Fairly Redistributing Failed Server Load in a Distributed System*

Venkatesh Sangam, Christopher B. Mayer **, and K. Selçuk Candan

Computer Science and Engineering Department
Arizona State University
{venkatesh.sangam, chris.mayer, candan}@asu.edu

Abstract. We recently proposed a novel method for large-object replication and load balancing. Our method is particularly well-suited to data grids, data warehousing providers, and hosting of dynamic web sites. The method attempts to distribute object request load fairly to servers according to server capacity so that the likelihood of them overloading, and hence failing, is reduced. Unfortunately, server failures cannot be eliminated entirely. When a server fails, the load carried by that server must be absorbed by the rest of the system. Unless this load is distributed fairly across the remaining servers, they may also overload, creating a cascade of failures and reduced quality of service. In this paper, we propose an efficient method for fairly redistributing the load of a failed server or set of failed servers within our replication system. We also report on experimental results that verify the validity of our approach.

1 Introduction

We are investigating the replication of large data objects in distributed environments. Such environments include data grids, data warehousing services, and dynamic web site hosting. Individual data grid members (typically research institutions) must maintain data generated by their own endeavors and be ready to share that data with other data grid members [7]. Data warehousing services archive data from numerous sources. Typically this data is stored as views that are made available to the warehouse's clients. Dynamic web site owners outsource their sites to hosting services in order to obtain an instant Internet presence and economies of scale. The hosting service maintains copies of the databases and application logic constituting a dynamic web site on its servers.

In order to ensure availability and meet user demand, replication of the data (and any needed application logic) in each of these example environments may be required. These environments share several properties which must be considered in any replication solution.

* This research funded by NSF grant 998404-0010819000.

** The views expressed in this paper are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

- **The basic unit of data access is on the order of megabytes.** For example, the basic unit of access for a data grid is an experiment or a series of data collection efforts, a view for a warehouse, and the database for a dynamic web site. Data size precludes rapid, on-demand replication. Thus content needs to be pre-positioned in anticipation of demand.
- **Data objects number in the hundreds or low thousands with new objects being introduced infrequently.** Since we are dealing with a low number of infrequently changing objects, it becomes possible to locate servers with desired objects using a directory service.
- **Replication may be implemented as a distributed system.** The replication system may use dozens or hundreds of servers located in multiple buildings or scattered across the globe. Servers may vary in terms of performance.
- **Servicing user requests is the main contributor to server load.** While updating data on a server is also a source of load, updates are infrequent and can be effectively ignored. For our purposes, a server's load is only its read load – the load of servicing user requests.
- **Response times are high.** In the case of data grids and warehouses, transmitting megabytes of data dominates the response time. For dynamic web sites, processing a user's query through the database is the main response time component [9].
- **Server load is a key concern.** Transmitting large amounts of data or performing database queries uses a significant amount of server capacity. Servers become sluggish and can appear to fail when they become overloaded. The main way to ensure system responsiveness is to control server load. Controlling responsiveness by placing objects close to users (see [1, 2, 5, 6, 8] for examples) is not as important.
- **There is an authority that can dictate the rules with which various data objects are replicated.** For instance, a web data hosting service can decide how its clients' sites will be replicated on its many servers and how the requests to those replicas will be directed.
- **The authority has no control over which objects user will access nor which server users will contact when initiating a request.** Also, we cannot assume that users have any knowledge of the replication system's inner workings. Therefore, the authority must direct requests to the most suitable replica server in a manner transparent to the users. As with many distributed systems, each user has a regular set of entry points (servers) from which they access the system.

In a previous work [3], we proposed an architecture and design method for replication systems for large object environments such as those mentioned above. While it addresses the issues listed above, our proposal also focuses on distributing system read load to servers in proportion to each server's capacity. That is, a server should expect to shoulder the same fraction of system load as it contributes to system capacity. Such a server is said to be **fairly loaded**. The advantages of a load-fair system are twofold. First, systems can use servers of

varying capacities. Secondly, server load tracks system load so that it is unlikely that a server will overload (temporarily fail) unless the system is approaching its capacity limit. In [3] we showed how to construct highly load-fair systems when all servers are operating. We will review our proposed architecture later.

In this paper we extend the work of [3] by showing how to maintain load fairness in the face of server failures. In a load-fair system server failures can be extremely detrimental. A failed server means lost system capacity and may make certain system services or data unavailable, which is bad enough. Worse though, the load normally carried by a failed server must be distributed to other servers in order to maintain quality of service. To reduce the likelihood of more failures, the failed server's load should be redistributed to other servers so that load fairness is maintained.

The paper is organized as follows. Section 2 contains a review of our replication system architecture. In Section 3 we explain how failures affect our replication system. Section 4 describes how to maintain fairness when servers fail. The effectiveness of our failure handling strategy is demonstrated in Section 5. Section 6 summarizes our efforts.

2 Replication System Background

Our replication system organizes a system's servers, $S = \{s_1, s_2, \ldots, s_n\}$, into sets called write-sets, W, and read-sets, R. Replicating a data object is done by selecting a write-set and copying the object to all the servers in it. To access an object or perform a query, a read-set is selected and an appropriate server from that read-set is selected to perform the requested action. To ensure that user requests can always find a server, every write-set must intersect every read-set. The system implementer (data grid member, warehouse service, or hosting service) decides to which write-sets an object should be placed and relocates objects in order to balance load across the write-sets.

Since each server in the system can have a different capacity, forming write-sets and read-sets that promote load fairness can be difficult. To make it easier, we transform each server into a number of **virtual servers** as follows. For each $s_i \in S$, turn s_i into n_i virtual servers by first dividing s_i's data transmission capacity, C_i, by a capacity, C_{base}, that is less than or equal to the capacity of the lowest capacity server in S and then compute the floor of the result, i.e., $n_i = \lfloor \frac{C_i}{C_{base}} \rfloor$. The set of s_i's virtual servers is $V(s_i) = \{s_{i,1}, s_{i,2}, \ldots, s_{i,n_i}\}$. Notice that each virtual server has about the same amount of capacity. Instead of forming write-sets and read-sets from physical servers, we form them from virtual servers. Note that virtual servers are instances of the same physical server. Each virtual server of a physical server has access to the same content as its physical server. The content on a physical server is determined by its membership in write-sets via its virtual servers.

In [3] we used a simple grid structure for establishing write-sets and read-sets, where the l rows of a grid become the write-sets and the k columns are

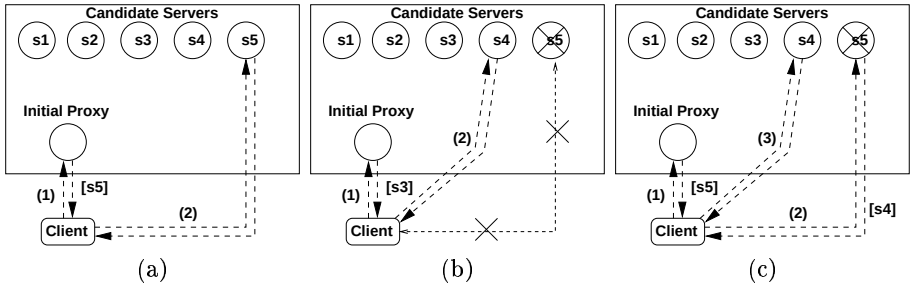


Fig. 1. Examples of replication system operation. In (a) there are no failures. In (b) server $s5$ has hard-failed and the proxy must act as the failure-handling entity. In (c) $s5$ is soft-failed and acts as its own failure-handling entity by redirecting requests sent to it by a proxy to other servers.

the read-sets.[1] We showed how to map virtual servers, one per grid cell, to form write-sets and read-sets that promote fairness. Using this structure, each virtual server appears in only one write-set and one read-set. However, because a server can have many virtual servers, a server can be in multiple write-sets and read-sets. We also introduced the following generic read protocol.

1. The user selects a server (a **proxy**) and sends an **initial-request** indicating the content desired to the proxy.[2]
2. The proxy selects a read-set according to a probabilistic **proxy strategy**.
3. The proxy identifies the server(s) in the selected read-set capable of handling the request by consulting directory service or a lookup table. If more than one server is capable, one of them is picked equiprobably.
4. The proxy redirects the request to the selected server.
5. The selected server processes the request and returns a response to the user.

This read protocol is illustrated in Fig. 1(a). In step 1 a user (client) contacts a proxy with a request. The proxy then selects a read-set (candidate servers) and determines which server should handle the request, in this case s_5. The proxy redirects the user to s_5 and s_5 fulfills the user's request (step 2).

Note that proxies actually select virtual servers during the read protocol. A virtual server's "load" is determined by its selection by proxies. Since a server's load is the sum of the "load" of its virtual servers, a server's load is a factor of the proxy strategies and the structure of the write-sets and read-sets.

Using this read protocol and write-sets and read-sets formed from virtual servers, we showed in [3] how to formulate proxy strategies that, given a pattern of initial-requests to the proxies, distributes load fairly to read-sets and

[1] Although we used a grid, almost any logical structure can be used. The method for promoting fairness depends on the structure. For this paper, what is important are the write-sets and read-sets and that they intersect.

[2] While we consider proxies to be a server which contains replicated content, the proxy may exist solely to direct requests.

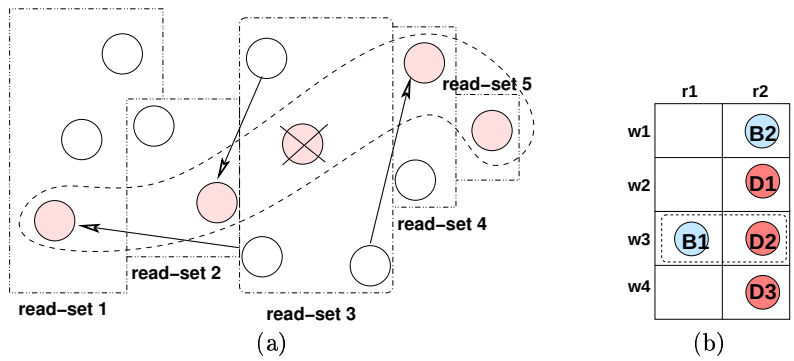


Fig. 2. A failure example using write-sets and read-sets. In (a) a failed server (marked with an X) causes servers in read-set 3 to use servers outside the read-set to access content in the write-set (the servers contained inside the dashed line). In (b) virtual servers allow redirection within a read-set.

ultimately to the system's servers. In [4], we examined the sensitivity of our approach to variations in user requests arriving at the proxies.

3 Fairness and Failures

A fair replication system distributes system read load to its member servers such that the fraction of load each server experiences is equal to its contribution to the system's capacity. Equivalently, the loads experienced by any two servers should be roughly equal to the ratio of their capacities. Modeling a replication system as a set of servers, $S = \{s_1, \ldots, s_i, \ldots, s_n\}$ where each server has a known amount of serving capacity, C_i, the **fairness condition** to be enforced can be written as:

$$\forall s_i, s_j \in S, \ \frac{L(s_i)}{L(s_j)} \approx \frac{C_i}{C_j}. \tag{1}$$

Replication systems built and operating as described in [3] have a high degree of fairness when all servers are operating. However when a server fails, the system becomes unfair causing more servers to fail due to overloading.

When a server fails, another server needs be selected to serve the user's request. Since servers in a write-set have the same content, this **replacement server** should come from the same write-set as the failed server. Selecting a replacement server has to be done quickly so that response times remain low. Of course, we do not always know that a server has failed before we attempt to use it. Requests sent to a failed server will unavoidably have poor response times. However, once detected, we can avoid the failed server and keep response times low by directing requests only to live servers.

One way to work around a failed server is to avoid selecting read-sets with failed servers. However, the proxy strategies for selecting read-sets are meant to

ensure system-wide fairness. If we were to suddenly start avoiding read-sets with failed servers, system-wide fairness could be severely affected. A better solution is to continue selecting read-sets according the proxy strategies, but to select other servers within the read-set *or* modify the read-set's membership to include servers with the same content as the failed server. Thus we can avoid paying time and fairness penalties associated with choosing another read-set.

Virtual servers allow write-sets and read-sets to intersect at multiple servers. Intersecting at multiple servers decreases the need to modify a read-set's membership. For example, in Fig. 2(a) there are no virtual servers. When a server fails (the crossed-out server), requests arriving at servers in the failed server's read-set must be redirected to servers outside the read-set. However, if virtual servers are used, redirection can take place within the read-set and no modification is needed. Figure 2(b) shows an example of this. Here, write-set $w3$, which contains virtual servers of B and D, intersects read-set $r2$ with four virtual servers. If D fails, B can still service requests directed to $r2$ for objects stored on $w3$.

Suppose that a system using virtual servers is load-fair when no servers have failed. While write-sets and read-sets can intersect with multiple servers, a failed server may still cause an imbalance in load fairness. Also, it may not be possible to ensure such intersections in all cases. Thus, we must consider modifying a read-set's membership to have a better chance of keeping the system fair.

4 Preserving Fairness When Servers Fail

Above, we made the case for introducing new servers into a read-set to replace a failed server and to preserve fairness. In this section we introduce a method for selecting these replacement servers in a way that promotes load fairness for the remaining live servers. We tackle this in a number of steps. First, we list driving concerns that the method should meet. Next, we introduce the concept of replacement sets and then state the constraints that need to be met in order to ensure fair load distributions. We then present a technique for maintaining fairness when servers fail, first for the simple case of single-server failures and then for the more complicated case where several servers have failed simultaneously. Finally, we discuss the information needs and responsiveness of our failure handling technique.

4.1 Driving Concerns

The way in which a read-set is modified can have a big impact on the performance (in terms of cost, response time, failure probability etc.) of the overall system. In addition to maintaining fairness, there are the following concerns.

- Creating the re-routing plan and the actual re-routing of requests should both be fast.
- Since the system might be distributed, the changes to the routing policy should be computed in a distributed fashion, too. The entity responsible for

redirecting requests around a failed server, the **failure-handling entity**, should use mostly local information.
- The failure-handling entity can be located at many places within the system.

To further motivate our approach for preserving fairness and to see how these concerns relate, consider Fig. 1 which shows two possible failure-handling entities and how they deal with the failure of server $s5$. In both parts of the figure, $s5$ has been selected by the proxy for handling a user's request. In Fig. 1(b) $s5$ has "hard-failed" and is totally unresponsive. Someone else, such as the proxy, must handle request redirection. Any requests sent to $s5$ before the proxy learns of $s5$'s failure will have enlarged response times. In Fig. 1(c) $s5$ has "soft-failed"; it has communication capabilities but cannot fulfill the user's request (perhaps the disk with the needed data is down). Knowing its problem, $s5$ acts as its own failure-handling entity and redirects the user's request to server $s4$. A server, like $s5$, that is its own failure-handling entity saves other system components the burden of redirecting requests. Also, since the soft-failed server is immediately aware of its problems, the lag time in detecting and adapting to failures can be reduced. Thus, we see the advantages of having multiple locations for the failure-handling entity. We also see an overall need for fast re-routing of requests.

4.2 Replacement Sets

For each virtual server in the system, we can define a **replacement set**. A replacement set consists of all the servers that appear in the virtual server's write-set. Since (*i*) all servers in a given write-set have the same objects and (*ii*) a server is in a write-set if it has a virtual server the write-set, only the servers in the replacement set can be replacement servers. Replacement sets conform nicely with our read protocol. A proxy can select a read-set and a server (via a virtual server) within that read-set as the read protocol states. If the selected (virtual) server has failed, the request can be redirected to a server in the virtual server's replacement set. The example below illustrates the replacement set concept.

Example 1. We have a small replication system using six servers A, B, C, D, and E. The servers have the following capacities: $C_A = 4$, $C_B = 4$, $C_C = 2$, $C_D = 3$, and $C_E = 3$. The number of virtual servers for a server is equal to the server's capacity. Using a grid to define write-sets and read-sets, a possible mapping of virtual servers is shown below.

A1	B1	C1	D1
B2	A2	A3	E1
E2	B3	D2	C2
D4	A4	B4	E3

Let us assume that server E fails. E appears in the second, third, and fourth rows (write-sets) and first and fourth columns (read-sets). Using $R(x)$ to denote the replacement set for virtual server x, the replacement sets for E's virtual servers are: $R(E1) = \{A, B\}$, $R(E2) = \{B, C, D\}$, and $R(E3) = \{A, B, D\}$. $\diamond$

Unless replacement sets are used carefully, redistribution of load may not be fair. For instance, in the above example, server B would get a higher portion of E's load than server A if the load is uniformly distributed within the replacement sets since B is in all three of E's replacement sets versus two for A. However, both servers have the same capacity (4 each). To be fair, they should share an equal portion of E's load. In the next section, we formulate constraints that ensure that the load of the failed servers is redistributed fairly.

4.3 Fairness and Replacement Sets

Say server s_f fails. To fairly distribute its load, the load s_f would normally have carried must be directed to the servers in its replacement sets in proportion to their capacities. Below we list conditions that, when satisfied, ensure a fair distribution. Before describing these conditions, we need to define some terms and introduce some notation. Let

- $S = \{s_1, s_2, ..., s_n\}$ be a set of servers, where each server s_i is split into a set of n_i virtual servers, $V(s_i) = \{s_{i,1}, s_{i,2}, ..., s_{i,n_i}\}$;
- C_i denote the serving capacity of server s_i;
- $w(s_{i,j})$ be the write-set containing the jth virtual server of server s_i;
- $R(s_{i,j})$ be the replacement set for the jth virtual server of server s_i such that
$$R(s_{i,j}) = \{\{s_k | s_k \in w(s_{i,j})\} \ - \ s_i\}$$
(i.e., $R(s_{i,j})$ contains servers in the same write set as $s_{i,j}$ except for s_i itself);
- $R(s_i) = R_{i,1} \cup R_{i,2} \cup ... \cup R_{i,n_i}$ be the set of replacement servers of server s_i;
- $L(s_{i,j})$ be the load on the jth virtual server of server s_i;
- $X(s_i)$ be the **extra load** on server s_i due to directing requests from failed server s_f ($s_i \neq s_f$) to s_i.

The constraints for fairly distributing the load of failed server s_f are as follows.

- **Constraint 1:** Servers in s_f's replacement set get a fair share of s_f's load
$$\forall s_i, s_j \in R(s_f), \ \frac{X(s_i)}{C_i} = \frac{X(s_j)}{C_j}. \tag{2}$$
- **Constraint 2:** The amount of extra load replacement server s_i receives depends on the load of each the failed server's virtual servers and the **redirection probability**, $p(s_{f,j}, s_i)$, of sending requests from each of those virtual servers, $s_{f,j}$, to s_i
$$\forall s_i \in R(s_f), \ X(s_i) = \sum_{s_{f,j} \in V(s_f)} p(s_{f,j}, s_i) \cdot L(s_{f,j}). \tag{3}$$
- **Constraint 3:** All of a failed virtual server's load must be redirected to servers in its replacement set
$$\forall s_{f,j} \in V(s_f), \sum_{s_i \in R(s_{f,j})} p(s_{f,j}, s_i) = 1.0. \tag{4}$$

Next, we show how to use these constraints (Equations 2 – 4) to calculate redirection probabilities for single-server and multiple-server failure scenarios.

Let s_f be the failed server

1. Get the replacement sets for s_f.
2. Calculate redirection probabilities for each server in the replacement set using Equations 2 – 4.
3. When one of s_f's virtual servers, $s_{f,j}$, is selected for handling a request, substitute a server in the replacement set of $s_{f,j}$ based on the redirection probabilities.

Fig. 3. Pseudocode for calculating redirection probabilities.

4.4 Calculating Redirection Probabilities

The algorithm in Fig. 3 outlines how to calculate redirection probabilities. Example 2 demonstrates how the algorithm might be employed for a simple single-failure scenario.

Example 2. Using the system and replacement set from Example 1 let

- $p(E1, A)$ and $p(E1, B)$ denote the redirection probabilities for servers A and B for $E1$'s replacement set $R(E1)$,
- $p(E2, B)$, $p(E2, C)$, and $p(E2, D)$ denote the redirection probabilities of servers B, C, and D for replacement set $R(E2)$, and
- $p(E3, A)$, $p(E3, B)$ and $p(E3, D)$ denote the redirection probabilities for servers A, B, and D for replacement set $R(E3)$.

Via (4) we have:

$$\begin{aligned} p(E1, A) + p(E1, B) &= 1.0 \\ p(E2, B) + p(E2, C) + p(E2, D) &= 1.0 \\ p(E3, A) + p(E3, B) + p(E3, D) &= 1.0. \end{aligned}$$

Using (3) we can express the extra load on each server due to the failure of E as

$$\begin{aligned} X(A) &= p(E1, A) \cdot L(E1) + p(E3, A) \cdot L(E3) \\ X(B) &= p(E1, B) \cdot L(E1) + p(E2, B) \cdot L(E2) + p(E3, B) \cdot L(E3) \\ X(C) &= p(E2, C) \cdot L(E2) \\ X(D) &= p(E2, D) \cdot L(E2) + p(E3, D) \cdot L(E3) \end{aligned}$$

Using the read protocol and assuming objects in each write-set are equally popular and that read-sets are selected uniformly, we use the methods of [3] to determine that the fraction of system load on each of E's virtual servers is $L(E1) = 0.08125$, $L(E2) = 0.05347$, and $L(E3) = 0.08125$. To balance the extra load fairly, (2) has to be satisfied, so we write

$$\frac{X(A)}{C_A} = \frac{X(B)}{C_B} = \frac{X(C)}{C_C} = \frac{X(D)}{C_D}.$$

Solving these equations gives numbers for the redirection probabilities which ensure a fair distribution of load: $p(E1, A) = 0.1821$, $p(E1, B) = 0.8179$, $p(E2, B) = 0.0$, $p(E2, D) = 0.6214$, $p(E2, C) = 0.3786$, $p(E3, A) = 0.6358$, $p(E3, B) = 0.0$, and $p(E3, D) = 0.3642$. ◇

When multiple servers fail, the calculation of redirection probabilities is more complicated for two reasons. First, replacement sets may contain failed servers. Since a failed server cannot act as a replacement server, failed servers have to be removed from all replacement sets of which they are members. Secondly, newly failed servers and any failed server that has a newly failed server in one or more of its replacement sets need to have their redirection probabilities (re)calculated. We demonstrate how to handle multiple server failures in the following example.

Example 3. We continue Example 2 where E was the only failed server. Suppose server A also fails. A's failure means that E's replacement sets and redirection probabilities need to be recomputed. Since the extra load that can be carried by server A in the replacement set of E is now zero (i.e., $X(A) = 0.0$) we need to set the redirection probabilities for server A to zero by making $p(E1, A) = p(E3, A) = 0$. The load and redirection constraints become, respectively,

$$\begin{aligned} X(B) &= p(E1, B) \cdot L(E1) + p(E2, B) \cdot L(E2) + p(E3, B) \cdot L(E3) \\ X(C) &= p(E2, C) \cdot L(E2) \\ X(D) &= p(E2, D) \cdot L(E2) + p(E3, D) \cdot L(E3) \end{aligned}$$

and

$$\begin{aligned} p(E1, B) &= 1.0 \\ p(E2, B) + p(E2, C) + p(E2, D) &= 1.0 \\ p(E3, B) + p(E3, D) &= 1.0. \end{aligned}$$

The fairness constraint becomes

$$\frac{X(B)}{C_B} = \frac{X(C)}{C_C} = \frac{X(D)}{C_D}.$$

Solving these equations gives new redirection probabilities for E: $p(E1, B) = 1.0$, $p(E2, B) = 0.0$, $p(E2, C) = 0.8795$, $p(E2, D) = 0.1025$, $p(E3, B) = 0.1891$, and $p(E3, D) = 0.8186$.

Since E has failed, the replacement sets for A's virtual servers are $R(A1) = \{B, C, D\}$, $R(A2) = \{B\}$, $R(A3) = \{B\}$, and $R(A4) = \{B, D\}$. The redirection probabilities for A are calculated using these replacement sets. ◇

Sometimes a system of equations may not be solvable because there is no solution that is perfectly fair. To account for this, we can add error terms (that describe how far we are from satisfying a constraint) to each of the constraints. The error terms catch any load that cannot be distributed fairly. Once modified, we can solve a system of equations as a linear program, with the objective of minimizing the sum of the error terms, to get the fairest solution possible.

4.5 Information Requirements and Solution Times

We mentioned back in Section 4.1 that failure handling should be distributed in nature and fast. We now discuss these two issues as they relate to our approach.

The information needed to calculate redirection probabilities do not conflict with our requirement that failure handling be a distributed process. The information needed by each failure-handling entity includes:

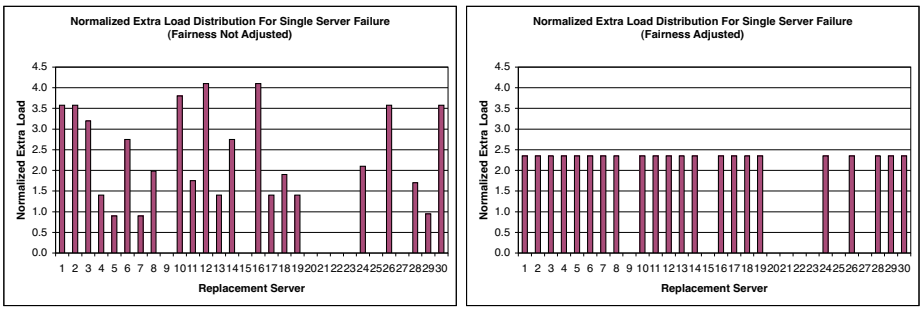
- **The read-sets under the entity's purview and the virtual servers in those read-sets**. This is local knowledge that should not change much over time. Since there are many redirection entities each of which is responsible for a small number of read-sets, storing this information is a minor burden.
- **The replacement set for each virtual server for which the entity is responsible**. This is global knowledge. However, since replacement sets are derived from the write-sets, which are highly static, the replacement sets do not change often. Thus, each entity can be provided with the replacement sets it needs to know about whenever those sets change.
- **The capacities of each server that may be chosen for redirection**. Since servers enter and leave the system relatively infrequently, making this information available globally is not a problem.
- **The load on each virtual server**. This information can come from either of two sources. If the system is kept perfectly tuned such that it is running as it was designed (as described in [3]), then the load on each virtual server can be computed at system (re-)design time and distributed to all entities. The second source is to have servers monitor how much load is coming from its virtual servers and report that load to the appropriate redirection entities.
- **The failed servers in the union of its replacement sets**. This information can be obtained by pinging servers or through communication timeouts.

As for the speed of calculating redirection probabilities, formulating, but especially solving, a linear program can take a while, even though the solution time is polynomial. We can get around this potential difficulty by pre-calculating some probabilities in advance, for example, for single server failures. Pre-calculating for multiple server failures is not feasible since the number of multiple failure scenarios is exponential. However, we note that some multiple server failure scenarios can be rather common (e.g., servers relying on the same network link), making pre-calculation feasible. In this paper, we do not consider the problem of identifying correlated multiple-failure scenarios.

5 Effectiveness of the Approach

In order to validate our approach, we have conducted experiments for both single- and multiple-server failure scenarios. This section details the results.

To see how fairly load is distributed, we compare replacement servers in terms of **normalized extra load**. Normalized extra load is the percentage of a failed server's load (or failed servers' load in case of multiple failures) directed to a replacement server divided by the number of virtual servers the replacement server has. Thus, if load is redistributed fairly, then all the replacement servers in a failure scenario will have the same amount of normalized extra load. For a given failure scenario, the standard deviation of normalized extra load for



FV = 1.1 (a) FV = 3.68e-8 (b)

Fig. 4. Example normalized extra load distributions for a single-server failure scenario when (a) fairness is not enforced (b) and when it is enforced.

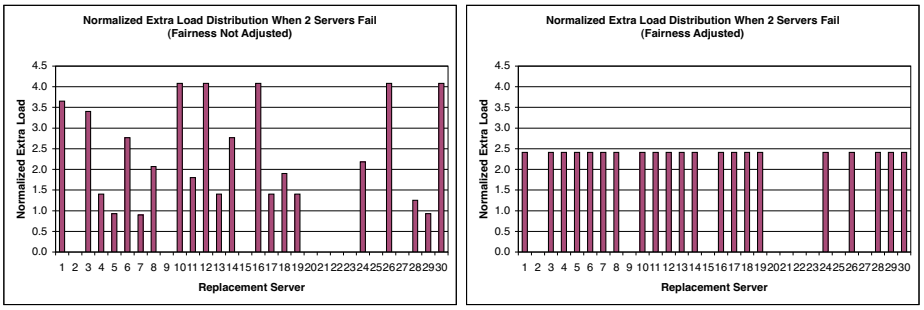
	fairness not attempted	fairness attempted
mean FV	0.59	0.48
num. scenarios w/ FV $\approx$ 0.0	0	19

Fig. 5. Comparison of fairness values (FVs), for single failures when enforcing fairness was and was not attempted.

the replacement servers, or **fairness value** (FV), indicates how well load was distributed. An extremely fair system has an FV close to 0.0.

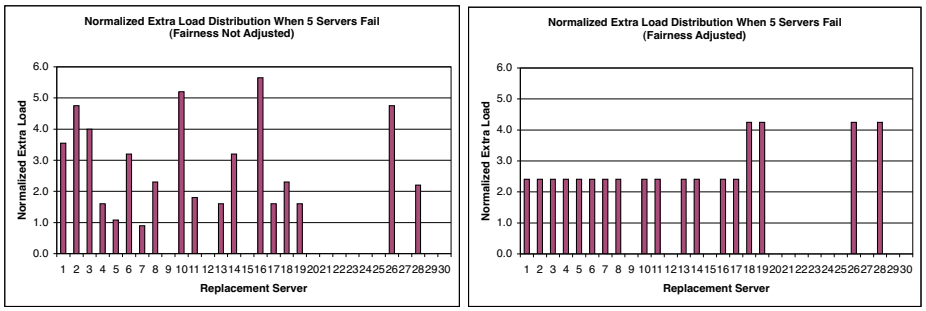
In the first set of experiments an 8×8 grid containing 30 servers was used to test the effectiveness of our approach when only one server fails at a time. Each of the 30 servers was independently failed and the loads on the other servers with and without fairly redistributing load was calculated. When fairness was not attempted, replacement servers were selected equiprobably *within the write-set of each failed virtual server*. A typical result is shown in Fig. 4 for the scenario where server 9 fails. When we do not ensure fairness, the servers in 9's replacement set (all servers except 15, 20-23, 25, and 27) are not fairly loaded as evidenced by the uneven bars of Fig. 4(a). When the fairness algorithm of Fig. 3 is used, a fair load distribution results as the level bars of Fig. 4(b) indicate. In this particular scenario, the unfair system has an FV of 1.1 while the fair system has a value of 3.68e-8; the fair system is about 30 million times as fair. Figure 5 compares the FVs of fair systems to those where fairness was not attempted and shows how fairly load was redistributed across all the single-server failure scenarios. While the mean of the FVs in all 30 scenarios are close regardless of whether or not fairness was enforced, the number of scenarios experiencing very low fairness values is much greater (19 versus 0) under fairness.

The second set of experiments focused on multiple server failures. Using the same 8×8 grid as for single failures, groups of two and five servers were picked at random five times each and the servers in a group failed together. Figure 6 shows the normalized load distributions for a 2-failure scenario. Figure 7 contains an



FV = 1.2 (a) FV = 0.0 (b)

Fig. 6. Normalized extra load distributions when 2 servers have failed when (a) fairness is not adjusted and (b) fairness is adjusted.



FV = 1.5 (a) FV = 0.78 (b)

Fig. 7. Normalized extra load distributions when 5 servers have failed when (a) fairness is not adjusted and (b) fairness is adjusted.

example 5-failure scenario. In both figures we see that enforcing fairness does a much better job of distributing load than not doing so.

We also see in Fig. 7(b) that perfect fairness was not possible for the 5-failure scenario. This is not surprising since one-sixth of the system's servers had failed, leaving limited options. Instead, we had to settle for a redistribution that is the fairest possible under the circumstances (see the last paragraph of Section 4.4). Note that extra load was distributed fairly within two groups of servers (the normalized extra load assumes only two values). This is certainly better than not having attempted fairness at all as the FVs of Figs. 7(a) and 7(b) indicate.

Figure 8 provides a comparison of fairness values for the 2-failure and 5-failure scenarios. Notice that enforcing fairness produces lower average FVs than not doing so. Examining Figs. 5 and 8, we notice that the mean FV rises with the number of failures. Indeed, since redirection options decline as the number of failed servers increases, the odds of finding a fair solution should decline also.

	fairness not attempted	fairness attempted
mean FV for the 2-failure scenarios	1.2	0.3
mean FV for the 5-failure scenarios	1.4	1.0

Fig. 8. Comparison of mean standard deviations in server loadings, adjusted for capacity, for 2-failure and 5-failure scenarios.

Overall, the experiments indicate our methods for fairly redistributing failed server load works well and has a very good chance of succeeding.

6 Conclusion

We have reviewed our proposed method of replicating large objects such as those found in data grids, data warehouses, and dynamic web sites [3]. Since servers can fail because their read loads become too large, efforts need to be taken to distribute read load fairly to servers so that it is less likely that they overload.

In this paper, we proposed a method to maintain fair server loadings when servers fail. Subject to a set of constraints, we developed a distributed approach for deciding how to re-route requests that would normally have been handled by failed servers to a set of replacement servers. Re-routed requests are distributed fairly to the replacement servers in order to prevent any one server from overloading due to its increased number of requests. We also presented experimental results that verify the validity of our approach.

References

1. J. Kangasharju, J. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. In *WCW*, 2001.
2. B. Li, M. J. Golin, G. F. Italiano, X. Deng, , and K. Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM*, pages 1282–1290, 1999.
3. C. Mayer, K. S. Candan, and V. Sangam. Constraints, parameters, and strategies for replicating large content for web delivery. Technical report, Computer Science and Engineering Department, Arizona State University, 2002.
4. C. B. Mayer, K. S. Candan, and V. Sangam. Effects of user request patterns on a multimedia delivery system. In *Intl Workshop on Multimedia Info Systems*, 2003.
5. L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, pages 1587–1596, 2001.
6. P. Radoslavov, R. Govindan, and D. Estrin. Topology-informed internet replica placement. In *WCW*, 2001.
7. H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. In *HPDC*, pages 76–86, 2001.
8. R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design considerations for distribued caching on the internet. In *ICDCS*, pages 273–284, 1999.
9. U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of e-commerce traffic. In *WECWIS*, pages 137–146, 2002.