A Reconfigurable Byzantine Quorum Approach for the Agile Store*

Lei Kong†, Arun Subbiah‡, Mustaque Ahamad†, Douglas M. Blough‡

† College of Computing
‡ School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332 USA
{konglei,mustaq}@cc.gatech.edu,{arun,dblough}@ece.gatech.edu

Abstract

Quorum-based protocols can be used to manage data when it is replicated at multiple server nodes to improve availability and performance. If some server nodes can be compromised by a malicious adversary, Byzantine quorums must be used to ensure correct access to replicated data. This paper introduces reconfigurable Byzantine quorums, which allow various quorum protocol parameters to be adapted based on the behavior of compromised nodes and the performance needs of the system. We present a protocol that generalizes dynamic Byzantine quorums by allowing the system size to change as faulty servers are removed from the system, in addition to adapting the fault threshold. A new architecture and algorithm that provide the capability to detect and remove faulty servers are also described. Finally, simulation results are presented that demonstrate the benefits offered by our approach.

1. Introduction

The Agile Store project at Georgia Tech targets new information rich applications, which will need to access and manipulate sensitive and critical information. Such applications will require that their information is stored securely and made available to only authorized parties when needed. Our research seeks to build agile services that can be used by these applications to store and access information in pervasive computing environments. The motivation for such services comes from the fact that many future applications will span resource-limited embedded processors or mobile or hand-held computers that cannot be trusted to store information securely (e.g. they can be easily compromised, lost or stolen). Thus, the applications will require trusted services that can store their information while addressing both security and performance needs that may change over time.

Since these services will be accessible over public infrastructure, we must allow for the possibility that they will be subjected to attacks. Thus, we are interested in storage services that can guarantee confidentiality, integrity, and availability despite intrusions into and/or compromises of some elements. Quorum systems [5] provide efficient and highly available storage of data across multiple servers. Byzantine quorum systems have been recently introduced [11] as a way of guaranteeing high availability despite a limited number of servers being compromised. Earlier works from the Agile Store project explored approaches to provide various types of consistency within the store and studied how secret sharing techniques can be integrated with replication to provide confidentiality and availability [6, 7].

Assuming the worst case number of failed servers at all times in a Byzantine quorum system can be quite expensive in terms of the required quorum sizes and the associated load they impose on the system. To improve performance, we would like the quorum protocols to be *agile* in the sense of providing highly efficient execution under normal circumstances and a high degree of resilience with perhaps lower performance at times of attack. The approach we take in the Agile Store project is to monitor the current threat level of the system through intrusion and failure detection mechanisms. These mechanisms feed information to the quorum system protocols, which adapt their behavior accordingly, to provide the desired agile operation.

The first important step toward agile quorum systems was the development of dynamic quorum protocols in [1]. These protocols assume some way of estimating the current number of compromised servers and adapt the quorum sizes accordingly. In this paper, we expand on this idea by adding explicit fault detection and reconfiguration capabilities into

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

Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)
1060-9857/03 $17.00 © 2003 IEEE

the system to further optimize performance and availability. We provide a statistical fault detection technique that can detect and identify compromised servers even when they try to escape detection by returning incorrect responses only occasionally. Furthermore, we use the identities of compromised servers to reconfigure the quorum system. This improves system performance by keeping quorum sizes and system load as low as possible, and it also extends system lifetime. Simulations that are reported in Section 5 demonstrate that these benefits are, in fact, substantial.

2. System Model

As shown in Figure 1, our reconfigurable Byzantine quorum system consists of servers, clients, and a special diagnosis node. Each data object is replicated on a subset of servers. The diagnosis node collects fault detection information from servers, makes diagnosis decisions, and updates quorum variables, which are defined later. The total number of servers in the system is not fixed because we allow removal of servers that are diagnosed as faulty.

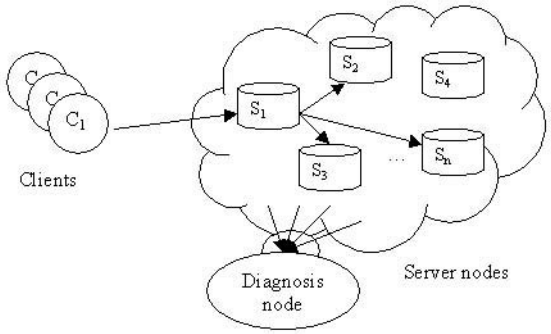

Figure 1. System Architecture

Unlike the traditional method where a client accesses a quorum of servers directly to perform a read or a write, we have for each read or write operation, a server picked randomly by a client to act as a proxy. A client sends its read or write request to the chosen proxy server, and the proxy server forwards the request to the quorum of servers specified by the client. Server responses are returned through the same proxy server to the client. This enables proxy servers to monitor the responses of other servers for failure detection purposes. Both comparison-based detection [9] and explicit testing [3, 13] approaches are employed in our system. Cryptographic methods are used to protect messages from being tampered by faulty proxy servers.

Both clients and servers can experience Byzantine faults, i.e. they can behave arbitrarily. In this paper, we focus on detection and removal of compromised servers and do not consider, in detail, detection of compromised clients. Access control mechanisms are used to limit damage by faulty clients, and clients do not control system operation in our approach, contrary to [1]. Furthermore, correct proxy servers can easily detect and prevent some faulty client behavior such as writing to a partial quorum, which can leave the system in an inconsistent state. We leave dealing with collusion between faulty clients and faulty servers as future work. Faulty clients can have some impact on the accuracy of our fault detection technique. This issue is discussed when describing the fault detection algorithm.

The diagnosis node, which is responsible for diagnosing faulty servers and maintaining the quorum variables, is assumed to be fault-free. The diagnosis node does not, in general, accept connections from the outside world, and communicates with only known nodes (the servers) to receive some fault information. Furthermore, the diagnosis node runs a single, very simple application. These elements should make it feasible to guarantee the security and protection of the diagnosis node. Alternatively, the diagnosis node could be implemented as a Byzantine fault-tolerant state machine [14].

The data service model in our system is based on *masking quorum systems* [11], but it could also work with other quorum systems, such as *dissemination* [11] and *grid* [4] quorum systems. A *masking quorum system*, defined on a *universe* U of servers, requires that $|Q_r \cap Q_w| \geq 2b + 1$, where the read quorum Q_r and the write quorum Q_w are subsets of U, and b is the *resilience threshold* [1] of server faults. Since at least $b+1$ non-faulty servers are in $Q_r \cap Q_w$, the correctness of read operations is guaranteed. In quorum systems, timestamps are used to distinguish different versions of data. Clients choose timestamps from non-overlapping sets when writing data, and each client generates its timestamps in a monotonically increasing order.

It is assumed that the actual number of faulty servers in the system does not exceed the fault threshold b. This assumption is commonly made in work on Byzantine quorum systems, secure group communication, secret sharing, threshold cryptography, and other topics as well. Software diversity can help prevent too many servers from being compromised by the same attack method. For example, different servers could be run on different operating systems, thereby preventing certain classes of attacks from being applied to all servers.

We assume authentication, authorization, and key management services for the system, and digital certificates from a certificate authority bind all parties to their public keys. All parties use cryptographic methods to protect the confidentiality and integrity of their communication.

Communication channels are assumed to be asynchronous but reliable. Generally, clients wait for responses from servers during both read and write operations. Non-

if $(N(new) < N(old))$
  then $X_1 = Q_{min}(old) - (N(old) - N(new))$;
  else $X_1 = Q_{min}(old)$;
$X_2 = \lceil (N(new) + 2B(new) + 1)/2 \rceil$;
$Q_{min}(new) = \min(X_1, X_2)$;

Figure 2. Updating Q_{min}

confirmative write [12] is useful only for special applications. In an asynchronous environment, it is impossible to tell if a server is slow or not responding because it is faulty. One solution is to send a request to all servers; this will succeed when at least one quorum is available, which occurs when when $n \geq 4b + 1$ for *masking quorum systems* [11]. In masking quorum systems, the client can also access servers incrementally until enough respond, or access b or b_{max} more servers to guarantee that a quorum of responses will be received. If a client times out waiting for responses, then the proxy server may be faulty. The client then chooses another proxy server and retries. However, delayed responses received later from the first proxy are still accepted. To simplify the presentation, we omit details on how quorums are collected in our protocols.

3. Reconfigurable Byzantine Quorums

This section details the protocols that are used to maintain quorum variables' values and to read and write data objects in a reconfigurable Byzantine quorum system. Proofs of the results in this section are omitted due to length restrictions.

3.1. Quorum Variables

In order to make the system adaptive, four *quorum variables* are defined in our system. These variables are denoted by N, B, Q_{min} and S. (For clarity, we use n, b, q_{min} and s to denote their values respectively.) A copy of these variables is maintained by every server in the system. N stores the current system size, which can decrease as servers are removed. B records the estimated upper bound on the number of faulty servers in the system. Its value is an integer in the range $[b_{min}, b_{max}]$, where b_{min} and b_{max} are the minimum and maximum possible number of faults in the system respectively. S is a boolean array, with an entry for each server, which indicates if the server is faulty. Upon diagnosing a server as faulty, the diagnosis node marks that server as faulty in S. Clients do not choose servers that have been marked as faulty to be in their read/write quorums, which logically eliminates faulty servers from the system.

The dynamic Byzantine quorum protocols of [1] will not work without modification if the system size n decreases. Quorum protocols require the intersection between a read quorum and a write quorum to be of some minimum size. When computing the read quorum size in [1], the write quorum size of an object is assumed to be $\lceil (n+2b_{min}+1)/2 \rceil$. However, faulty servers that are removed might have participated in some write operations and $\lceil (n + 2b_{min} + 1)/2 \rceil$ might not decrease as much as n does. Thus, it is necessary to keep track of the effective minimum write quorum size. The quorum parameter Q_{min} is defined for this purpose. Suppose $Q(V)$ stands for the number of servers that hold the current value of data object V. $Q(V)$ can be different from the most recent write quorum size of the data value, because removal of faulty servers can decrease $Q(V)$. We use Q_{min} to store the minimum value of $Q(V)$ over all data objects stored in the system. This enables us to compute the proper read quorum size when both N and B are variables, and it also leads to a smaller read quorum than in [1] for the same system size and $Q_{min} \geq \lceil (n + 2b_{min} + 1)/2 \rceil$.

Initially when the system starts, $N =$ initial system size, $B =$ some value in the range $[b_{min}, b_{max}]$, and $Q_{min} = \lceil (N + 2B + 1)/2 \rceil$. Every time N or B is updated, the system tries to update Q_{min} using the algorithm in Figure 2. Here, $Y(new)$ and $Y(old)$ denote the new and old versions of variable Y respectively. The decrease of Q_{min} could be caused by two reasons. First, faulty servers that have been removed could belong to some write quorums, so Q_{min} should decrease as much as N does. Second, Q_{min} should be no greater than the new write quorum size. X_1 and X_2 in the algorithm denote the new value of Q_{min} computed according to these two factors, and their minimum is assigned as the new value of Q_{min}.

3.2. Read/Write Protocols for Quorum Variables

Quorum variables are read and written according to the protocols shown in Figure 3. The four quorum variables are read and written together. We use P to denote the four-tuple of the variables and use t_P to denote their timestamp. A pair $< p, t_P >$ is *countermanded* if at least $b_{max} + 1$ servers return pairs with timestamp higher than t_P.

Since the diagnosis node is the sole writer for the quorum variables, the diagnosis node will always be able to update t_P with a value greater than any other value used so far without having to perform an explicit read on t_P to obtain t_P's current value. Also, there cannot be concurrent writes on quorum variables. Theorem 1 states that the quorum variables' protocols guarantee a property that is stronger than safe variable semantics [8].

Theorem 1 *A quorum variable read that overlaps with no quorum variable write returns the most recently written value. A quorum variable read never returns values older than the values written by the most recently completed write.*

Read Protocol
1) Randomly choose a quorum Q_r such that $|Q_r| = 3b_{max} + 1$;
2) Read $< p, t_P >$ from servers in Q_r;
3) $F_1 = \{< p, t_P > | < p, t_P >$ is returned
by at least $b_{max} + 1$ servers $\}$;
4) $F_2 = \{< p, t_P > | (< p, t_P > \in F_1)$
$\wedge (< p, t_P >$ is not *countermanded*) $\}$;
5) if ($|F_2| == 1$) then return the pair in F_2;
6) else return error;
endif;

Write Protocol
1) Randomly choose a quorum Q_w such that $|Q_w| = n - b_{max}$;
2) Write $< p, t_P >$ to servers in Q_w;

Figure 3. Quorum Variable Protocols

3.3. Read/Write Protocols for Data Objects

Read and write operations proceed in two phases. First quorum variables are read, from which the quorum size is computed and a quorum is chosen, and then data objects are read from or written to this quorum. Figure 4 shows the read/write protocols, in which $< v, t >$ stands for the value and timestamp pair of a requested data object, and $< p, t_P >$ stands for the pair of quorum variables. In the write protocol, v_{new} denotes the new value to be written and $t_{V_{new}}$ denotes its new timestamp.

In read/write operations, servers send back their quorum variable values $< p, t_P >$ as well as the requested data object value. Thus, the client can check if it is using the most up-to-date quorum variable values. After step 7 in a read operation, the client recomputes quorum variable values based on $< p, t_P >$ pairs it received in step 7. If quorum variables have different values from what were computed in step 1, then the client updates its quorum variable values, and restarts the read operation from step 2. To ensure clients are notified of changes to quorum variables when they read data objects, the intersection of any data object read quorum and any quorum variable write quorum needs to contain at least $2b_{max} + 1$ servers, which gives N a lower bound: $N \geq 6b_{max} - 2b_{min} + 1$, as in [1].

Theorem 2 states that the data object protocols guarantee safe variable semantics [8].

Theorem 2 *A read of V that overlaps with no write on V returns $< v_{cur}, t_{V_{cur}} >$, which is the value of the most recently completed write to V.*

3.4. Other Protocol Issues

Since all client requests and server responses go through proxies, we need to prevent faulty proxies from tampering with them. MACs (message authentication codes) are used to protect message integrity. Each client establishes a symmetric key with each server using their asymmetric keys periodically. Client write requests and server responses are protected by MAC based on symmetric keys shared by the client and each server node in the chosen quorum. In order to make read requests anonymous to servers for diagnosis purpose, proxy nodes don't forward MACs of read requests to servers, then servers have no way to check if read requests are generated by clients or they are test requests from other servers. This makes it possible for servers to generate extra test read requests to speed up fault detection when necessary.

In the data read/write protocols, quorum variables are read in every operation. However, since quorum variables do not change frequently, it is better to use cached values when generating requests. Since servers send back their current quorum variable values in their responses to data object

Read Data Object V
1) Execute a read on quorum variables;
2) Randomly choose a server as the proxy;
3) Randomly choose a quorum Q_r s.t. $|Q_r| = n + 2b + 1 - q_{min}$;
4) Randomly choose Q_{r_1}
s.t. $(Q_{r_1} \subset Q_r) \wedge (|Q_{r_1}| = n + b + b_{min} + 1 - q_{min})$;
5) Send the read request and Q_{r_1} to the proxy;
6) The proxy reads $< v, t >$ and $< p, t_P >$ from each server in Q_{r_1};
7) The proxy forwards $< v, t >$ and $< p, t_P >$ pairs to the client;
8) $R_1 = \{< v, t > | < v, t >$ returned by $\geq b_{min} + 1$ servers $\}$;
9) if ($R_1 \neq \phi$) then
10) $R_2 = \{< v, t > | < v, t >$ returned by $\geq b + 1$ servers $\}$;
11) if ($R_2 = \phi$) then
12) Notify the proxy to read from $Q_r - Q_{r_1}$;
13) The proxy reads $< v, t >$ and $< p, t_P >$ from
$Q_r - Q_{r_1}$ and forwards them to the client;
14) $R_2 = \{< v, t > | < v, t >$ returned by $\geq b + 1$ servers $\}$;
15) if($R_2 \neq \phi$) then
16) return $< v, t >$ in R_2 with the newest time stamp;
17) else return error;
18)else return error;

Write Data Object V
1) Execute a read on V to get the current t_V and $< p, t_P >$;
2) if (read quorum size increases according to step 1) then
repeat
3) Read t and $< p, t_P >$ from every server node;
4) until at least $n - b_{max}$ servers return the same $< p, t_P >$;
5) Generate $t_{V_{new}}$ higher than any timestamp from Step 1 or Step 3
6) Randomly choose a server as the proxy;
7) Randomly choose a quorum Q_w s.t. $|Q_w| = \lceil (n + 2b + 1)/2 \rceil$;
8) Send $< v_{new}, t_{V_{new}} >$ and Q_w to the proxy;
9) The proxy writes $< v_{new}, t_{V_{new}} >$ to servers in Q_w;
10)The proxy forwards write confirmations to the client;

Figure 4. Data Object Protocols

read/write requests, clients can detect quorum variable updates and regenerate requests issued with old values.

4. Fault Detection

A new fault detection algorithm that identifies faulty servers thus enabling their removal is described in this section. Proxy servers monitor server responses during read operations and, over time, the proxy servers are able to determine if other servers are faulty with a specified false-alarm probability. Proxy servers communicate their findings to the diagnosis node, which is responsible for updating the quorum variables N, B, $Q_{\min}$, and S. Since faulty proxy servers may report non-faulty servers as faulty, the diagnosis node employs a voting mechanism to decide if a server is indeed faulty. Hence, the fault detection algorithm uses a two-tiered approach: the proxy-node algorithm and the diagnosis-node algorithm.

4.1. Fault Detection Algorithm at the Proxy Node

To better illustrate the algorithm, we make some simplifying assumptions. The fault detection algorithm is run periodically and, in between two executions of the algorithm, the following assumptions hold:

1. There are no changes in the quorum parameters.
2. No server becomes faulty.
3. There are no concurrent reads and writes.
4. A quorum of size q is chosen randomly from all possible quorums of size q.

Assumptions 1 - 3 will be relaxed in Section 4.3. Assumption 4 will be commented on in the context in which it appears in this section.

During a data read operation, the proxy server is aware of the responses returned by the servers in the read quorum. Hence, it can determine the result of the read as would be chosen by the client, which will be termed here as the "correct response." In a read operation, some servers will be unable to give the correct response because they were not part of the write quorum of the last completed write to the data object. Hence, to be able to distinguish faulty servers from non-faulty servers using this approach, statistical analysis is used over a sequence of read operations.

When a non-faulty server is part of a read operation on a data object, the probability that it will return a correct response is the probability that it belonged to the write quorum of the last completed write on the same data object, which is $|Q_w|/n$. If read operations to r objects are monitored, then the probability that a fault-free server returns u correct responses is given by

$$\binom{r}{u}\left(\frac{|Q_w|}{n}\right)^u\left(1-\frac{|Q_w|}{n}\right)^{r-u} \quad (1)$$

Hypothesis testing is used to determine if a server is faulty. The null hypothesis, H_0, is defined as a server being non-faulty, and for a fixed false-alarm probability the null hypothesis H_0 is either rejected or accepted. The false-alarm probability is fixed at 0.05, meaning the probability that a non-faulty server is found faulty by a fault-free proxy server is 0.05. From the false alarm probability, a threshold value for u, denoted by $u_{\rm th}$, is determined, and if a server returns $u_{\rm th}$ or fewer correct responses in r read operations, then the null hypothesis H_0 is rejected and the server is said to be faulty. $u_{\rm th}$ is the maximum value of u such that the following inequality is satisfied:

$$\sum_{i=0}^{u}\binom{r}{i}\left(\frac{|Q_w|}{n}\right)^i\left(1-\frac{|Q_w|}{n}\right)^{r-i} \leq 0.05 \quad (2)$$

Since Byzantine faults are considered, faulty servers could try to avoid detection by sporadically giving incorrect responses. Since the number of faulty servers is never greater than b, incorrect responses by faulty servers will not affect the result of a read operation when the read is not concurrent with a write to the same data object. Note that faulty servers do, however, have the incentive to respond incorrectly as soon as they are compromised because, under Byzantine quorum system operation, they can force old values to be read when a read operation is concurrent with a write to the same data object.

Let $p_{\rm ic}$ denote the probability with which a faulty server returns an incorrect response when it has the correct value for the data object being read. $p_{\rm ic}$ is a probability representative of the faulty server's behavior over the sequence of r read operations being monitored and cannot be estimated or observed. An ideal fault detection algorithm would detect a faulty server for all $p_{\rm ic}$, $0 < p_{\rm ic} \leq 1$.

The probability that a faulty server returns a correct response during a read operation is $\frac{|Q_w|}{n}(1-p_{\rm ic})$ and the probability that an incorrect response is returned is $1-\frac{|Q_w|}{n}+\frac{|Q_w|}{n}p_{\rm ic}$. Since a server is said to be faulty if it returns $u_{\rm th}$ or fewer correct responses in r read operations, the probability that a faulty server will be detected as faulty in r read operations is

$$\sum_{i=0}^{u_{\rm th}}\binom{r}{i}\left(\frac{|Q_w|}{n}(1-p_{\rm ic})\right)^i\left(1-\frac{|Q_w|}{n}+\frac{|Q_w|}{n}p_{\rm ic}\right)^{r-i} \quad (3)$$

For a store consisting of $n = 70$ servers and with a write quorum size $|Q_w| = 42$ servers, Figure 5 shows the probability that a faulty server is detected for various values of

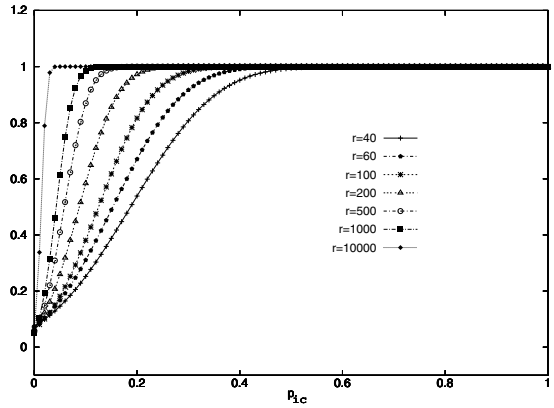

Figure 5. Probability that a faulty server is detected vs p_{ic} for several r

p_{ic} and r with a false alarm probability of 0.05. Even for r as low as 100 read operations, the probability that a faulty server is detected is essentially one for a wide range of p_{ic}. The probability that a faulty server is detected improves as the number of monitored read operations increases.

It is interesting to note that a faulty server has to behave at a very small p_{ic} to avoid detection. Thus, the algorithm forces faulty servers to behave almost as though they were correct in order to avoid detection.

The above analysis is valid as long as Assumption 4 holds when applied to write quorums. If the quorum selection strategy is defined such that all the possible quorums of size q are not equally likely to be chosen during write operations, then the fault detection algorithm can be suitably modified. Assumption 4 could be *violated* when faulty clients do not adhere to the chosen strategy. For example, faulty clients can deliberately exclude a particular non-faulty server in their write operations. Although we do not consider faulty clients in this paper, we note that it is possible to have proxies monitor clients' quorum selections to detect this type of behavior.

Proxy servers run the above described algorithm periodically for each server in the store. At the end of one such period for a monitored server, the proxy server determines if the monitored server is faulty with the specified false alarm probability and informs the diagnosis node of the result.

4.2. Fault Detection Algorithm at the Diagnosis Node

For each server in the store, the diagnosis node maintains a list of servers that found this server to be faulty. This list may increase or decrease with time. If at any point, a certain minimum number of servers, denoted by m, claim that some server is faulty, then that server is diagnosed as faulty and is removed from the system. The quorum parameters (N, B, $Q_{\min}$, and S) are updated accordingly by the diagnosis node. The value of m can be chosen to achieve a false alarm probability lower than the false alarm probability used by the proxy fault detection algorithm. The choice of m is also influenced by the number of concurrent reads and writes in the system, which are a potential source for false alarms.

For example, if the final desired false alarm probability is 10^{-4}, then let m'' be the smallest m' such that the following inequality is satisfied:

$$\left(\begin{array}{c} n - b_{\max} \\ m' \end{array} \right) (0.05)^{m'} (1 - 0.05)^{n - b_{\max} - m'} \leq 10^{-4} \tag{4}$$

where 0.05 is the false alarm probability used in the proxy-node fault detection algorithm. $n - b_{\max}$ and not n is used in the above inequality because, in the worst case, $b_{\max}$ faulty servers could try to vote a non-faulty server out of the system. Hence, m given by $m'' + b_{\max}$ guarantees a final false alarm probability of at most 10^{-4}.

A faulty server could defeat the above algorithm by returning incorrect responses only when a particular set of fewer than m servers are proxy servers and behaving correctly when other servers are proxy servers. Since the diagnosis node will then not be able to gather sufficient (m) votes to identify the faulty server, the faulty server will go undetected. Hence, we make the following assumption.

Assumption 5: The behavior of faulty servers is independent of the identity of the proxy servers.

The above assumption can be relaxed by having proxy servers drop servers they have found to be faulty from quorums specified by a client. The client will then not be able to get a quorum of responses and will generate requests to other servers, ensuring that the read or write completes. Another technique is to have a proxy server tunnel the client's requests through other proxy servers before the request reaches a quorum of servers. Thus, faulty servers will not be able to single out specific proxy servers. Evaluating these ideas more thoroughly is an area for future work.

4.3. Relaxing Assumptions 1 - 3

Assumption 1 states that $\frac{|Q_w|}{n}$ is assumed to be constant. Reconfigurable quorums will have different $\frac{|Q_w|}{n}$ ratios associated with different data objects. This is handled by clients writing the quorum variables' timestamp along with the data object during a write. To overcome this storage overhead, when a server receives a write request for a data object using a particular quorum timestamp as seen by the client, the data object is stored in a directory identifying the timestamp of the quorum variables with which the data object is written. All previous instances of the data object present in other directories are deleted.

At the end of r read operations that a server participated in and were monitored by a proxy server, let the number of data objects read that were written with quorum parameters $|Q_w|_1$ and n_1 be r_1, the number of data objects read that used quorum parameters $|Q_w|_2$ and n_2 be r_2, etc. If servers in the read quorum return the timestamp of the quorum variables associated with the data object to the proxy server during a read operation, then the proxy server can easily estimate $r_1, r_2, \ldots$ and the $|Q_w|/n$ ratios associated with the different quorum timestamps. The correct response in a read operation has at least $b + 1$ servers returning the same value and timestamp of the data object and timestamp of the quorum variables that were used at the time the data object was last written. This modification to the read protocol will not affect its correctness because when a data object is written, the client specifies the quorum timestamp that it is using during the write and the data timestamp and value and the quorum timestamp are protected from tampering. Taking into account varying write quorum sizes, the probability that a faulty server is detected is

$$\sum_{i=0}^{u_{\rm th}} \sum_{\substack{x'_{\rm j}s, \\ 0 \le x_{\rm j} \le r_{\rm j}, \\ \sum x_{\rm j} = i}} \binom{r_{\rm j}}{x_{\rm j}} \left(\frac{|Q_{\rm w}|_{\rm j}}{n_{\rm j}}(1 - p_{\rm ic})\right)^{x_{\rm j}} \cdot \left(1 - \frac{|Q_{\rm w}|_{\rm j}}{n_{\rm j}} + \frac{|Q_{\rm w}|_{\rm j}}{n_{\rm j}} p_{\rm ic}\right)^{r_{\rm j} - x_{\rm j}}$$

With $p_{\rm ic}$ equal to zero, the minimum value of $u_{\rm th}$ such that the above expression is not greater than the false alarm (0.05) gives the threshold value of u in r read operations.

Assumption 2 states that no servers become faulty during the execution of the fault detection algorithm. If a server becomes faulty during the r read operations that were monitored, the average $p_{\rm ic}$ over the r read operations will be lower than the actual $p_{\rm ic}$ that is representative of the faulty server's behavior. Hence, if a server became faulty recently, the effective $p_{\rm ic}$ over the r read operations monitored could be small enough so that it goes undetected. Then, the faulty server will be detected in the next round of r read operations provided it continues to perform with a $p_{\rm ic}$ in the detectable range. Small values of r are particularly useful in detecting servers that became faulty recently. Simulation results in Section 5 show that even with r as low as 100 read operations, faulty servers with small $p_{\rm ic}$ are detected.

If Assumption 3 is relaxed, then concurrent reads and writes can occur. It could then be possible that a wrong "correct" response is determined during a read operation, thereby counting fault-free servers as faulty. This would increase the actual false alarm probability to be higher than the target value used in Equation 4. This can be dealt with simply by lowering the target false alarm probability to counter the impact of concurrency.

4.4. Comparison with Related Work

The primary work in fault detection for Byzantine quorum systems is [2]. The Write Marker Protocol of [2] can identify faulty servers in a single read operation provided it is guaranteed that no read will be concurrent with a write. The Write Marker Protocol incurs an overhead of n bits per data object, where n is the total number of servers in the system. This overhead can be quite significant if a large number of data objects are stored.

The overhead incurred in the proposed fault detection algorithm described in this paper is the tracking of read responses of every server in the store by the proxy servers. This storage overhead does not depend on the number of data objects in the system. For each server in the store, proxy servers maintain a chronological list of read operations in which the server participated. Each entry in the list contains the quorum timestamp associated with the data object that was the result of the corresponding read operation, and a Boolean value indicating if the server returned the correct response. Since quorum parameters are assumed not to change often, two bytes should be sufficient to hold each entry in the list. Assuming a maximum of 1000 read operations are monitored for each server, a storage space of 2000 bytes will be required. For a store of size 70 servers, each proxy will need approximately 140 KB of main memory space, which is independent of the number of data objects in the system and can be easily accommodated with typical memory configurations in today's server class machines.

Another benefit of our approach is that it can tolerate a limited amount of concurrency between read and write operations, while this is not true for the Write Marker Protocol. In Section 5, simulation results demonstrating this capability are presented.

5. Simulation Results

5.1. Comparison between Reconfigurable, Dynamic, and Static Quorum Systems

The read and write protocols and fault detection algorithm of our approach were simulated and compared to dynamic and static Byzantine quorum systems. A system size of 70 servers was assumed, with the fault threshold b ranging within $[1, 8]$. Since the estimation of b is beyond the scope of this paper, it is assumed, for both reconfigurable and dynamic quorums, that b is updated as soon as a server becomes faulty or when a faulty server is removed from the system. In the simulations, b is maintained as the actual number of faulty servers in the system plus one to model

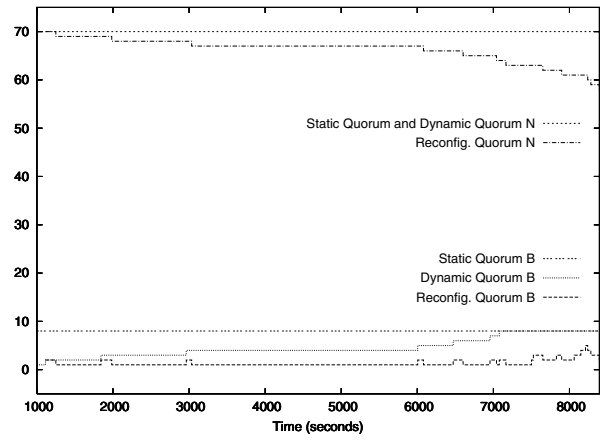

Figure 6. Comparison of n and b between Reconfigurable, Dynamic and Static Quorums

a safety margin that should be built into the b estimation. Since n is constant and b is always equal to $b_{\max}$ for static quorums, the read and write quorum sizes and workload remain constant in that case.

The time at which a server becomes faulty is computed for each server using an exponential distribution with mean $MTBF\sqrt{\frac{B_{\min}+B_{\max}}{2(\#faulty\ servers)}}$, where $MTBF$ is the Mean Time Between Failures for a server and $\#faulty\ servers$ is the total number of servers that became faulty since simulation start. Thus, the effective $MTBF$ decreases as the number of faulty servers increase. This is intended to model scenarios such as virus propagation and progressive attacks. The value of $MTBF$ used in the simulations is 2 days. At system start, there are no faults in the system and thus, b is equal to 1 initially. When a server becomes faulty, the p_{ic} value that will govern its subsequent behavior is chosen uniformly over $(0, 1]$. When the number of faulty servers reaches $b_{\max} = 8$ in the case of dynamic and static quorum systems, no more fault events are introduced.

The time taken for a message to traverse a communication link is a minimum message travel time (t_1) + a random amount of time exponentially distributed with mean t_2. In the simulations, t_1 is 1 second and t_2 is 4 seconds. Read and write requests for each data object are modeled as Poisson processes. For each object, the mean interarrival time for reads is 80 seconds and the mean for writes is 150 seconds, thus producing an average read/write ratio of $15 : 8$.

Figure 6 compares the system size and the fault threshold b between reconfigurable, dynamic and static quorum systems during the length of one simulation run. Since the fault detection algorithm is incorporated into the simulation of reconfigurable quorum systems, faulty servers are removed some time after the fault event thus maintaining the fault threshold b fairly constant. The false alarm probability is set to 0.05 in the proxy-node fault detection algorithm and 10^{-20} in the diagnosis-node fault detection algorithm. For a system size of 70 servers and with $b_{\max} = 8$, at least 38 servers must notify the presence of the faulty server to the diagnosis node before the faulty server is removed from the system. Proxy servers run the fault detection algorithm on other servers every $r = 100$ read operations. There were no incorrect diagnoses even though the measured concurrency between read and write operations was 32.63%.

In the depicted simulation, our fault detection algorithm identified all but one failure and the average latency in detecting faults was found to be 164 seconds. The fault event that went undetected happened at time 8062.04, and the reason it was not detected is because the p_{ic} value governing its faulty behavior was a very small 0.0125. On the other hand, a faulty server with a p_{ic} as low as 0.0775 was detected, albeit with a high latency of 714 seconds. In general, it was observed that faulty servers that behave with a small p_{ic} require a long time to be detected. From Figure 5, we see that, with $r = 100$, it is remarkable that a faulty server with a p_{ic} equal to 0.0775 was detected, and a faulty server with $p_{\text{ic}} = 0.0125$ going undetected is not unexpected. The smallest latency in diagnosing a faulty server was found to be 68.99 seconds, corresponding to a p_{ic} value of 0.807.

The system sizes for dynamic and static quorum systems remain constant at 70 because faulty servers are not removed. The fault threshold b for the dynamic quorum system increases with every fault event until $b_{\max} = 8$, beyond which no more fault events are introduced. Simulation for reconfigurable quorum systems is shown till $Q_{\min}$ reaches $3b_{\max} + 1$, beyond which no more servers can be removed.

Once $Q_{\min}$ reaches $3b_{\max} + 1$, every additional fault event increases b by 1 until b reaches $b_{\max}$. The reconfigurable quorum system will stop functioning properly once a fault event occurs after b reaches $b_{\max}$. For the simulation shown, the system lifetime for reconfigurable quorum systems was found to be 9885.12 seconds. In the case of dynamic and static quorum systems, when a fault event happens at time $t = 7501.81$ seconds (see Figure 6), b will exceed $b_{\max}$ which is not allowed. Thus, in this simulation, the system lifetimes for dynamic and static quorum systems were found to be 7501.81 seconds. Note that the MTBF value used in the simulation is quite low so as to generate a large number of failures in a short period. Therefore, the system lifetime values found in the simulation are much lower than they would be in practice. However, the relative lifetimes should be consistent with reality and they demonstrate that reconfigurable quorums can significantly extend a system's life as compared to dynamic and static quorums.

Figures 7 and 8 compare the read and write quorum sizes. The quorum protocol for dynamic quorums [1] specifies that a read quorum of size $\left\lfloor \frac{n+2b+1}{2} \right\rfloor$ is tried first for a read operation, and if there is a need to query more servers, an ad-

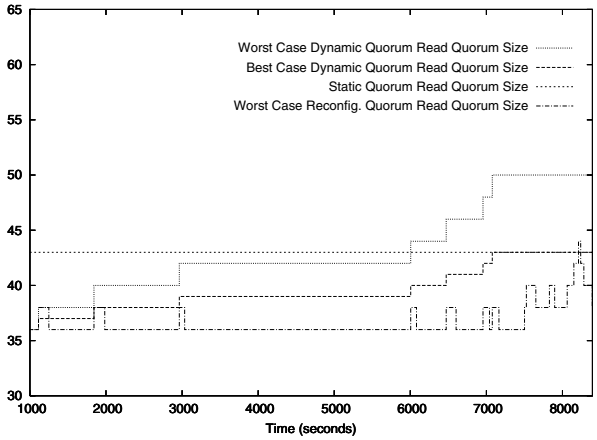

Figure 7. Comparison of Read Quorum Sizes between Reconfigurable, Dynamic and Static Quorums

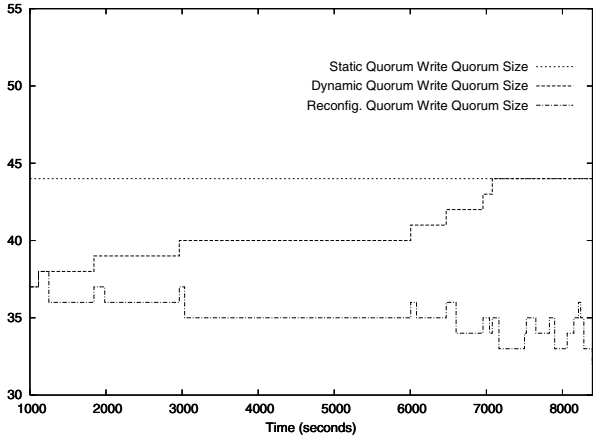

Figure 8. Comparison of Write Quorum Sizes between Reconfigurable, Dynamic and Static Quorums

ditional $b - b_{\min}$ servers are queried. These two cases are identified as the best and worst cases in the plots for the read quorum size and the workload for dynamic quorums. Reconfigurable quorum systems also use a two step read, where $n + b + b_{\min} + 1 - q_{\min}$ servers are queried first, and, if required, an additional $b - b_{\min}$ servers are queried. To simplify the simulations, we always used a read quorum size of $n + 2b + 1 - q_{\min}$ servers. A single curve referring to this case as the worst case for reconfigurable quorums is shown for the read quorum size and the workload plots.

In the case of dynamic quorums, the read and write quorum sizes increase when b increases. For reconfigurable quorums, the read quorum size increases when a fault event happens, but decreases when a faulty server is removed due to the simultaneous decrease in n and b. For the same reason, the write quorum size follows a similar pattern.

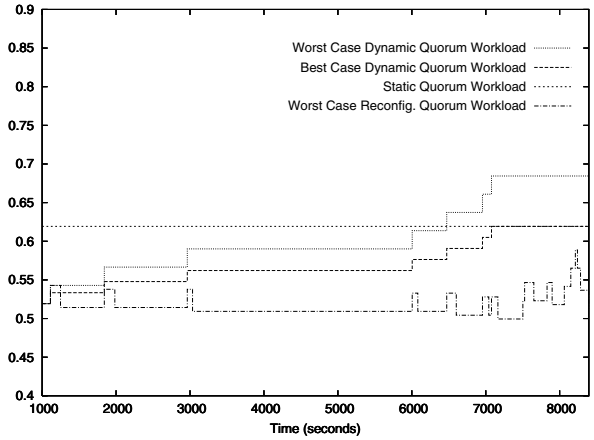

Figure 9. Comparison of Workload between Reconfigurable, Dynamic and Static Quorums

Figure 9 compares the workload between reconfigurable, dynamic, and static quorum systems. The workload is defined as the average fraction of servers involved in a read or write operation. This is equivalent to $\frac{r|Q_r|+(1-r)|Q_w|}{n}$, where $|Q_r|$ is the read quorum size, $|Q_w|$ is the write quorum size, n is the current system size, and r is the fraction of operations that are reads. Figure 8 shows that the smaller read and write quorum sizes for the reconfigurable quorum system as compared to the dynamic quorum system result in a lower workload for reconfigurable quorums despite the fact that n is smaller also. After the dynamic quorum system reaches its fault limit, the reconfigurable quorum system has a worst case workload approximately 20% lower than the best case dynamic quorum workload.

5.2. Concurrency Analysis

Figure 10 shows the performance of our fault detection algorithm in the presence of concurrent reads and writes for the same system parameters used in Section 5.1. In Section 5.1, the read and write requests were generated by separate Poisson processes. To do concurrency analysis, finer control over concurrency is needed. We achieved this by having, for each data object, a write request generated periodically at the same time as read requests. For example, if 50% concurrency is desired, then a write request is issued with every other read request. In each concurrency interval shown in the figure, 20 simulations were run and the number of simulations that had any incorrect diagnoses were noted. This is shown as a fraction of the total number of simulations run for that particular concurrency interval.

Generation of write requests at exactly the same time as read requests produces maximum overlap between them. It can, therefore, be considered as the worst-case concur-

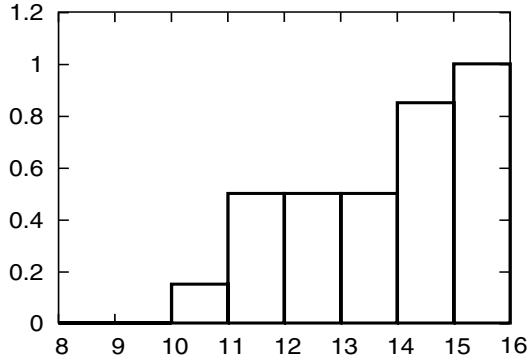

Figure 10. Fraction of Simulation Runs with Incorrect Diagnosis vs. Percentage of Reads Concurrent with Writes

rency scenario for a particular concurrency rate. From Figure 10, we find that no incorrect diagnoses were produced in 20 simulation runs at a concurrency rate in the range $9\% - 10\%$, while the number of runs with incorrect diagnoses rose rapidly as the concurrency rate increased beyond this point. In Section 5.1, there were no incorrect diagnoses reported despite a concurrency rate of more than 32%. In those simulations, however, read and write operations began at random times and concurrency happened naturally as a result. When read and write operations overlap only partially, incorrect diagnosis becomes less likely than in the maximum overlap case. If the approximate concurrency rate in the system is known, we could factor this into the diagnosis threshold calculation and eliminate incorrect diagnosis even for higher concurrency rates. Verification and quantification of this idea is left for future work.

6. Ongoing Research

Increasing Q_{min} will result in a lower read quorum size, resulting in better performance and lower server work load. As defined in Section 3.1, Q_{min} stores the minimum value of $Q(V)$ for all data objects in the system. Between two successive writes to a data object V, $Q(V)$ could be increased if background dissemination is done. Quantitative analysis of dissemination done in [10] suggests that it is reasonable and feasible to increase Q_{min} when it is less than $\min(Q(V))$ over all data objects in the system. Algorithms to update Q_{min} based on background dissemination are a topic of current research. Since servers diagnosed as faulty are removed, fault-free servers must be added to keep the system running indefinitely. Dynamically adding servers to the store is also a topic of current research.

Additional work underway involves implementation of a prototype file system based on the agile store concepts discussed herein. On the client side, a local NFS loop back server is employed to provide transparent access to our agile storage service for applications. The local NFS server accepts requests from applications forwarded by the operating system and executes our reconfigurable Byzantine quorum protocols. File blocks are replicated on servers, and the server side software implements the functionality of both servers and proxies in our system architecture. A logical meta data server is implemented using the state machine approach [14] to provide authentication, authorization and access control service.

References

[1] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. N. Wright. Dynamic byzantine quorum systems. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 283–292, 2000.

[2] L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for byzantine quorum systems. *IEEE Trans. on Parallel and Distributed Systems*, 12(9):996–1007, Sep. 2001.

[3] F. Barsi, F. Grandoni, and P. Maestrini. A theory of diagnosability of digital systems. *IEEE Trans. Computers*, pages 585–593, June 1976.

[4] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *Knowledge and Data Engineering*, 4(6):582–592, 1992.

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

[6] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. A secure and highly available distributed store for meeting diverse data storage needs. In *Proc. Intl Conf. on Dependable Systems and Networks*, pages 251–260, 2001.

[7] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. In *Proc.Intl. Conf. on Distributed Computing Systems*, pages 146–154, May 2003.

[8] L. Lamport. On interprocess communication, part I: Basic formalism. *Distr Comp.*, 1(2):77–85, 1986.

[9] M. Malek. A comparison connection assignment for diagnosis of multiprocessor systems. In *Proc. of the 7th Annual Symp. on Computer Architecture*, pages 31–36, 1980.

[10] D. Malkhi, Y. Mansour, and M. K. Reiter. On diffusing updates in a byzantine environment. In *Symp. on Reliable Distributed Systems*, pages 134–143, 1999.

[11] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[12] J. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proc. of the Intl. Conf. on Dependable Systems & Networks*, pages 374–383, 2002.

[13] F. P. Preparata, G. Metze, and R. T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Trans. Electronic Computers*, pages 848–854, Dec 1967.

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