An Efficient Scheme for Dynamic Data Replication

Swarup Acharya and Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-43
September 1993

An Efficient Scheme for Dynamic Data Replication*

Swarup Acharya
Stanley B. Zdonik
Dept. of Computer Science
Brown University
Providence, RI 02912-1910
email: {sa,sbz}@cs.brown.edu

Abstract

This paper presents an efficient scheme for dynamic replication of data in distributed environments. The aim of the scheme is to increase system performance by intelligent data placement so as to optimize the message traffic in the network. Research in the recent past has comparatively focussed very little on using replication for increasing performance but has instead been directed more at improving system availability through replication. However, with the advent of mobile or nomadic computing, research in replication needs to change direction– the underlying assumption of high speed networks no longer hold true. Wireless networks not only have lower bandwidth but are also very expensive to use. In such an environment, it is imperative that data be distributed intelligently to achieve a good system performance in terms of message costs and turnaround time. Besides, with mobility introduced in the system, earlier static schemes for improving performance (e.g., the File Allocation Problem [DoF82]) are no longer applicable. As a first step, we propose a totally distributed dynamic replication scheme, which uses a finite automaton based technique to learn access patterns. This accrued information is then used to predict future access sequences and dynamically reorder– replicate or delete copies of data in the network, in anticipation of predicted accesses, to reduce network message costs. Being distributed, each node of the network bases its decision to replicate or delete an existing copy solely on local information. Our algorithm is also integrated and corrects some of the drawbacks of earlier theoretical work in this area. Finally, we present some preliminary simulation results which show that our scheme is very promising.

*This research was supported in part by DEC Research Agreement 1139.

1 Introduction

Replication of data is an essential component of distributed databases. There are two major motivations for replication– increasing availability and increasing system performance. Replication creates redundant information in the network, which allows the system to remain operational in spite of node and link failures and thus increase reliability. Also, if data is replicated near the node where it is accessed, communication cost is greatly reduced. However, replicas incur space overhead and require special effort to maintain data consistency.

Research in the recent past on replication has focussed mainly on the availability issue ([AhA89], [JaM90], [PaL88], [KRS93], [AgE90]) though the performance angle of replication has received attention earlier ([DoF82]). Improvements in technology have been mainly responsible for such changes in focus. When systems evolved from centralized to distributed systems, the network bandwidth was relatively very low. That led to a focus on reducing network transmission cost and hence on the performance issue. But as distributed networks became commonplace and technology improved, communication costs were no longer as important as other site related resource costs. Consequently, research in replication started to address issues of fault tolerance in replicated systems. But with the advent of mobile networks, the underlying assumption of high speed links will no longer be true [AlK93]. In such an environment, users will have the ability to move from one place to the other and yet be connected to their computer systems through a wireless link. While the growth in physical network bandwidth has been tremendous (current technology will support ≈600Mbit/sec), it is very unlikely that the wireless bandwidth will be able achieve a comparable value. For example, NCR's WaveLAN achieves ≈2Mbit/sec, a typical bandwidth over the air. Besides, data transmission over the air is monetarily expensive too [Hay92]. Consequently, for such a system to achieve a level of performance comparable to current static systems, it is very crucial how data in the network is distributed. Not only should data be distributed to reduce message overhead but it may also have to take into account the monetary expenses of each operation. Consequently, we predict that replication of data– how many replicas to have and where to place them – will be a very important issue in design of such systems. This does not imply that availability and fault tolerance issues in replication will become passé but that with the advent of nomadic computing, significance of performance issues in replication will greatly increase.

As a first step in the direction of intelligent replication for better performance, we propose a dynamic replication algorithm for a typical distributed network. The performance of a distributed system is very sensitive to the distribution of data among the nodes. Since a node reads an object[1] from the nearest neighbor having a copy (sometimes depending on the protocol, more than one replica may have to be read), it is very critical that objects be as near the nodes that read them. However, an update is usually done on many replicas (depending on the protocol followed, updates could be done on all or a majority or a write-quorum subset of the replicas) and this necessitates copies be as close to each other as possible. A good replication scheme should thus be able to correctly trade-off between the two conflicting criteria based on the read-write pattern of the object.

The algorithm we propose is adaptive, integrated [WoJ92a], online and distributed in nature. It

[1] A generic term – could be an object, file, relation etc.

continually reorders the replication scheme, adapting to changes in access patterns of each object. Unlike other algorithms which work in two phases– one phase in which read-write requests are satisfied and the other in which data is redistributed over the network, our algorithm *integrates* the movement of data with the reads and writes, thus saving messages. Being distributed in nature, any decision to save or delete an object from a node is made solely based on the information (statistics about reads and writes) available locally in each node.

An online algorithm should have the ability to learn characteristics of its environment to work well. We use a *deterministic finite state automaton* (DFSA) based learning technique to predict future object accesses. Based on these predictions we re-order the placement of objects (either shrink or grow the number of nodes with the object) to suit its (predicted) future access patterns.

The contributions of this paper are as follows. Firstly, we improve upon current theoretical work in the area of dynamic data distribution by correcting some impractical assumptions and bringing some realism into the models. In particular, ours is the first model that restricts the amount of memory in each node of the network as opposed to infinite memory in the system. Consequently, we have had to explicitly account for the *replacement* problem, akin to that in paging. Secondly, we present a novel direction of research in data replication by incorporating dynamic techniques which adapt efficiently to changes in access patterns in the network and thus perform better that conventional static schemes.

The rest of the paper is as follows. The next section describes the network model. Section 3 is about related research. Section 4 discusses basic distributed database terminologies and protocols. In Section 5, we describe the three states that an object can be in vis a vis its replication status. Section 6 gives the overview of the algorithm along with the tradeoff involved between read and a write in terms of the cost involved. Our learning model is described in section 7 and in section 8 the object deletion protocol is explained. Space and message overhead analysis is presented in Section 9. Finally, we present simulation results and offer our conclusions.

2 Network Model

Our network model is based on an earlier work by Jajodia and Wolfson ([WoJ92a]) and has the following features :

- The network is an undirected tree, T = (V,E), where V is the set of processors (nodes) and E the set of bi-directional links between two processors. The processors and links are assumed to be failure free.
- A *Replication Scheme* for an object is a subset of the nodes in the network which hold a copy of that object. The Replication Scheme for an object does not have to be a *connected*; it could be a set of *disjoint* subtrees. To be formal, we define a Virtual Replication Scheme (VRS, henceforth called RS) as the smallest connected subtree of the network containing all nodes which hold the copy of the object. The Real Replication Scheme (RRS) is just the set of nodes which hold a copy of the object. Thus, RRS $\subseteq$ RS (Fig 1).
- The replica consistency protocol is the Read One Write All (ROWA) protocol. Thus, a read is done from the nearest node having a copy but an update is transmitted to all nodes having a

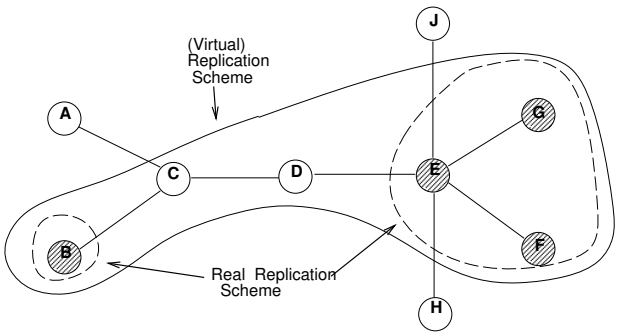

Figure 1: A typical network. Nodes with object a are shaded.

copy of that object. There exists at least one copy of every object somewhere in the network.

- The *Read Cost* of a node i for object a, denoted r_i^a, is the minimum number of messages needed to transfer a copy of the object to node i. This is usually some factor Ω_r, times the length (in edges) of the shortest path between i and a processor in the replication scheme which holds object a. If the node i has a copy, then a read incurs no cost.

- The *Write Cost* for node i to write object a, denoted by w_i^a, is the minimum number of messages required to update all the copies in the network. This is usually some factor, Ω_w times the number of edges in the smallest subtree of T that contains {i} $\cup$ RS.

- The size of memory in each node is restricted. The upper limit may vary for each node.

The values Ω_r and Ω_w may be the same or different depending on the type of concurrency control protocol used. For simplicity, in this paper we assume $\Omega = \Omega_w = \Omega_r$. We do not presume any particular concurrency control algorithm; our scheme works for any algorithm preserving one copy-serializability. Our algorithm aims at optimizing the communication cost in terms of messages involved.

Though our model is similar to [WoJ92a], there are some major differences as follows :

- Every node has a restriction on its memory size. The [WoJ92a] model assumes infinite memory.

- The [WoJ92a] scheme restricted all nodes with copies of an object be contiguous. Ours relaxes this restriction and thus nodes with copies could be *disjointed* with *holes* (i.e., nodes which are in the RS but not in the RRS, e.g., subtree C-D in Fig 1).

- The cost function is different too– our model optimizes the number of messages involved in every read or write access whereas the [WoJ92a] model considers the number of edges involved.

3 Previous Work

As mentioned before, there is a huge volume of research in replication for increasing availability ([AhA89], [JaM90], [PaL88], [KRS93], [AgE90] to name a few). There have been three main approaches to increasing performance using replication– static schemes, dynamic schemes and schemes which relax the consistency criterion by allowing controlled divergence within replicas.

In most current systems, data is replicated in a static fashion at system design time, based on proba-

ble access patterns. The replication scheme remains fixed at run time. Since the optimal distribution problem is NP-complete, various heuristics have been proposed. It is also called the File Allocation Problem (FAP) and has been extensively studied ([DoF82]). The static scheme performs well if the access patterns are known a priori, but this is not always possible.

Dynamic schemes on the other hand continually re-order the placement of data and thus automatically react to changes in access patterns in the network. The need for dynamic replication schemes in mobile networks has been pointed out in [BaI92] and [ImB93]. Earlier work in dynamic schemes have been theoretical in nature – [BFR92] presented competitive algorithms for the problem. But these were not strictly distributed in the sense that one processor had to be aware of all requests. Besides, unlike ours, their algorithm was not integrated. [WoJ92a] give an optimal algorithm for the tree network. However, they make the impractical assumption of no restriction on memory in each node of the network. Consequently, the problem of replacement of objects, akin to that in paging, albeit with a different cost function, does not arise in their model. They use a simple learning scheme, which in fact, it is a special case of our DFSA based scheme and can be reduced from it by choosing parameters appropriately. The tree topology restriction was lifted in [WoJ92b] and [HuW93] but the models still assumed infinite memory. Besides, in an arbitrary graph message routing is a very important issue which was not addressed. [HuW93] was based on the primary copy scheme which requires nodes without a copy of an object to read from the primary-copy node rather than the nearest neighbor. Firstly, this requires more messages than necessary. Secondly, the book-keeping related to the replication scheme (e.g., determining when to grow the replication scheme), is done by a single node – the primary copy node and thus making it a bottleneck. In our case, every node can make such decisions to grow or shrink the replication scheme.

The third approach is to assume static replication but to relax the serializability criterion. Papers on quasi-copies ([ABM90], [BaM90]), lazy replication ([LLS88]), epsilon serializability ([PuL91]) and bounded ignorance ([KrB91]) fall in the category. The serializability relaxation can be done along two dimensions of time and space([ShR90]). While divergence may be acceptable in some cases (e.g., stock market data), strict consistency is a must in many distributed applications like banking databases. Our scheme is geared towards such applications.

A logical parallel to the dynamic data placement is that of caching. Though similar in nature (the cached copy is just another replica), the cost functions and domains are different. The only goal of caching is to improve the response time where as data replication may address many more issues like message traffic, turnaround time etc. Caching is a means to improve the client-server interaction where as dynamic replication addresses the issues of server-server interaction in distributed environments. In other words, the domain of caching is the main memory as opposed to data replication, which deals with (but is not limited to) the organization of data in the external storage. Unlike ours, if the goal was to use replication to improve the response time, some of the caching techniques would be applicable. Our objective function is reducing message traffic. A distributed system has to solve both these problems efficiently to perform well. Caching and prefetching is also one area where learning models are frequently used. These models

are often probabilistic [KrV91] , statistical [Sal91] or nearest-neighbor techniques [PaZ91]. As explained above, these models are just interested in predicting the next (few) object(s) accessed so as to be able to prefetch it (them). The learning system in the dynamic replication is interested in a longer future time slice and has to predict the likelihood of an object being accessed for a read vis a vis the likelihood of being accessed for a write. So, while in prefetching inter-object access relationships are learned (and reads and writes aren't differentiated), in data replication, knowledge about the read versus the write access pattern of the same object is accrued, independently of other objects. Our methodology uses a tree-shaped finite automaton and uses statistics stored at every state to make predictions.

4 Distributed Database Basics

Every node of a distributed database maintains a set of tables/directories to help it locate data items in the network. These tables typically contain an object identifier (which is unique over the network) and a boolean which tells if the node has a copy of the object. All nodes also maintain a list, called the *Direction Vector* (DV), for every object. The DV is a list of neighboring nodes which are in the *direction* of a copy of the object. The first element of the DV is the closest node with a copy. The DV is used to locate objects which do not reside in the node. Consider Fig 1. Nodes B,E,G and F are in the RRS for the object concerned. The DVs for some of the nodes are– node A: {C}, node C: {B,D} and node D:{E,C}. Notice the order of elements of the DV of nodes C and D– they are sorted by the direction of the closest copy. When a node receives a read request for an object (the node itself may be the source of this request), as part of the transaction semantics, it checks in the directory entry for the object. If it owns a copy, the request is satisfied locally at no cost. If not, the node passes on the request to the neighbor at the head of the DV (the closest copy). This request is propagated until it reaches a node with a copy. The procedure is similar for a write access except that the request is propagated to every node in the DV. Based on how soon the responses for the write request arrive back from the neighbors, the DV is re-ordered (if necessary) to bring closest neighbor with a copy to the head of the list. The DV could get out of date in the prolonged absence of writes. The directory also contains a boolean to mark if the copy is the last copy in the network. If so, it is never deleted until it is replicated elsewhere. For writes, we assume some concurrency control mechanism to guarantee serializability.

In the spirit of being completely distributed, we let every node view the network solely through its neighbors. The motivation for such a choice is as follows. Once hosts are allowed to move in a mobile environment, the underlying network structure will no longer remain static. Limiting a node's outlook to only its neighbors makes it independent of other nodes (except its neighbors). Consequently, the only node affected by changes to the network structure due to mobility and/or failures are just the neighbors; non-neighbor nodes can function uninterrupted. Since our goal is to finally extend our technique to such a scenario, it would be imprudent to assume otherwise. For example, consider Fig 1. A request for the object from node A would appear to node B (the closest neighbor with the object) as a request from C. Similarly node C would differentiate between requests from nodes A and B but not between D and

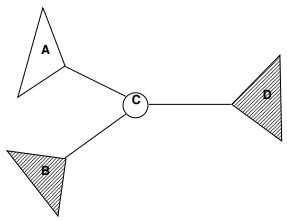

Figure 2: Network according to node C

E. So node C's perception of the network in Fig 1 is as in Fig 2. Notice that to node C, node D is in the RS though it does not own a copy. This is because the DV of node C is {B,D}. Henceforth, a statement of the form *node X receives a request from node Y*, means a request from any node in the subtree rooted at Y and not including X (and not just node Y alone). Similarly, a statement *an access at node* A implies an access generated not only by node A but also from its neighbors which get routed through A. The restriction of full distribution has been deliberately imposed on the algorithm. However, our algorithm is easily adaptable to networks where the restriction is relaxed. In fact, from the network layer point of view, the algorithm will actually be more efficient if it has knowledge of nodes with copies rather than just the direction.

At this point some definitions are in order. Based on where a node is in the RRS, it is either a *fringe* node or a *center* node. As the name suggests, a fringe node is on the fringe of the virtual replication scheme. Fringe nodes have a single element in the DV. For e.g., in Fig 1, B,F and G are fringe nodes for that particular object. All non-fringe nodes are center nodes. For e.g., node E in Fig 1 is a center node. A center node may have neighbors which are not in the RS (e.g., nodes E and H).

5 Object Status

Every object present in a node can be one of the three possible states based on its access pattern.

- *Live* : A *live* object is one which has very high likelihood of being read-accessed at the node (by the node itself and its neighbors) in the near future. This object will be flushed only if there are more live objects than the node has buffer space.

- *Dead* : A *dead* object has a strong likelihood of being written elsewhere in the network without being read at the node itself. Consequently, holding on to the object would just incur additional update messages which have to be propagated to this node. Since it is unlikely to be read, the motivation to hold onto it no longer remains. An object can be marked *Dead* only if the node is a fringe node in the RS. If so, it is flushed immediately.

- *Coma* : This is the equivalent of the *Dead* case for nodes in the *center* of the replication scheme. Being in the center, message traffic related to updating the object has to pass through this node independently of whether the node has a copy or not (e.g., in Fig 1, independent of the object's presence in node E, the cost of the write is going to be the same). So, unlike the *Dead* case, holding onto the object, does not increase the number of messages during an update. So, the

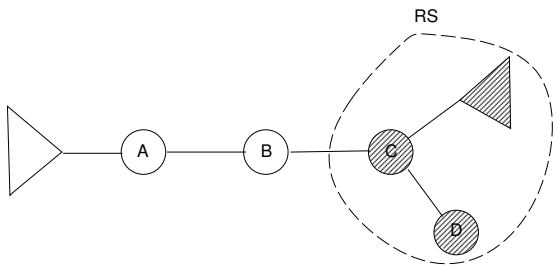

Figure 3: Read-Write-tradeoff

node holds onto the object marked *Coma* until either the space is needed for a new object or it becomes *Live* again.

6 Algorithm Overview

As has been pointed out before, a dynamic replication algorithm has to be able to trade off between read and writes to an object. If the object is being predominantly read, it makes sense to give a copy to all nodes reading it (save messages by avoiding remote reads). Alternatively, if it is regularly written, the RS should have as small a diameter as possible to avoid huge update costs. The problem of dynamic replication is, thus, composed of two complementary sub-problems – the problem of determining when to grow the replication scheme for an object to include read-intensive node(s) (*grow-RS problem*) and when to shrink the replication scheme in the face of increasing number of writes (*shrink-RS problem*). Both the problems have analogous solutions – each can be inferred easily from the solution of the corresponding complementary subproblem. Hence, in the rest of the paper we deal mainly with the grow-RS problem. As and when necessary, we briefly indicate how these solutions could be extended to the shrink-RS case.

6.1 Read Write Tradeoff

In this section we try to quantify the trade-off between reads and writes to an object.

Consider Fig. 3. Node C and the rest of the network on the right is in the RS for some object a. Let there be a significant number of accesses to this object from node A. There are two possibilities. Node A joins the replication scheme by holding onto the object after the next access or alternatively, it remotely accesses the object from C.

Considering the two cases separately,

- *Case 1:* Node A saves object a and joins the RS. Now, $r_A^a = 0$. But a write at C or any node in the original RS requires 2Ω (due to edges C-B, B-A) *extra* messages than what it did before A joined the RS.
- *Case 2:* The RS remains unchanged. In this case, $r_A^a = 2\Omega$ (nearest copy is at C). Writes at C or any of the other nodes in the RS of a requires no *extra* messages.

Thus if A joins the replication scheme, every write from the original RS, *cancels* a read at A, in terms of the benefit of joining the RS. Consider an example. Let the object a be read thrice by A and written once by C in the near future (we will clarify later what we mean by near future). In the case 1 scenario, 2Ω *extra* messages are required (update message from

C to A). If case 2 were true, 6Ω *extra* messages are required (remote read by A from C). Obviously, in this example, node A should have joined the replication scheme after the first read. Alternatively, if the object would have been read once by A and written thrice by C, the RS should have remained unchanged.

So, the problem of growing the RS for an object, then reduces to determining if that object would be read more often by the new node joining the RS and its neighbors or written more often from within the RS. If the former is true, then the RS is expanded to include the new node, otherwise, the RS is left unchanged.

In the general case where $\Omega_r \neq \Omega_w$, the picture would be marginally different. In that case, for a node to join the RS, it would not only have to read-access the object more often than it is written but it would have to do so by at least a factor of $\delta = \Omega_r/\Omega_w$. Since we assumed $\Omega_w = \Omega_r$, here $\delta = 1$. Our algorithm can be trivially extended to the general case.

The trade-off for the shrink-RS case is similar. Here a node owns a copy of an object and has to weigh the benefits of holding onto it (and thus saving during reads at the node) vis a vis the extra cost of update messages transmitted to it. On the lines of the argument above, a node should delete its copy if the object is going to be read less often than it is written elsewhere in the network. The read-write ratio could be a simple greater than or it could be a factor of δ for the general case. How exactly our learning system works and its prediction technique is explained in the next section.

6.2 Classifying Access Sequences

Consider a sequence of accesses arriving at a node. We extract a new sequence from it by filtering out accesses to a particular object. This extracted sequence consisting of reads and writes to this single object could be of two types – one, a sequence with more reads than writes or two, a sequence with more writes than reads (if they have the same number of reads and writes, we arbitrarily classify them as one or the other). We use this classification to label the original access sequence.

Let a *read-dominant* access sequence for an object at a node denote a sequence which has more reads to that object at the node (i.e., from the node and its neighbors not in the replication scheme) than writes from within the replication scheme[2]. Similarly, a *write-dominant* sequence is a sequence which has more writes than reads to the object. Based, on the arguments of the last subsection, a node should join the replication scheme if the learning system can predict a read-dominant sequence at the node. Alternatively, a node with a copy of an object should exit the replication scheme if it can predict a write-dominant sequence. The immediate problem then is to identify read and write-dominant sequences. We explain next how to do so by locating signatures typical to these sequences.

Any sequence can begin with one of the three prefix subsequences – successive reads(α), successive writes(β) or alternating reads and writes(γ). The prefix subsequences β and γ can never have more number of reads than writes. We now make a claim about read-dominant sequences. Any read-dominant sequence, i.e., one which has more number of reads than writes necessarily begins with at least two successive reads(prefix α). If it did not do so, it would have to begin with either subsequence β or γ. Since these subsequences can never have more reads than writes, all such leading β or γ subsequences can be safely pruned from the beginning without destroying

[2]By a factor of $\delta = \Omega_r/\Omega_w$ for the general case.

the read-dominance property. Such pruning could be repeated until the sequence began with (at least two) successive reads. Hence, every read-dominant necessarily begins with at least two successive reads (prefix α). So, the learning system can identify a potential read-dominant sequence by waiting for its prefix signature – two consecutive reads. However, a sequence beginning with successive reads isn't necessarily read-dominant. Consequently, the sampling window has to be of a longer length before a prediction can be made.

Parallel to the read-dominant case, the prefix signature of a write-dominant sequence is necessarily a subsequence of two or more successive writes.

We need to specify something more about the length of a sequence which has been labeled *read-dominant* or *write-dominant*. How long should a sequence be? Lets call the length *epoch_size*. Beginning at point when some object, say a is accessed, we track the next epoch_size number of accesses arriving or generated at the node. At the end of the sequence of *epoch_size* length, we filter out the accesses to a and compare the number of reads and writes. According to which is greater, the sequence is labeled read or write dominant for object a. What should epoch_size be? We believe it has to be related to the space in the memory in terms of the number of objects it can hold. Intuitively, it should be the length of time in terms of accesses to the node within which the object should be read at least once to pay for the cost of holding onto space in memory.

6.3 The Algorithm, Briefly

Before we describe the algorithm, a point has to be made about who decides when to grow or shrink the replication scheme. Consider the case of growing the RS. Is it the node wanting to join the replication scheme that requests permission or is it the node with the copy that instructs a new node to join the RS. We believe that the later should be true. The node wanting to join the replication scheme sees no accesses to the object except those arising from itself until it joins the RS and thus, is in no position to make a correct decision. Consequently, the node in the RS requests the new node to join the RS. On the receipt of such a message this new node saves the copy of the object and joins the RS. For the shrinking case, the node wanting to exit the RS can make the decision since being in the RS already, it has the complete picture of the access sequence.

We describe the algorithm to grow the RS using Fig 1. Let there be some means to determine a read or a write-dominant sequence. Only nodes in the RS of an object (node E in Fig 1) who have neighbors not in the RS (node H) are involved in the process. When E receives a read request for the object from H, it predicts if the sequence that follows would be read-dominant at H based on the accesses seen as yet. If not, H receives the object from E and flushes it after the transaction is completed. If the prediction is a read-dominant sequence, along with the data, E tags a message for H to join the RS. H saves the copy and tracks the next epoch_size requests arriving there, keeping a running total of the number of reads and writes to this object. Based on the scores, it determines the label of the sequence. If it was read-dominant, then the decision to replicate by E was correct, else, it was wrong. In either case, at the next write to the object, H gives this feedback (positive or negative) which E uses to adjust the learning system.

The algorithm to shrink the RS is analogous. In this case, only the fringe nodes in the RS are involved (nodes B,G,F in Fig 1). When there is a write request propagated to it from within the RS, the (fringe) node

predicts if the sequence to follow is write-dominant using access seen as yet. If not, it just updates its copy. If it is write-dominant, it exists the RS by deleting its copy. The shrinking could either be gradual (e.g., F exists) or it could be drastic (e.g., B exists shrinking the RS to just {E,F,G}). Analogous to the growing case, after epoch_size access, the corresponding node in the RS (in this case node E) informs this (ex)-fringe node about the correctness of its decision to exist the RS. Accordingly, the node updates the learning system.

The major difference between the growing and shrinking of the replication scheme is that while both processes could be gradual, i.e., an edge at a time, the later could also be drastic. The reason for this is the presence of *holes* in the RS, something unique to our model because of the restriction of memory in each node. Consider a possible scenario for the network in Fig 1. Initially the RS started out with nodes {E,F,G} and then gradually grew to include D,C and B. In due course, however the object is no longer read accessed at nodes C and D and is marked *Coma* (if they were *fringe* nodes, then the object would have been marked *Dead* and deleted immediately). When space for a new object was needed at these nodes, the *Coma* object was deleted giving rise to the *hole* in the RS. Consequently, when B deleted its copy, the RS suddenly shrank to just three nodes. If there were infinite memory in each node as in the [WoJ92a] model, there would have been no need to replace the *Coma* objects at all and thus the replication scheme would have been a contiguous subtree.

7 The Learning Model

In this section we explain our learning model, the data structures involved and how we use them to predict the class of a sequence.

We use a simple deterministic finite state automaton [HoU79] to learn access sequences. The DFSA makes state transitions on two tokens– `r` and `w`– read and write to an object. The DFSA stores information which is used to make predictions. A node maintains a DFSA per every neighbor per every object to record each neighbor's access pattern for each object. This is used by the node to decide whether to replicate to the neighbor. A node also maintains DFSAs for every object which it uses to decide whether to exit the replication scheme (this is also per neighbor). An example of a DFSA is in Fig 4. Since the DFSA resembles a binary tree, henceforth DFSA, FSA and tree will be used interchangeably.

As before, we will describe in detail the working of the learning system for the grow-RS problem followed in brief, by analogies to the shrink-RS case.

7.1 Data Structures

Each state of the FSA stores the following data structures:

- A boolean array *history_register* of length k. The *history_register* is maintained as a shift register wherein new information is inserted at one end and the oldest data shifted out at the other. The information is in the form of booleans – a 1 representing a read-dominant sequence, a 0 representing a write-dominant sequence.

- An integer field *score*. The value of the score is the difference of the number of 1's and 0's in the corresponding history_register. Informally, if the *score* of a state has a positive value, it implies that the sequence following the one seen from the root to this state is very likely *read-dominant*.

Alternatively, if it is negative, the sequence following it is a *write-dominant* sequence.

Besides, every FSA also maintains a location pointer, *Loc*, which denotes the current state of the tree.

Each FSA has two variable parameters – the height, H of the tree and k, the size of the *history_register*. The height of the tree denotes the maximum prefix sequence window sampled before classifying the sequence read or write-dominant. This size of the history_register is the window in the past history used as a basis for future prediction. Obviously, bigger the values of H and k, the better the prediction. Unfortunately space limitations force us to bound the size.

7.2 Tokens of the FSA

Consider the two nodes E and H in Fig 1, where node E has to decide whether to let node H join the RS. There are four classes of requests – reads and writes originating *at* H and reads and writes originating *at* E. Note that *at* implies not from the node alone but all requests which get routed through the node. We categorize which of the four are actually significant in the decision to replicate to H. Obviously, as explained earlier a read at node H and a write at E are significant. A read at E can be ignored. That leaves writes at the node wanting to join the replication scheme. We claim that it (in this case a write at H) is not significant either. The cost of a write initiated at H is the Ω times the number of edges in the subtree {H}$\cup$RS. This cost is independent of whether H owns a copy or not and thus, we ignore such requests when deciding whether to replicate. Hence, the only two significant accesses are reads at the node wanting to join the RS and writes initiated anywhere in the RS (and thus routed through node E). Henceforth, `r` and `w` stand for these respectively.

These tokens are the same for the shrink-RS case. The difference here is that only the fringe node wanting to exit the RS is involved unlike two nodes in the grow-RS case. So, the significant accesses are reads from itself and writes generated elsewhere in the replication scheme.

7.3 Grow-RS Problem

This section describes the learning subsystem for growing the replication scheme. Fig 4 shows a typical FSA of height 4. The transitions at the start state S are used to trap the signature of the read-dominant sequence; the state S_1 can be reached only when the prefix signature of two reads is sampled. In the tree of Fig 4, the size is eight (k=8). Such a tree is maintained by every node for each of its neighbors for each object. The next subsection explains how the system uses such a DFSA to make predictions.

7.3.1 Predicting *Read-Dominant* sequences

Let the tree in Fig 4 be the tree node E maintains for node H to decide whether to replicate to it. Consider the state S_3 in the tree, reached on tracking the sequence `rrw`. The *score* field is +2. It reflects the difference between the number of 1's (read-dominant sequences) and 0's (write-dominant sequences) in the history_register. What that means is that the previous k (eight, in this tree) times, that a sequence beginning with `rrw` (two consecutive reads from H followed by a write in the RS) was encountered, the number of *read-dominant* sequences that followed were two more than the *write-dominant* sequences. The *history_register* records them chronologically– the more recent in the right– five read-dominant and three write-dominant. So based on previous history,

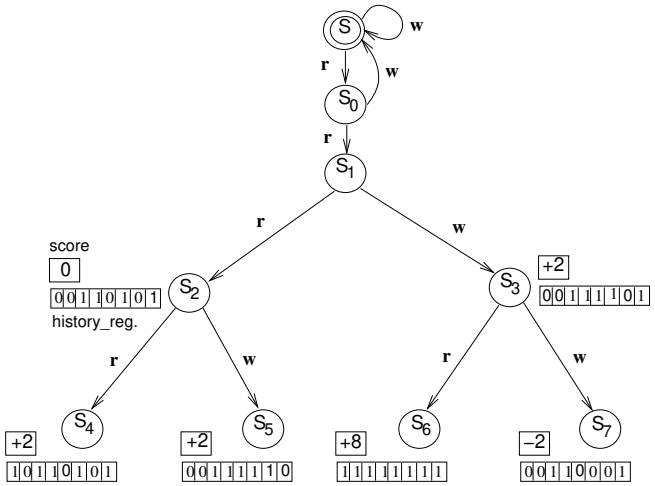

Figure 4: A sample FSA

the sequence which would follow is very likely read-dominant. State S_7 has a negative score. Hence, it is very likely that a sequence beginning with `rrww` is write-dominant and hence, if this state is reached the object should not be replicated. Consider the sequence `rrr` leading to state S_2. The previous 8 times that some application running on H caused three successive reads to the object, the sequences that followed were evenly read and write dominant. At this stage, node E has to process (at least) one more access before it can make a decision. We now clarify some terminology about FSAs. Analogous to trees, the states of a FSA are classified as leaf states (e.g., S_4, S_5) and non-leaf states (e.g., S_2, S_3). Also, the top three states (S, S_0 and S_1) are called *non-valid* states because they contain no information and are used just to trap the signature. The other states ($S_2, S_3 \ldots$) are *valid* states.

7.3.2 Algorithm to grow the RS

In this section we informally describe the algorithm to grow the RS. How the FSA structure is generated and maintained is described in the next subsection. Rather than speak of the algorithm in general terms, we use the specific example of node E having to decide if node H should join the RS (Fig 1). Since all cases of growing the RS are similar, this may easily be generalized. As explained before, the only accesses significant to replicating the object to H are reads from H and writes in the RS routed to E. The following algorithm is executed by all nodes like E, which have neighbors not in the replication scheme.

At the start, Loc, the location pointer, points to the start state S.

1. E polls for next access to the object (either from H or from the RS).
2. If it is a non-significant request (a write from H or a read at E), ignore and Goto 1.
3. If it is a read request from H or a write from the RS, update Loc to reflect the new state. If it was a write, Goto 1.
4. If the new state is a non-valid state (i.e., S, S_0 or S_1), Goto 1.
5. If it is a valid state, there are two choices.

- The new state is not a leaf node. There are two cases here.

- The score is positive, i.e., replicate. E then tags a message to this effect with the copy of the object. H saves the copy and thus, joins the replication scheme. Once this happens, the FSA is left untouched until H returns the feedback information about the correctness of E's prediction. The next section details this process of feedback. (Case α)
- The score field is negative (i.e., by past history, likely write-dominant). E just sends a copy of the object with no additional message. Node H thus flushes the copy, after the transaction is over. Goto 1.(Case β)

- The new state is a leaf. This implies that the score field in all the states in the path from the root to this leaf were negative and thus the learning system predicts a write-dominant sequence. E just sends a copy of the object with no additional message. However, E verifies the correctness of its write-dominant prediction by determining the actual of the sequence. Accordingly, an appropriate feedback is given.(Case γ)

There is some additional work that needs to be done in case of a write in step 3. Consider node E. Let it receive a write request from H. As explained before, this request is ignored by E when deciding whether to replicate to H. However, this request influences E's decision to replicate to J (and all such neighbors without a copy) because if J joins the RS every write from H and other RS nodes is going to cost more. A write from a neighbor already having a copy (e.g., G) affects the decision to replicate to *all* non-RS neighbors equally. In some sense, while a read is limited to E and the node from which the read arose, a write involves all neighbors of E which do not have a copy. So in case of a write, Step 3 is as follows:

- If the write is from a neighbor in the RS (e.g.,G), update Loc by making a `w` transition for the FSA's of all the nodes without a copy (H and J).

- If the write is from a neighbor not in the RS, update Loc for the FSA's of all the nodes without a copy except the node from which the request arose. This is because a write from the node is not significant to its joining the RS.

Both fringe and center nodes can be involved in growing the RS because both can have neighbors not in the RS.

7.3.3 Maintaining the FSA

This subsection is about maintaining a FSA by incorporating the feedback. Cases α and γ from Sec. 7.3.2 the two cases which require update of the FSA structure. There are two basic problems– one, to figure out the actual label of the sequence- read or write-dominant and thus, verifying the correctness of the prediction, and two, identifying the path from the root to the leaf which requires feedback.

Updating an FSA with the feedback information is very simple. Assume that the label of the sequence and the update path is known. Starting from the root, on every state along the path the following step is performed. Depending on the label– read-dominant or write-dominant, a 1 or a 0 is left shifted into the history_register respectively. According to what value is shifted out, the score field is updated to reflect the new difference between 1's and 0's. By incorporating the feedback information, the FSA now encapsulates the most recent access pattern of the node for the object.

Consider Case α where the RS grew to include a new node (call this node n-node, H in the last example). The n-node has to supply the actual label of the sequence to the neighbor storing the FSA (call it o-node, node E). Also, since the replication took place when Loc was at a non-leaf state, the n-node has to supply the first few access sequence so that a path from the current state to a leaf can be traced giving a complete path from the root to a leaf. Note that, since the n-node is now in the RS, reads at it are no longer routed to the o-node and hence the need for the first few accesses. The n-node determines the label of the sequence as follows. Once it joins the RS, it scans the next epoch_size accesses arriving there, keeping a running total of the number of reads and writes to the object. Based on which of reads or writes were greater, the sequence is determined to be read or write-dominant respectively. The n-node also stores the first few accesses to the object. At the next write after epoch_size accesses, the n-node tags to the update message for o-node the following data – the actual label of the sequence and the first few accesses to the object in the n-node. Using these accesses, the o-node makes state transitions, until Loc reaches a leaf node at which point a complete path from the root to a leaf is available. By doing this, the o-node has the complete prefix window of this access sequence. This path is, then, updated with the actual label of the sequence.

The feedback for Case γ is simpler than Case α. First, the Loc is already at a leaf state and thus, the path is known. Second, since the object was never replicated, the o-node itself determines the actual label of the sequence after epoch_size accesses (by keeping a total of the reads and writes). After epoch_size requests, if the label is read-dominant, its prediction was wrong or else, it was correct. Either way, the DFSA is updated by left shifting the appropriate bit into the history_register and simultaneously updating the score along the path from the root to the leaf.

In α, there is a special case. If the object is flushed at the n-node before epoch_size access, then it was a bad decision to replicate in the first place. In this case a negative feedback is given to the o-node during the delete protocol (of Sec 8). Alternatively, if the n-node replicates the object further before epoch_size accesses, the sequence is deemed read-dominant.

7.4 Shrinking the RS

This is the problem of deciding when to delete objects in order to shrink the size of the replication scheme. The issues here are analogous to the grow-RS case. However, there are some differences which we highlight in this section. First, every node decides its own fate in the replication scheme. Hence, it uses and maintains its own FSA to predict write-dominant sequences unlike in the grow-RS case where each node does so for its neighbor. Second, depending on whether the node is a fringe node or a center node, exit from the RS is handled differently. In the former case, the object is marked *Dead* and deleted. In the latter case, it is marked *Coma* but not necessarily deleted immediately.

The fringe node case is similar to the grow-RS case. Analogous to it, every fringe node polls the accesses for a write-dominant sequence. It does this by traversing the object's FSA at every request and checking on the score field. If the score is negative, implying a possible write-dominant sequence, it deletes its copy (using the Delete protocol of Sec 8) and exits the RS (case α). If the score is positive (case β) or Loc, the location pointer reaches a leaf state (case γ) then the sequence to follow is very

likely read-dominant and thus the node continues in the RS. However, it requires feedback for cases α and γ. For α, this feedback is given by the node's neighbor in the RS after epoch_size accesses. For, γ, the node itself can determine the actual label of the sequence by tabulating the reads and writes. The FSA is then updated to reflect the new information. Similar to the grow-RS case, there are two quirks which need to be taken care of. One, if the object is replicated back to the node again before epoch_size requests or, two, if the neighbor in the RS exits the RS too, before epoch_size requests. The first one is a case of wrong prediction whereas the latter is deemed a correct prediction.

Notice that flushing an object can leaves a free buffer in the node. While there is an option to replace it with another object, this is avoided because of the lack of information for a good replacement. This buffer would be used up by the next new object replicated to the node.

Now consider the case in which the node is in the center of the RS for some object. If the object hasn't been read-accessed recently at the node, it is very unlikely to be accessed in the future. However, as explained in Sec 5, cost of a write to this object is the same independent of its presence or absence in this (center) node. So, rather than delete the object, in the absence of a read for epoch_size accesses, at the next write it is marked *Coma*. *Coma* objects are first to be replaced when space is required for new objects. Every node maintains an LRU queue for *Coma* objects. Every *Coma* object is automatically enqueued into the LRU queue. The object at the head of the queue is the least recently read-accessed object. An object is dequeued, for the following reasons. One, the node no longer is a center node in the object's RS (become a fringe node). Two, when the node requires buffer space, the object is at the head of the queue is deleted and its buffer used instead. Three, every time a *Coma* object is read-accessed, it become *Live*, and, it is dequeued.

7.5 Replacing Objects

In our model space is limited in each node. Consequently the problem of replacement arises whenever a new object is replicated to the node. The following protocol is followed to obtain buffer space for the new object:

- If there are free buffers in the node, then such a buffer is used by the new object.
- If there are no buffers in the free list, then the LRU list maintained for *Coma* objects is used. The new object replaces the object at the head of the LQU queue.
- If the LRU queue is empty, then an object is randomly selected and flushed. This occurrence of this case is rare because the possibility of more than the memory-size number of objects being *Live* simultaneously is very unlikely.

7.6 Startup Configuration

As yet we have not talked about the startup time distribution of data in the network. The initial distribution of objects to nodes can either be random (i.e., arbitrarily decide the number of copies of an object and place them randomly to nodes) or using one of the static replication techniques . The latter is not only an expensive procedure but also requires a priori knowledge of accesses. Besides, in very dynamic environment such schemes are not effective anyway. So, we chose a random startup configuration. The

values of the data structures in the learning system should be as follows. For the FSAs used to grow the replication scheme, the score field in all states are given a slightly positive value, where as in the trees used to shrink the replication scheme, the score field is given a slightly negative value (e.g., 2 and -2, the history_registers initialized accordingly). This would imply that, for example, every sequence with two successive reads would be a read-dominant sequence to begin with. Given, a quiescent time, the learning system will stabilize to reflect the true characteristic of the environment.

8 Object Deletion Algorithm

This section describes the protocol which has to be followed when a node decides to delete a copy of the object. The crux of algorithm is to ensure that the object being deleted is not the last copy in the network. Deletion of an object can cause the replication scheme to decrease by one edge (e.g., G exits the RS in Fig 1) or by many edges (e.g., in Fig 1, if B exits, the RS reduces to just three node).

The deletion algorithm is simple. When a node $\mathcal{P}$ decides to delete an object, it first checks to see if it holds the only copy in the network. If so, it aborts the deletion. If the node is in the *center* of the replication scheme for the object being deleted, then the node does so without informing any other node. This is because there are at least two other fringe nodes which hold a copy. The tough case is when the node is a fringe node. Exit of a fringe node, like B in Fig 1, can sometimes cause a huge change in the size of the replication scheme. Along the way, all nodes affected by the exit (e.g., C and D in Fig 1 will no longer be in the RS, if B exits) need to update their Direction Vectors for the object. The fringe node sends a message to its neighbor requesting permission to delete. This message is recursively transmitted until it reaches, say, node $\mathcal{Q}$.

If $\mathcal{Q}$ does not own a copy then:

- If it has two or more neighbors in the replication scheme than the one it got the message from, it allows deletion. (case α)
- If it has one other neighbor in the replication scheme, it re-transmits this message onto that neighbor.(case β).

If $\mathcal{Q}$ owns a copy, then:

- If it has one or more other neighbors in the replication scheme beside the one it got the message from, it allows deletion.(case γ).
- If it has no other neighbor in the replication scheme, it is a node with the only other copy in the network and now becomes the node with the last copy. Node $\mathcal{P}$ is allowed to delete.(case δ).

For cases α, γ and δ, all the nodes in the path from $\mathcal{P}$ to $\mathcal{Q}$ reset their *Direction Vectors* to reflect the deletion in node $\mathcal{P}$. This algorithm automatically detects the last copy in the network, when the second-last copy is being deleted (case δ). If both the nodes having the last two copies of the network decide to delete simultaneously, some arbitrary tie-breaking scheme could be used.

A very unlikely case requiring attention is when all the objects in a node are last copies in the network and consequently, cannot be deleted. The problem arises if a new object is replicated to the node but the node has no space for it since no existing object can be replaced. To take care of this eventuality, each node leaves a few *emergency* buffers free for such a

situation. While the transaction requiring the new object is being processed, the node randomly chooses one of the other (last copy) objects and *migrates* it to another node. The space of this migrated object is then inserted back to the emergency pool. This ensures that the transaction does not block for the time required to migrate the object.

9 Algorithm Analysis

9.1 Space Analysis and Optimizations

We now analyze the space overhead of our learning scheme. We consider the space occupied by each FSA. Let H denote the height of the tree and k the size of the *history_register*. Once these parameters are fixed, the structure of the tree gets fixed too. So, the tree might be stored as an array and nodes could be accessed by indexing into the array. This would considerably save space of pointers. Considering the worst case scenario in terms of space,

- Space per node (in bits):S=k (*history_register*) + ($\lceil log_2(\mathsf{k}) \rceil$ + 1) (*score* field + sign bit)
- Space per FSA = $(2^H - 1)$S + H (*Loc*)

For $H = 4$ (sampling subsequence of 5) and k= 16, space per FSA is about 40 bytes. This is the space per object per neighbor. There are two FSA's per neighbor, one to grow the RS and the other too shrink it. So if a node had two neighbors, the overhead would be 160 bytes per object. Another structure requiring space is the LRU queue for *Coma* objects. This will require another four to five bytes per object. This space overhead is reasonable for any medium to large sized object. If the objects are smaller sized, say on an average 200 bytes, then some space optimizations might be performed. For e.g., the *score* field is superfluous, its value can be gleaned from the *history_register*. That would save nearly ten bytes per FSA in the last example. Besides, some compression technique might also be used to save space. This would be at the expense of extra computation, but that would not matter since message transmission time would dominate the computation time anyway. In any case, the variable parameters of the FSA – its height and the size of the history_register can be chosen accordingly.

9.2 Message Complexity

The only component of the algorithm which requires messages not directly related to satisfying read-write requests is the Delete protocol. It can be shown that the cost of the Delete algorithm in terms of messages is twice the number of edges in the replication scheme. The cost is amortized over deletes at the center and the fringe nodes – the former costs nothing, the latter accounts for the total cost. So even though in the worst case, deletion of an object at a fringe node might cause messages to be recursively transmitted, since deletion at center nodes cost nothing, the total cost is amortized to linear in the number of replicas. This is no more than what would be required for schemes where the replication scheme is contiguous without holes [WoJ92a] and thus, detecting the last copy is considerably easier. Being integrated, the algorithm has no superfluous messages except those satisfying read-write requests.

10 Simulation results

Preliminary simulation studies were performed to analyze the performance of our algorithm. Unfortunately, we had two major drawbacks in our simulation. One was the absence of any other algorithm to compare against since to the best of our

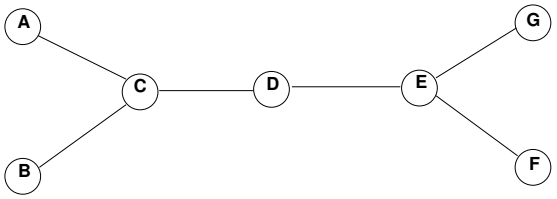

Figure 5: The simulation network

Figure 6: The simulation access pattern

knowledge, this is the only distributed algorithm for a restricted-memory tree-network model. Secondly, publicly available real access traces are mainly for read-shared systems and do not have a good mix of accesses which are both read and write shared. Consequently, the objective of the simulation study was not a comparison test but a preliminary evaluation of the various design decisions. The test sequences had varying percentages of reads and writes and are used to illustrate the working of the algorithm.

As has been pointed out, we are assuming that objects are independent in the sense that there is no structure in the data that we can exploit to prefetch objects optimistically. Instead all information regarding expected access patterns is determined solely from the objects own access history. Consequently, we scaled down the magnitude of simulation parameters by a small factor to make computation feasible and still achieve our objective. The 7-node distributed network of Fig 5. was the basis of our simulation. The total number of objects in the network was 10,000. The nodes C,D and E had a maximum memory of 8000 (80% of total) objects and nodes A,B,F and G have a 4000-object (40 %) memory size.

All access patterns presented were of the form in Fig 6. Each subsequence L_i (also called a cycle) was related to its previous subsequence L_{i-1} in some fashion which will be explained as we go along. Each access had 16 such subsequences. The height of the learning FSA was 5. The start up time distribution of objects to nodes was done randomly. It was ensured that there was always one copy of every object some where in the network.

Consider Fig 7. In this experiment, an access sequence of the form in Fig 6 was fed to the algorithm. Each subsequence L_{i+1} was generated by introducing some θ noise in the previous subsequence L_i. θ was varied from 0% to 100%. So for the case where 20% noise was introduced, each L_{i+1} had 20% of its tokens different from L_i. Each subsequence L_i was of length 40000, i.e., four times the total number of objects. The figure shows the number of messages traversed in the system for every such subsequence. The downward slope of the curve implies that with successive input of each subsequence, the system utilized lesser messages to satisfy requests until such time that it could learn nothing further and thus settle down to a fixed level. The best performance is obviously in the case where the same sequence was repeated over and over again (0% noise). The algorithm was able to reorder data placement and reduce the message traffic by more than 50% of the startup value. With increasing noise, the performance degrades and it does the worst for 100% noise, where each successive subsequence is completely different. In all the cases, the curves flatten out to best performance the algorithm can produce. For the 0% case, the algorithm can ac-

tually extract a pattern where as in the 100% case it tries the best it can do. Starting from 100% down to 0% sequence, the graphs show the messages saved in able to extract the added pattern in 20% of the tokens. An observation about the cost function has to made at this point. The goal of the algorithm is to minimize the total message traffic. The number of messages traversed for a read is a lot lesser than that in a write since the latter affects all the copies. Consequently, the system tries to optimize the cost of writes at the expense of costs for reads. This comes from the fact that even though the algorithm weighs a read equivalent to a write, the accesses used to determine whether to delete or replicate, are reads from the node (and its neighbors) and writes from *anywhere* in the network. So for example, if a node has to get a copy its read demand should have to be significant in comparison to the write demand in the network. Consequently, a bunch of random writes in the network can rapidly cause a shrinking of the replication scheme in the absence of any regular read pattern. This is what happens as the noise level increases – the replication scheme for all the objects gradually shrink. However, unlike when the access patterns are regular and the replication scheme gradually moves to the centers of read-write activity, in a noisy scenario the replication scheme is volatile in the absence of a focussed center of activity. This is the reason why even in the 100% case, the curve flattens out – the placement of data is not geared towards a particular pattern and consequently, the system performs more or less the same for any patternless sequence. Whenever, a pattern is apparent the algorithm is able to adapt and thus save messages (reflected by the drop in the level to which it settles as the noise levels reduce). In fact, Fig 8 which gives the amount of data movement (replications and deletions) in each subsequence highlights this point. When the pattern is regular, the replication scheme very quickly moves to the center of read-write activity and remains unchanged. On the other hand, for the random sequence, the system has to continually move the replicas in the absence of a focussed center and thus, data movement and message traversal is high. Jaggedness in the curves are absent mainly because of amortization over the length of each subsequence and the resolution of the sampling. A number of such experiments were run and the curves are representative of the behavior. The sensitivity towards writes is particularly pronounced for small networks like the one chosen as the simulation network. This bias would diminish as the network grew in size.

To verify the observation of reads versus writes we presented the system with a doctored sequence where the percentage of reads and write were kept constant over portions of the sequence. This is how it was done. Subsequences $L_0, \ldots L_7$ were generated each having γ% writes followed by $L_8, \ldots L_{15}$ which had γ% reads (i.e., (100-γ)% writes). γ was varied from 0% to 100%. Besides, as before there was 20% noise introduced in successive cycles. We denote each sequence by the percentage of reads in the first half. The two extremes were 100%R – first half (8 cycles) all-read sequence followed with an all-write last half and 0%R – 8 cycles of writes followed by 8-cycles of all read accesses. While such a real life sequence is very unlikely, it helps display the working of the algorithm. Consider Fig 9. This is the graph of the message traffic generated at each subsequence L_i. Notice the sharp peaks after the 8^{th} cycle at which point the composition of the sequence changes from predominantly read to write or vice versa. Consider the two

extreme cases – 100%R and 0%R. The former has a sharp increase in the number of messages traversed after the 8^{th} cycle. This is because, in the first half the replication scheme expands in the face of continuous reads. However, once the sequence becomes predominantly writes the message traffic greatly increases until for about four cycles later when the algorithm learns the new pattern and shrinks the RS suitably. The jump is not as startling for the 0%R case because after the write-intensive first half, the replicas are very close to each other and the message cost in the latter half are mainly for reads, which require fewer messages. So the bias of the system in favor of writes even at the expense of increasing read cost pays off in such a case. This bias is also apparent in Fig 10, which shows the average number of replicas in the system in each cycle. At the end of the 4^{th} cycle, there is a difference of 1.5 between 100%R and 80%R as opposed to just 0.5 between 80%R and 60%R. This shows that the system is very sensitive to the presence of writes.

The last two graphs (Fig 11 and Fig 12) show how objects are distributed in each node during the course of the 100%R and 0%R run respectively. The results here are surprising. Once the sequence becomes predominantly write accesses, none of the nodes are utilized to their full capacity (the nodes at the ears like A are only about 10% full to about 50% in the rest). This could lead to inefficiency if, say, the old fashion caching technique of LRU was used. With a LRU technique, once an object is brought into a node it is held until such time that it becomes the least recently used object and is given up to bring in a new object (this is under the assumption that updates are propagated to all cached copies as in the ROWA protocol as opposed to invalidate [CFL91]). Even though only a small portion of the space is really necessary, an LRU technique would hang on to memory size number of objects incurring unnecessary messages. However, as in Fig 12, after the eight cycle, the memory at each node gradually gets completely full. An LRU based scheme may possibly perform comparably in such a case. The biggest advantage of our technique is its ability to handle both extremes equally well, adapting automatically by adding or deleting objects at nodes. This figure also shows how the replication scheme gradually grows (the order of nodes becoming full)– first D, then C,E and finally the ear nodes A,B etc. The graphs for the intermediate levels lie between the two extremes. The graphs of Figs 11 and 12 are input specific and their shapes vary depending on where the centers of read-write activity are. For example, if the center of read-write activity was node A, it would be nearly full all the time. Figs 9 and 10, however, are representative of the behavior of the algorithm for sequences which switch the percentage of reads and writes midway.

11 Conclusions and Future Work

In this paper we presented an efficient scheme for dynamic data replication over a distributed network. The objective function was to minimize the number of messages in the network required to read and write objects. The algorithm was completely distributed with each node making decisions solely on information available locally. We used a finite automaton based structure to maintain statistics about reads and writes and used it to learn access patterns of nodes. We analyzed the trade-off between reads and writes to an object and used that information as a basis to dynamically change their location in the network.

Our model assumed a restriction on the memory at each node. Consequently, we had to solve the problems of replacing objects and determining the direction of the closest neighbor containing a given object. These problems do not arise in earlier work where models had unlimited memory in each node. We also presented preliminary simulation results which show how the scheme adapts to changing access patterns.

Our current work involves generalizing this strategy to work for any network topology. The major bottleneck in arbitrary graphs is the routing of messages efficiently through the network. Our approach is to superimpose a virtual tree structure on the graph and then use the algorithm on the virtual tree. The problem of finding a cost-efficient virtual tree is hard (it is equivalent to the steiner tree problem) and we are using approximation techniques published in literature to attack the problem. This work is in progress and beyond the scope of this paper.

In this paper, the objective function involved the reduction of message traffic where reads and writes, though treated differently, were given equal importance. However, as apparent from the simulation results, by the very nature of the cost function, the algorithm is more sensitive to writes. Consequently, trying to optimize the total message traffic may increase other performance related cost functions like response time. However, our scheme can be generalized to solve such performance related problems which have a read-write tradeoff by suitably weighing reads over writes (as in optimizing response time) or vice-versa (optimizing transaction turnaround time).

Dynamic data replication has a promising future in many aspects of distributed systems. With the advent of mobile computing, dynamic and adaptive techniques for data management will come into prominence. Besides, having to handle the duality of both very high speed physical network and very slow wireless communication links and the explosion in the number of users in such a scenario makes intelligent data management a very major issue. We believe ours is an important step in this direction and based on the simulation results, we find our scheme of dynamic replication very promising.

12 References

[AgE90] D. Agarwal and A. El Abbadi, "The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data," *VLDB* (1990).

[AhA89] M. Ahmad and M. H. Ammar, "Performance Characterization of Quorum-Consensus Algorithms for Replicated Data ," *IEEE Trans. of Software Engg.* 15(4):492-495 (April 1989).

[ABM90] R. Alonso, D. Barbara, and H. Garcia Molina, "Data Caching Issues in an information retrieval system," *ACM TODS* 15:3 (1990).

[AlK93] Rafael Alonso and Hank Korth, "Database System Issues in Nomadic Computing," *SIGMOD* (1993).

[BaI92] B. R. Badrinath and T. Imielinski, "Replication and mobility," *2nd IEEE Workshop on Management of Replicated Data* (Nov. 1992).

[BaM90] D. Barbara and H. Garcia Molina, "The case for controlled inconsistency in replicated data," *IEEE Workshop on Management of Replicated Data* (1990).

[BFR92] Y. Bartal, A. Fiat, and Y. Rabani, "Competitive Algorithms for Distributed Data Management," *ACM STOC* (May, 1992).

[CFL91] M. C. Carey, M. J. Franklin, M. Linvy, and E. J. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures," *ACM SIGMOD* (1991).

[DoF82] D. Dowdy and D. Foster, "Comparitive Models of the File Assignment Problem," *Computing Surveys* 14(2) (June 1982).

[Hay92] D. Hayden, "The new age of wireless," *Mobile Office* (May 1992).

[HoU79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addision-Wesley, 1979.

[HuW93] Yixiu Huang and Ouri Wolfson, "A Competitive Dynamic Data Replication Algorithm," *ICDE* (1993).

[ImB93] T. Imielinski and B. R. Badrinath, "Mobile Wireless Computing: Solutions and Challenges in Data Management," *Draft* (1993).

[JaM90] S. Jajodia and D. Mutcher, "Dynamic Voting Algorithms for Maintaing the Consistency of a Replicated Database," *ACM TODS* 15(2) (June 1990).

[KrB91] N. Krishnakumar and Arthur Bernstein, "Bounded Ignorance in Replicated Systems," *ACM PODS* (1991).

[KrV91] P. Krishnan and J.S. Vitter, "Optimal Prefetching via Data Compression," *IEEE FOCS* (Oct 1991).

[KRS93] A. Kumar, M. Rabinovich, and R. K. Sinha, "A Performance Study of General Grid Structures for Replicated Data," *ICDCS* (1993).

[LLS88] R. Ladin, B. Liskov, and L. Shrira, "A technique for constructing highly available distributed services," *Algorithmica* 3 (1988).

[PaZ91] M. Palmer and S. Zdonik, "Fido: A Cache That Learns to Fetch," *VLDB* (1991).

[PaL88] J. F. Paris and D.E. Long, "Efficient Dynamic Voting Algorithms," *IEEE Intl. Conf. of Data Engineering* (1988).

[PuL91] Calton Pu and Avraham Leff, "Replica control in Distributed System: An asynchronous approach," *ACM SIGMOD* (1991).

[Sal91] K. Salem, "Adaptive Prefetching for Disk Buffers," CESDIS, Goddard Space Flight Center, TR-91-64, Jan, 1991.

[ShR90] A. Sheth and M. Rusinkiewicz, "Management of interdependent data:Specifying dependency and consistency requirements," *IEEE Workshop on Management of Replicated Data* (1990).

[WoJ92a] O. Wolfson and S. Jajodia, "Distributed Algorithms for dynamic replication of data," *ACM PODS* (1992).

[WoJ92b] Ouri Wolfson and Sushil Jajodia, "An Algorithm for dynamic data distribution," *2nd IEEE Workshop on Management of replicated data* (Nov 1992).

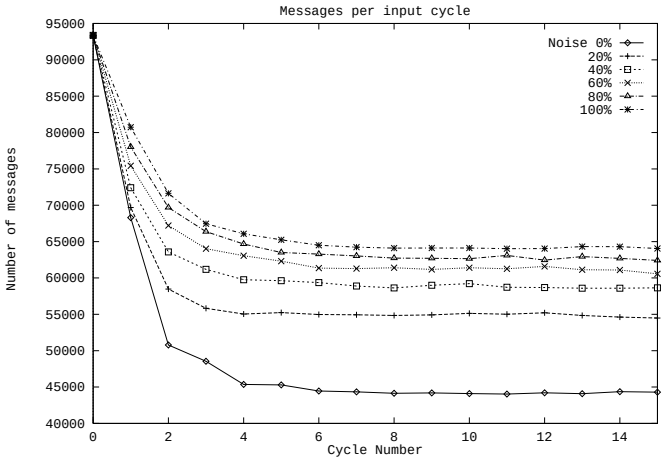

Figure 7: Messages traversed per subsequence

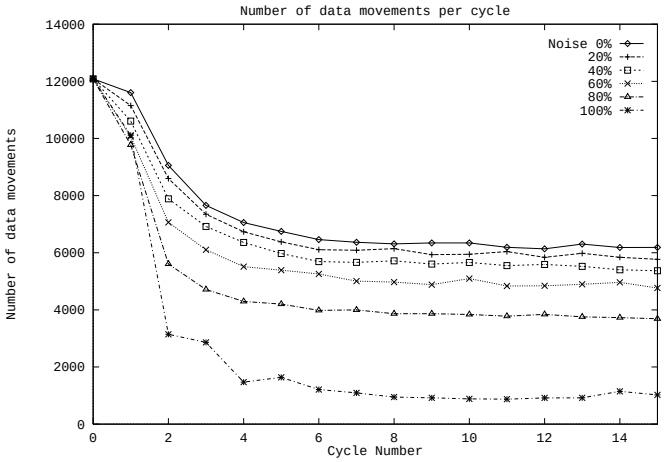

Figure 8: Number of data movements per subsequence

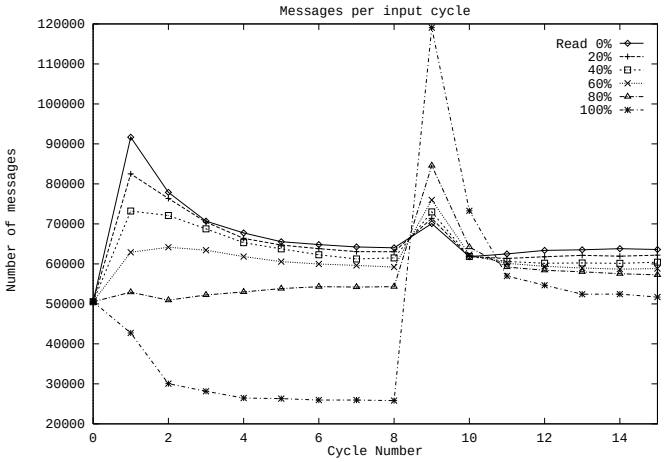

Figure 9: Messages traversed per subsequence. The keys denote the percentage of reads in the first half of the sequence.

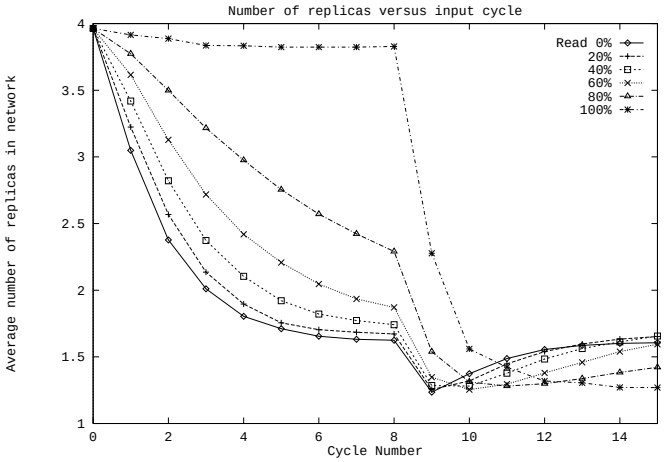

Figure 10: Average number of replicas in the system

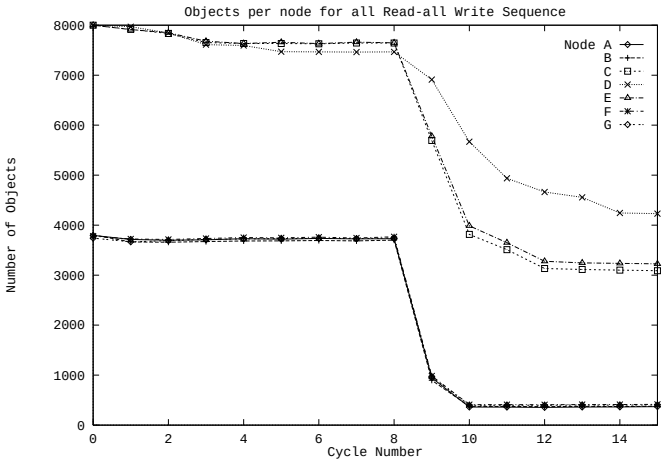

Figure 11: Number of objects per node for the 100%R-100%W sequence.

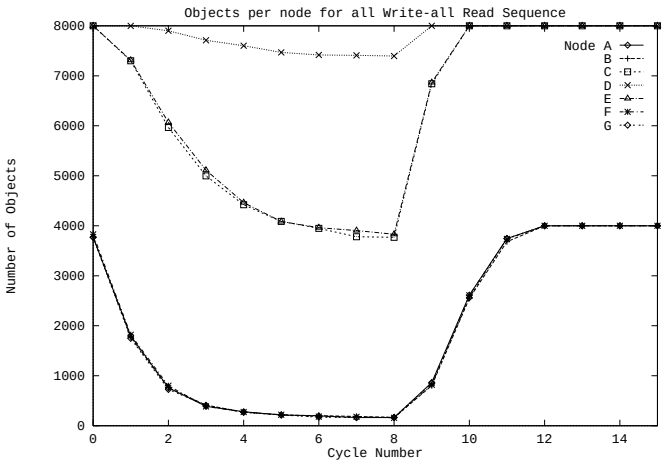

Figure 12: Number of objects per node for the 0%R-0%W sequence.