latency and increased throughput as predicted by Figure 5.

4.2. Performance with and without Proxy Servers

In our Agile Store architecture, using proxy servers that act as gateways for client requests allows the proxy servers to monitor other servers' responses and run a fault detection algorithm to detect faulty or compromised servers. Figure 7 shows the times required to read and write a 1 MB file versus system size with and without the use of proxy servers. Using proxy servers increases read latency by about 25% and write latency by about 10%. When the client was operated from a low bandwidth, high RTT link (40 ms) while the data servers remained on a LAN, the overhead of using proxy servers was lowered to approximately 5%.

The use of proxy servers is necessary for our fault detection service to be run. Proxies also facilitate clients that connect over an unreliable network and can aid resource-constrained clients interacting with large system and quorum sizes. Due to these reasons, we have chosen to maintain the use of proxy servers in our prototype. However, if other effective methods of fault detection can be developed, the use of proxies should be reexamined.

4.3. Optimistic Quorum Collection

Figures 8 and 9 show the performance improvement of using the "blind" version of our optimistic quorum collection ("optimistic" in the figure), as compared with single round full read quorum collection ("full" in the figure). Experiments of Figure 8 were performed on static threshold masking quorum systems with different b values, and system sizes equal to $4b + 1$. When there are no faults present, optimistic quorum collection reduces the number of servers accessed during read operations, and both client side latency and system throughput are improved significantly as shown in Figure 8. Though this performance improvement occurs in a fault-free situation, the lifetime of smaller systems is primarily composed of fault-free time and the improvement can therefore be quite significant. Figure 9 shows how the read throughput varies with the number of faults f in the system. With f growing from 0 to the fault threshold, read throughput of the "blind" protocol degrades gracefully, with the worst case on par with single round full read quorum collection.

Figure 10 evaluates the use of optimistic quorum collection in fault situations when $b = 4$ and $n = 17$, for different values of f. $\delta = 0$ and $\delta = 4$ correspond to "blind" optimistic quorum collection and single round full read quorum collection, respectively. We see that there is a penalty incurred for underestimating the number of faults, because two rounds may be necessary to finish a read operation. Figure 10 further shows how different δ values affect the read performance of optimistic quorum collection. As seen in the figure, for a given fault number f, there exists an optimal δ value that can be used to maximize performance. We found that setting $\delta = [\frac{f(q_r-b)}{n-f}]$ (expected fault number in the first round) works well practically for static threshold Byzantine quorums, where q_r is the read quorum size, b is the fault threshold, and n is the system size.

Both optimistic quorum collection and dynamic fault threshold schemes attempt to reduce the overhead of quorum operations, though with differing assumptions. Optimistic quorum collection attempts to estimate the actual number of faults, f, in order to optimize performance, but does not depend on accurate estimation for correct operation. Performance gains resulting from dynamic fault threshold schemes, such as dynamic quorum systems, *require* accurate knowledge of f for correct operation. The performance of these two approaches is compared in Figure 11 for $n = 17$. In order to do this comparison, it was necessary to simulate faulty server behavior, which we did by having faulty servers always return incorrect data for both approaches. Dynamic quorum systems were operated with a minimal fault threshold, b_{min}, set to 1 and a maximum fault threshold, b_{max}, set to 3 [1]. The optimistic quorum collection system used a static fault threshold, b, of 3. For both approaches, we assumed that the protocols are able to determine the actual fault number exactly. For dynamic Byzantine quorums, this set the fault threshold b, while for the optimistic approach, it was used to determine δ.

Optimistic quorum collection outperforms dynamic quorum systems in most cases of the experiment, even when best case scenarios were used for dynamic quorum systems (data objects read were always written under the current threshold for dynamic quorums). Optimistic quorum collection is not as efficient as dynamic quorum systems in write performance due to its use of an un-optimized, large write quorum size. But when overall file system performance is taken with the commonly assumed 80% to 20% read/write ratio, the combined latency of optimistic quorum systems still outperforms dynamic quorum systems.

Optimistic quorum collection is able to achieve its better performance because it sets the δ value lower than the actual number of faults f. This leads to execution of the second round with some probability, but correctness is not affected. As long as that probability is low enough, average read performance will benefit. Dynamic quorum systems achieve better read performance by setting the fault threshold as low as possible, but the threshold is limited by f for correctness.

4.4. Adaptive Probabilistic Hashing

In this section, we experimentally show the presence of a hash probability, p_h, where normalized read latency l reaches its minimum on $[0, 1]$ for different network settings. Here data objects are file blocks of 8 kilobytes. We recall that a high p_h means that most servers will return hashes and the protocol is likely to require a second round to determine the data object value, while a low p_h will cause most servers to return full data objects and the protocol will likely terminate after the (more costly) first round. We present read latency results from two network settings that represent a LAN (Figure 12) and a WAN (Figure 13). The LAN set-

[1]Note that dynamic quorum systems have the same read/write latency when $f = 0$ and $f = 1$. This is because $b = \max(f, b_{\min})$, which is equal to 1 in both of these cases.

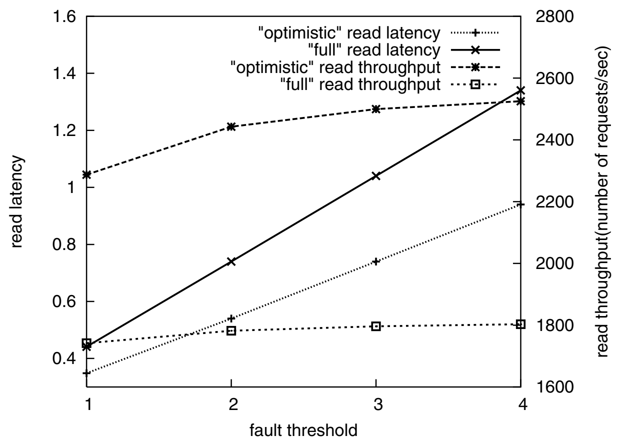

Figure 8. Performance improvement of optimistic quorum collection with no faults

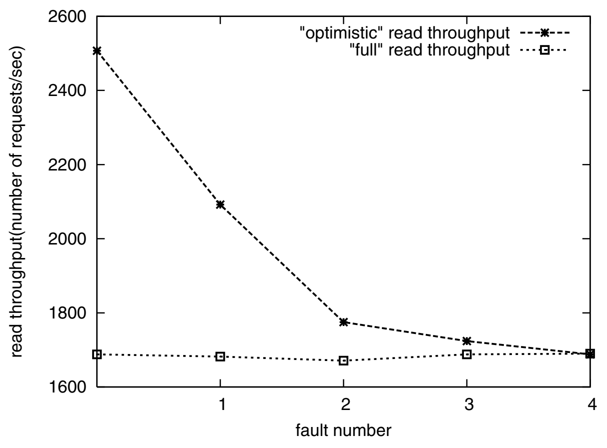

Figure 9. Read throughput with faults (n = 17, b = 4)

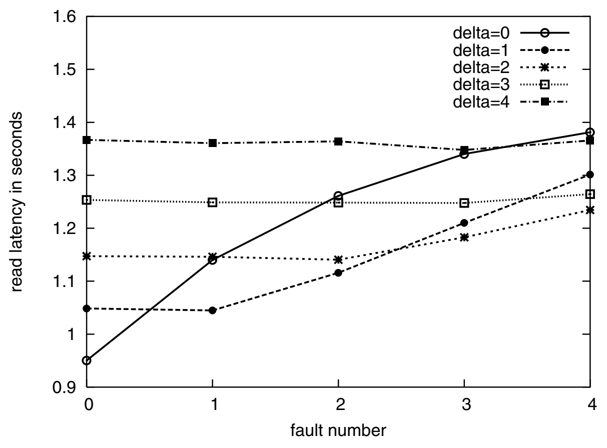

Figure 10. Read latency of δ-optimistic quorum collection

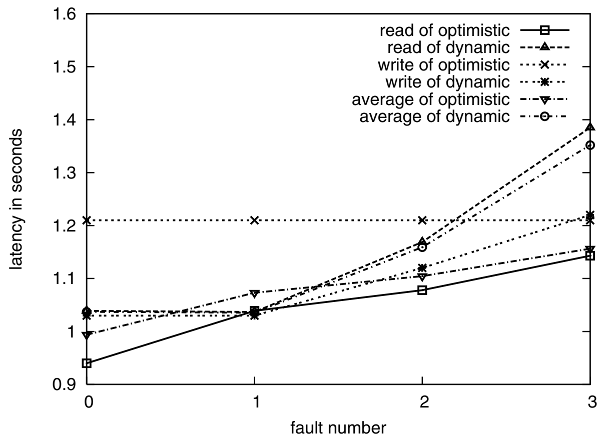

Figure 11. Optimistic quorum collection vs. dynamic Byzantine quorums

ting resulted from Emulab's actual network delays, while the WAN setting utilized Emulab's facility for emulating WAN networks by inserting additional delay components in message paths.

The figures show how hash probability affects read latency in the two network settings with varying bandwidth. In a LAN environment, it is always advantageous to use a high p_h irrespective of bandwidth, because the penalty for a second round is very low. In a WAN environment, a reversal in the slope of normalized read latency occurs between the 100 Mbps and the 10 Mbps cases. At low bandwidths (10 Mbps), a high p_h leads to low latency due to reduced demands on the limited bandwidth. However, at high bandwidths (100 Mbps), the bandwidth savings are insignificant when compared to the penalty (running an extra round) of hashing; a low p_h is thus a better choice. The curve for 95 Mbps further shows that, in some cases, the optimal hash probability takes an intermediate value, neither close to 0 nor close to 1.

These results demonstrate the importance of choosing p_h values that are optimized for current network conditions. We are exploring the use of empirical methods in determining good p_h values for specific network conditions. Since the Agile Store is designed to serve a large number of clients, we believe that empirical data collected from such a large number of clients would be sufficient in providing near-optimal values of p_h for most network conditions. The performance of hashing is dependent on data object size. For the fairly large block sizes typical in file systems, hashing is quite beneficial. However, in other applications with

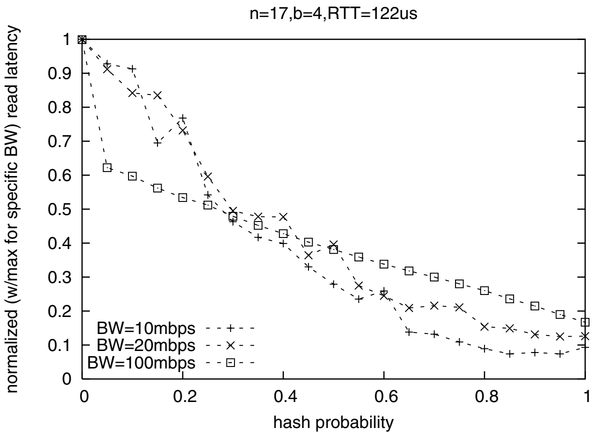

Figure 12. Read latency for LAN setting

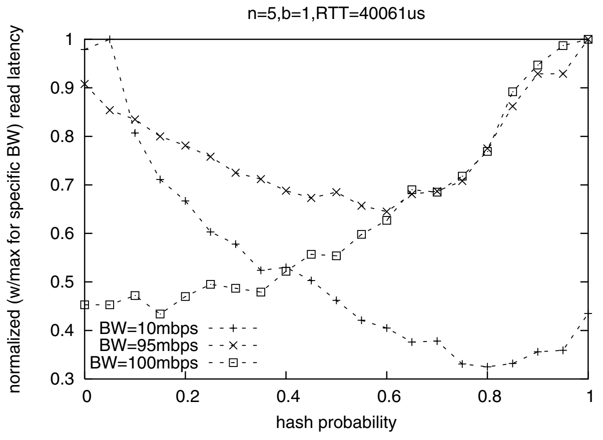

Figure 13. Read latency for WAN setting

smaller data object sizes, the benefits of hashing will likely be significantly reduced.

4.5. Performance of an Optimized Version of the Prototype

Optimal quorum collection and adaptive probabilistic hashing benefit only read operations. While usage studies indicate that reads are far more common than writes, very poor write performance could still have a substantial negative impact on overall performance. Since, in a LAN environment, optimized multicast primitives are usually provided, we considered the use of a multicast-based write operation along with the other two optimizations. In this section, we focus on LAN environments and we, therefore, set $p_h = 1$ based on the results of the previous section.

4.5.1. Microbenchmark Performance

Figure 14 compares the latency in reading and writing 1 MB files with the three optimizations, and compares these values against those of a single NFS v3 server from Redhat Linux 7.3. As seen in the figure, the latency in writing files for our optimized prototype is almost the same as that of NFS with the NFS server running in fast, but less reliable, asynchronous mode. Compared to the base (unoptimized) version of the prototype, write performance is improved by a factor of 3–5. Optimistic quorum collection and probabilistic hashing have significant performance benefits for read operations. From Figure 14, we see that reads in the optimized prototype are about 10–15% slower than NFS over the range of system sizes plotted. The low degradation rate in reads as system size increases is mainly due to the aggressive usage of hashing.

The read and write latencies can be further reduced by having the data servers cache data in volatile memory and

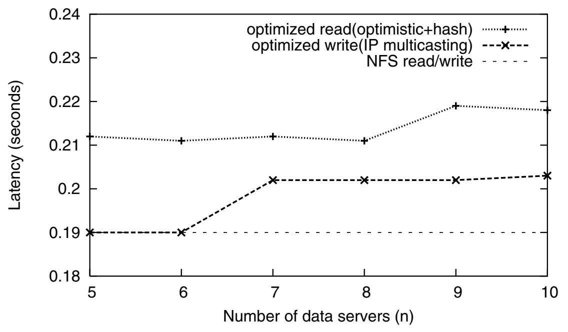

Figure 14. Comparison of the optimized version of our prototype with NFS in terms of read and write latencies on 1 MB files. The fault threshold for the prototype was kept equal to one.

committing it to local storage periodically. This technique has been investigated in [7]. Preliminary experiments with this technique have revealed that the read latency can be improved by an additional 4%. In a LAN setting, the benefit of hashing far exceeds the benefit of caching.

4.5.2. Performance on the Andrew Benchmark

We also compared the performance of our optimized prototype against NFS v3 using the Andrew benchmark [13]. The Andrew benchmark emulates typical workload of software development. It proceeds in five phases, with each phase emphasizing different file system operations. Our prototype was set up to use five data servers and four metadata servers with the fault threshold set to one. Table 1 shows the time taken in seconds for each phase of the Andrew benchmark

Table 1. Andrew benchmark: Execution times of Agile Store and NFS 3 (in seconds)

	Phase I	Phase II	Phase III	Phase IV	Phase V	Total
Agile Store	0.10	1.87	2.66	2.52	15.51	22.66
NFS	0.14	1.43	2.12	2.38	14.95	21.02

for our prototype and NFS. Overall, our prototype is only 7.8% slower than NFS [2]. Phase I consists of many small metadata operations that are handled by the BFT metadata server asynchronously, which accounts for our prototype outperforming NFS in this phase.

The results of this section indicate that a Byzantine fault-tolerant quorum-based file system, consisting of 5 data servers and 4 meta-data servers, can compete with raw NFS v3 in terms of performance, despite the overheads of communicating with multiple servers on each operation.

5. Conclusion

In this paper, we reported on a detailed performance evaluation of both existing and novel techniques for adaptive Byzantine quorum systems, implemented within a prototype file system. Among our most salient findings were:

- techniques for maintaining a low fault threshold, e.g. timely detection and removal of faulty or compromised servers, substantially improve performance;
- use of proxy servers has a moderate performance penalty in LAN environments but it also enables servers to monitor each other's behavior and eases the burden on clients;
- optimistic quorum collection techniques allow performance to degrade gracefully with the actual number of faults and provide similar performance benefits to dynamic Byzantine quorum systems without relying on an accurate fault estimate for correctness;
- probabilistic hashing significantly improves performance for large data objects but the optimal hash probability is dependent on network conditions; and
- single Byzantine fault tolerance can be provided by a quorum system having 5 data servers with less than 10% performance overhead, compared to NFS v3.

Areas for future study include techniques to dynamically optimize hash probability and investigation of Byzantine quorum performance for non-block-oriented data stores, which are used in most WAN-based storage systems.

[2]NFS was running in a fast but less reliable asynchronous mode

References

[1] Ntp v3 rfc 1305. http://www.faqs.org/rfcs/rfc1305.html, Mar. 1992.

[2] The aware home research initiative, college of computing, georgia tech. http://www.cc.gatech.edu/fce/ahri/, 2000.

[3] A. Adya and et al. Farsite: Federated, available, and reliable storage for an imcompletely trusted environment. *Proc.* 5^{th} *OSDI*, Dec. 2002.

[4] L. Alvisi and et al. Dynamic byzantine quorum systems. In *Proc. DSN*, 2000.

[5] L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter. Fault detection for byzantine quorum systems. *IEEE Trans. on Parallel and Distr. Sys.*, 12(9), 2001.

[6] R. J. Anderson. The eternity service. *Proc. of 1st Intl. Conf. on Theory and Appln of Cryptography (Pragocrypt)*, 1996.

[7] M. Castro and B. Liskov. Proactive recovery in a byzantine fault tolerant system. In *Proc. 4th OSDI*, 2000.

[8] Y. Chen and et al. A prototype implementation of archival intermemory. *Proc. of the* 4^{th} *ACM Intl. Conf. on Digital Libraries*, pages 28–37, Aug. 1999.

[9] S. Frolund and et al. A decentralized algorithm for erasure-coded virtual disks. In *Proc. DSN*, 2004.

[10] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th SOSP*, pages 150–162, 1979.

[11] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proc. DSN*, 2004.

[12] M. Herlihy and J. Tygar. How to make replicated data secure. In *Advances in Cryptology*, 1987.

[13] J. H. Howard and et al. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys.*, Feb. 1988.

[14] L. Kong, A. Subbiah, M. Ahamad, and D. Blough. A reconfigurable byzantine quorum approach for the agile store. In *Proc. 22nd SRDS*, pages 219–228, 2003.

[15] S. Lakshmanan, D. Manohar, M. Ahamad, and H. Venkateswaran. Collective endorsement and the dissemination problem in malicious environments. In *Proc. DSN*, 2004.

[16] L. Lamport. On interprocess communication, part 1: Basic formalism. *Distributed Computing*, 1:77–85, 1986.

[17] D. Malkhi and et al. Persistent objects in the fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition*, 2004.

[18] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[19] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. *Proc.* 17^{th} *SRDS*, Oct. 1998.

[20] J. P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *Proc. DSN*, 2004.

[21] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), Dec. 1990.

[22] D. Tulone. Enhancing efficiency of byzantine-tolerant coordination protocols via hash functions. In *Euro-Par*, 2004.

[23] White, Lepreau, Stoller, Ricci, Guruprasad, Newbold, Hibler, Barb, and Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, Dec. 2002.