An Adaptive Data Replication Algorithm

Ouri Wolfson
University of Illinois, Chicago and NASA/CESDIS Goddard Space Flight Center
and
Sushil Jajodia
George Mason University, Fairfax
and
Yixiu Huang
University of Illinois, Chicago

This paper addresses the performance of distributed database systems. Specifically, we present an algorithm for dynamic replication of an object in distributed systems. The algorithm is adaptive in the sense that it changes the replication scheme of the object (i.e. the set of processors at which the object is replicated), as changes occur in the read-write pattern of the object (i.e. the number of reads and writes issued by each processor). The algorithm continuously moves the replication scheme towards an optimal one. We show that the algorithm can be combined with the concurrency control and recovery mechanisms of a distributed database management system. The performance of the algorithm is analyzed theoretically and experimentally. On the way we provide a lower bound on the performance of any dynamic replication algorithm.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: System—*Transaction Processing*

General Terms: Database Management Systems

Additional Key Words and Phrases: Database Management Systems, Transaction Processing, Concurrency Control

1. INTRODUCTION

1.1 Motivation

The internet and the world-wide-web are are rapidly moving us towards a distributed, wholly interconnected information environment. In this environment an object will be accessed, i.e. read and written, from multiple locations that may

This research was supported in part by NSF grants IRI-90-03341, IRI 92244605, IRI-9408750 and AFOSR grants F49620-93-1-0059, 90-0135 and ARPA grant, administered by the Office of Naval Research grant number N0014-92-J-4038.

Authors' Address: O. Wolfson, Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607; NASA/CESDIS, Goddard Space Flight Center, Code 930.5, Greenbelt, MD 20771; Email: wolfson@eecs.uic.edu.
S. Jajodia, ISSE Department, MS 4A4, George Mason University, 4400 University Drive, Fairfax, VA 22030-4444; Phone: 993-1653; Email: jajodia@gmu.edu.
Y. Huang, Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607.

be geographically distributed world-wide. For example, in electronic publishing a document (e.g. a newspaper, an article, a book) may be coauthored and read by many users, in a distributed fashion. Financial-instruments' prices will be read and updated from all over the world. An image, e.g. an X-ray, will be read and annotated by many hospitals. Raw data (on a scientific experiment for example) will be used and modified by many laboratories. In mobile computing and communication environments of the future (see [Badrinath and Imielinski 1992; Imielinski and Badrinath 1992]) an identification will be associated with a user, rather than with a physical location, as is the case today. The location of the user will be updated as a result of the user's mobility, and it will be read on behalf of the callers.

In such environments, the replication of objects in the distributed system has crucial implications for the system performance. For example, consider an object O. Its replication scheme is the set of processors at which O is replicated. In world-wide-web terminology the replication scheme is the set of servers of O. Thus, the replication scheme determines how many replicas of O are created, and to which processors these replicas are allocated. This scheme affects the performance of the distributed system, since reading O locally is faster and less costly than reading it from a remote processor. Therefore, in a read-intensive network a widely distributed replication of O is mandated in order to increase the number of local reads and to decrease the load on a central server. On the other hand, an update of an object is usually written to all, or a majority of the replicas. In this case, a wide distribution slows down each write, and increases its communication cost. Therefore, in a write-intensive network a narrowly distributed replication is mandated.

In general, the optimal replication scheme of an object depends on the read-write pattern, i.e., the number of reads and writes issued by each processor.

Presently, the replication scheme of a distributed database is established in a static fashion, when the database is designed. The replication scheme remains fixed until the designer manually intervenes to change the number of replicas or their location. If the read-write patterns are fixed and are known a priori, then this is a reasonable solution. However, if the read-write patterns change dynamically, in unpredictable ways, a static replication scheme may lead to severe performance problems.

1.2 Dynamic object allocation

In this paper we propose and analyze a dynamic replication algorithm called ADAPTIVE DATA REPLICATION (ADR). The ADR algorithm changes the replication scheme of an object dynamically, as the read-write pattern of the object changes in the network. The changes in the read-write pattern may not be known a priori.

The algorithm is **distributed** as opposed to centralized. In a centralized algorithm, each processor periodically transmits the relevant information (usually statistics) to some predetermined processor, x. In turn, x computes the objective function and orders the change of replication scheme. In contrast, in a distributed algorithm, each processor makes decisions to locally change the replication scheme, and it does so based on statistics collected locally. An example of a local change to the replication scheme is the following: a processor, x, relinquishes its replica (by indicating to x's neighbors that writes of the object should not be propagated to x). This change may result from a comparison of the number of reads to the number of writes, at x (i.e. locally collected statistics).

Distributed algorithms have two advantages over centralized ones. First, they respond to changes in the read-write pattern in a more timely manner, since they avoid the delay involved in the collection of statistics, computation, and decision broadcast. Second, their overhead is lower because they eliminate the extra messages required in the centralized case.

The ADR algorithm works in the read-one-write-all context (see [Bernstein et al. 1987; Ceri and Pelagatti 1984; Ozsu and Valduriez 1991]), and may be combined with two-phase-locking or another concurrency control algorithm in order to ensure one-copy-serializability (see [Bernstein et al. 1987]). Read-one-write-all implies that writes cannot execute when there is a failure in the system. To allow some writes even in the face of failures, we propose in this paper a new protocol, Primary Missing Writes, that can be combined with dynamic replication.

An earlier version of the ADR algorithm (called CAR and first announced in [Wolfson and Jajodia 1992]) was proposed for managing a distributed database consisting of location-dependent objects in mobile computing [Badrinath and Imielinski 1992; Badrinath et al. 1992; Imielinski and Badrinath 1992]. In this database each object represents the location of a user. It is read by the callers of the user, and it is updated when the user changes its location. When a user is relatively static and is called frequently, then her location should be widely distributed throughout the network. When a user moves frequently and is called infrequently, then her location should be narrowly distributed, i.e. have a small number of copies.

We first introduce and analyze the ADR algorithm for a network having a logical or physical tree structure. Later we extend the algorithm to work for a network modeled by a general graph topology. The extension superimposes a dynamically changing tree structure on the network; it also revises the processing of reads and writes to take advantage of shorter paths that may be available in the network but not in the tree.

1.3 Analysis of the ADR algorithm

The ADR algorithm changes the replication scheme to decrease its communication cost. The communication cost of a replication scheme is the average number of inter-processor messages required for a read or a write of the object. Optimizing the communication cost objective function reduces the load on the communication network and the processors' CPU cost in processing the messages.

The problem of finding an optimal replication scheme, i.e. a replication scheme that has minimum cost for a given read-write pattern, has been shown to be NP-complete (see [Wolfson and Milo 1991]) for general graph toplogies even in the centralized case. Thus we first define and analyze the ADR algorithm for tree networks. The analysis is theoretical and experimental.

Theoretically, we show that the ADR algorithm is **convergent-optimal** in the following sense. Assume that the read-write pattern of each processor is generally regular. For example, during the first two hours processor 2 executes three reads and one write per second, processor 1 executes five reads and two writes per second, etc.; during the next four hour period processor 2 executes one read and one write per second, processor 1 executes two reads and two writes, and so on. Then, we show that the ADR algorithm will converge to the optimal replication scheme for the global read-write pattern during the first two hours, then it will converge to the optimal replication scheme for the global read-write pattern during the next four hours, etc. In other words, starting at any replication scheme, the ADR algorithm converges to the replication scheme that is optimal for the current read-write pattern; this convergence occurs within a number of time-periods that is bounded by the diameter of the network. In order to prove this we introduce a new model for analyzing adaptive replication algorithms.

Experimentally, we compare the performance of the ADR algorithm to the performance of static replication schemes for various randomly generated read-write patterns. We show that the communication cost of the ADR algorithm is on average between 21% and 50% lower than that of a static replication algorithm. The exact figure depends on whether or not the read-write pattern is fixed and known a priori, which in turn determines whether an optimal static replication scheme can be selected.

Finally, in order to put the proposed algorithm in the proper perspective, we devise a theoretical lower bound on the communication cost function. The lower bound is the cost of the ideal algorithm that has complete knowledge of all the future read-write requests, and their order. Obviously this algorithm is unrealistic, and it is used only as a yardstick. Then we show experimentally that on average, the communication cost of the ADR algorithm is 63% higher than that of the lower bound.

For general graph network topologies we show that under regularity assumptions the ADR algorithm improves the communication cost in each time period, until it converges, i.e. the replication scheme stabilizes. However, in contrast to the tree case, this replication scheme may not be optimal. In other words, starting with an initial replication scheme, the ADR algorithm will change it to reduce the communication cost until it reaches a local (rather than global) optimum. In this sense, the ADR algorithm on general graph topologies is **convergent** rather than convergent-optimal.

1.4 Paper organization

The rest of the paper is organized as follows. In section 2 we present and demonstrate the ADAPTIVE-DATA-REPLICATION algorithm. In section 3 we discuss various practical issues related to the implementation of the ADR algorithm in distributed systems. For example, we discuss storage space considerations in digital libraries, incorporation of ADR in various replica consistency protocols (e.g. two-phase-locking), discrepancy between the read-write unit and the data replication unit, object-orientation issues such as methods and complex objects, and a way of incorporating a priori information about the read-write activity in the ADR algorithm. In section 4 we introduce the Primary-Missing-Writes algorithm that handles failure and recovery in a dynamic replication environment. In section 5 we introduce a model for analyzing dynamic replication algorithms, and we prove that the ADR algorithm is convergent-optimal in the sense explained above. In section 6 we experimentally compare the performance of the ADR algorithm with that of static replication algorithms. In section 7 we devise a lower bound off-line algorithm for dynamic replication, and we experimentally evaluate the ADR algorithm using the off-line lower bound algorithm as a yardstick. In section 8 we extend the

ADR algorithm to general network topologies, and we prove that it is convergent. In section 9 we compare our work to relevant literature, and finally, in section 10 we summarize the results.

In appendix A we prove of one of the main theorems of the paper, and in appendix B we provide the pseudo code for the ADR algorithm.

2. THE ADAPTIVE-DATA-REPLICATION ALGORITHM

In this section we present the ADR algorithm which works for a tree network. The tree represents a physical or a logical communication structure. Metaphorically, the replication scheme of the algorithm forms a variable-size amoeba that stays connected at all times, and constantly moves towards the "center of read-write activity". The replication scheme expands as the read activity increases, and it contracts as the write activity increases. Roughly speaking, when at each "border" processor (i.e. processor of the circumference of the amoeba) the number of reads equals the number of writes, the replication scheme remains fixed. Then this scheme is optimal for the read-write pattern in the network.

The ADR algorithm services reads and writes of an object. They are executed as follows. A read of the object is performed from the closest replica in the network. A write updates all the replicas, and it is propagated along the edges of a subtree that contains the writer and the processors of the replication scheme. For example, consider the communication network T of the figure below, and suppose that the replication scheme consists of processors 3, 7 and 8.

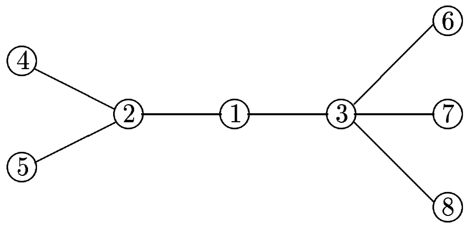

Figure 1 : A computer Network

When processor 2 writes the object, 2 sends the object to 1, then 1 sends it to 3, and then 3 sends it to 7 and 8 simultaneously. Overall, the communication cost of this write is four, namely the total number of inter-processor messages required.

In the ADR algorithm the initial replication scheme consists of a connected [1] set of processors, and at any time, the processors of the replication scheme, denoted R, are connected. For example, the ADR algorithm will never replicate the object only at processors 2 and 3 of the network in Figure 1. Consider a processor i that is an *$\overline{R}$-neighbor*, i.e. i belongs to R but it has a neighbor that does not belong to R. Denote by j a neighbor of i that does not belong to R. j sends read and write requests to i, each of which originates either in j or in some other processor,

[1]A set of processors is connected if it induces a connected subgraph of the tree network

h, such that the shortest path from h to R goes through j. For example, consider the network of Figure 1. If $R = \{3\}$, then processor 1 makes read-write requests that originate in 1, 2, 4 and 5.

Each processor executing the ADR algorithm receives a priori a parameter t denoting the length of a time period (e.g. 5 minutes). Changes to the replication scheme are executed at the end of the time period, by some processors of the replication scheme. The need for changes is determined using three tests, namely the expansion test, the contraction test, and the switch test. The expansion test is executed by each processor that is an $\overline{R}$*-neighbor*. Suppose that R is not a singleton set. Then we define an R*-fringe* processor to be a leaf of the subgraph induced by R. Each R*-fringe* processor executes the contraction test. Observe that a processor can be both an $\overline{R}$*-neighbor* and R*-fringe*. Then it first executes the expansion test, and if it fails, then it executes the contraction test. A processor p of R that does not have any neighbors that are also in R (i.e. R is the singleton set $\{p\}$) executes first the expansion test, and if it fails, then it executes the switch test.

Now we are ready to formally define the tests. At the end of each time period, each processor i that is an $\overline{R}$*-neighbor* performs the following test.

(Expansion-Test). For each neighbor j that is not in R compare two integers denoted x and y. x is the number of reads that i received from j during the last time period; y is the total number of writes that i received in the last time period from i itself, or from a neighbor different than j. If $x > y$, then i sends to j a copy of the object with an indication to save the copy in its local database. Thus, j joins R.

Practically, the actual expansion from i to j could be delayed until i receives the next read request from j, or i receives the next write to be propagated to j. The indication to join R can be thus piggybacked on the copy of the object sent from i to j in order to service the request.

Except for i and j, no other processor is informed of the expansion of R. The expansion test is performed by comparing the counters (one for the reads and the other for the writes). The counters are initialized to zero at the end of each time period, and incremented during the following time period. The expansion test *succeeds* if the (if) condition is satisfied for at least one neighbor. We say that the expansion test *fails* if it does not succeed.

Consider a processor i that is an R*-fringe* processor, i.e. it is in R and has exactly one neighbor j that is in R. Processor i executes the following test at the end of each time period.

(Contraction-Test). Compare two integers denoted x and y. x is the number of writes that i received from j during the last time period; y is the number of reads that i received in the last time period (the read requests received by i are made by i itself or received from a neighbor of i different than j). If $x > y$, then i requests permission from j to exit R, i.e. to cease keeping a copy.

Processor i does not exit unconditionally, since i and j may be the only processors of the current replication scheme, and they may both announce their exit to each other, leaving an empty replication scheme. Therefore, if the Contraction-Test succeeds, then i keeps its replica until it receives the next message from j. If this message is j's request to leave R, then only one (say the one with the smaller processor-identification-number) leaves R.

Otherwise, i.e. if the exit request is granted, processor j will not to propagate any more write requests to i; any further read requests arriving at i are passed along to j. Except for i and j, no other processor is informed of the contraction.

Finally, suppose that processor i constitutes the whole replication scheme. Then i is an $\overline{R}$-neighbor thus it must execute the expansion test. If the expansion test fails, then i executes the following test at the end of the time period.

(Switch-Test). For each neighbor n compare two integers denoted x and y. x is the number of requests received by i from n during the last time period; y the number of all other requests received by i during the last time period. If $x > y$, then i sends a copy of the object to n with an indication that n becomes the new singleton processor in the replication scheme; and i discards its own copy.

When the (if) condition of the contraction or switch test is satisfied, then we say that the test *succeeds*. Otherwise, we say that it *fails*.

Practically, the singleton replication scheme switch can be delayed until i receives the next write from any processor, or i receives the next read request from n. The switch from i to n means that, simultaneously, i exits from R and n enters R.

We summarize the algorithm as follows. At the end of each time period, an $\overline{R}$-neighbor executes the Expansion-Test. An R-fringe processor executes the Contraction-Test. A processor that is both an $\overline{R}$-neighbor and an R-fringe executes first the expansion test, and if it fails then it executes the contraction test. A processor of R that does not have a neighbor in R executes first the expansion test, and if it fails, then it executes the Switch-Test.

Example 1. In this example we demonstrate the operation of the ADR algorithm. Let's consider the network of Figure 1. Suppose that the initial replication scheme, R, consists of the processor 1. Suppose that each processor, except processor 8, issues 4 reads and 2 writes in each time period; processor 8 issues 20 reads and 12 writes in each time period. Suppose further that the requests are serviced in the time period in which they are issued.

At the end of the first time period processor 1 (as an $\overline{R}$-neighbor of processors 2 and 3) executes the Expansion-test. Since the number of reads requested by processor 2 is 12 and the number of writes requested by processors 1 and 3 is 20, processor 2 does not enter the replication scheme. The number of reads requested by processor 3 is 32 and the number of writes requested by processors 1 and 2 is 8. Thus processor 3 enters the replication scheme as a result of the test, and R becomes $\{1, 3\}$.

At the end of the second time period processors 1 and 3 will execute the following tests. First, processor 1 performs the Expansion-test (towards processor 2), and fails as in the first time period. Second, processor 1 performs the Contraction-test, and succeeds, since processor 1 receives 18 writes from processor 3, and 16 reads from processors 1 and 2. At the same time processor 3 executes the Expansion-test, and successfully includes processor 8 in the replication scheme, since the number of reads from processor 8 is 20, and the number of writes from the other processors is 14. As a result of these tests, the replication scheme becomes $\{3, 8\}$ after the second time period.

Starting from the third time period the replication scheme will stabilize at $\{3, 8\}$, i.e. it will not change further.

3. PRACTICAL ISSUES

In this section we will discuss the ADR algorithm, practical considerations in its incorporation into a distributed database system, and some of its applications.

ADR as a distributed algorithm. Consider the level of global knowledge required by a processor executing the ADR algorithm. Note that the algorithm requires that each processor of R knows whether it is an $\overline{R}$*-neighbor*, or an R-fringe processor, or a unique processor of R. Knowing this requires only that the processor knows the identity of its neighbors, and "remembers" for each neighbor whether or not it is in the replication scheme. A processor that does not belong to the replication scheme does not participate in the algorithm; nor does an internal processor of the replication scheme, i.e. a processor that is not an R-fringe, and all its neighbors have a replica.

What does a processor need to know in order to execute reads and writes? A processor j of the current replication scheme, R, satisfies a read request locally, and transmits each write request to the neighbors that are in the replication scheme (each of which in turn, propagates the write to its neighbors that are in R). Therefore, j has to know the identity of its neighbors, and to remember for each neighbor whether or not it is in the replication scheme (same as the information needed to execute the ADR algorithm). Interestingly, a processor that does not belong to the replication scheme does not have to "search" for the replication scheme in order to execute reads and writes. A processor j that is not in the replication scheme R, must remember the processor i to which j sent the last announcement that j exits R. i indicates the "direction" of R. Each read or write of j must be sent to i, which in turn, if is not in R any longer, routes the request in the "direction" of R. Therefore, for executing the ADR algorithm and for executing reads and writes, knowledge of the whole network topology is not necessary; nor is knowledge of the whole replication scheme necessary.

Connectivity of the replication scheme. Why should the replication scheme be connected under all circumstances? For example, suppose that communication between New York and Los Angles goes through Chicago, and all the accesses to the object originate from either New York or Los Angles. The ADR algorithm will store a copy in Chicago, and this may seem wasteful. The ADR algorithm can be modified to create at the inactive internal processors, e.g. Chicago, only pseudo-replicas. Pseudo replicas do not require real storage space. A processor storing a pseudo replica does not install any write locally, but it propagates each write to its neighbors in the replication scheme, as in the case its replica is real. However, pseudo replicas cannot satisfy reads either. In the ADR algorithm, the only reads satisfied by internal copies are reads originating from the respective internal processor, and at a pseudo replica processor these reads will have to be satisfied from a real replica. For example, if at some point Chicago does issue a read, the read will have to be satisfied from a real replica. Exit requests should also be handled more carefully. For example, if New York and Los Angles both request "exit" from the replication scheme, the pseudo-replica processor at Chicago cannot grant both requests at the same time (unlike the case in which Chicago holds a real replica), otherwise a real replica of the object will become unavailable.

A related subject is the following.

Multiple objects and space limitations. In digital libraries consisting of multiple objects, storage space considerations may play a significant role. In the ADR context, suppose that the expansion or switch test indicate that a processor p should keep a copy of an object, but p doesn't have enough storage space. This problem can be addressed by ensuring that, if p has space for only n objects (we assume for the moment that all objects are of equal size) then it stores the objects that provide the maximum cost benefit. A way to do so is for p to maintain a benefit value, denoted $b(o,p)$, for each object o stored at p. We will define $b(o,p)$ in three cases: (1) the replication scheme R of o is the singleton $\{p\}$, (2) p is an R-fringe processor, and (3) p is not an R-fringe processor.

In the first case $b(o,p) = \infty$ since o cannot be deleted from p. [2] In the second case $b(o,p)$ is the difference between the number of reads serviced by p in the last time period, and the number of writes propagated to p in the last time period. [3] In the third case $b(o,p)$ is the number of reads serviced by p in the last time period (writes are ignored since, as explained in the previous discussion item "Connectivity of the replication scheme", even if o becomes a pseudo replica at p writes will still be propagated to p).

Now, whenever p is ordered by a neighbor q to store an object o, p is also given the value of $b(o,p)$, which is computed by q as follows. If the order is a result of an expansion from q, then $b(o,p)$ is the difference between the number of reads that q received from p, and the number of writes received by q. If the order is a result of a switch from q, then $b(o,p)$ is the difference between the number of read-write requests of o issued by p to q, and the number of all the other read-write requests for o received by q.

Now, when p is ordered to store the object o but p does not have enough storage space, it compares the value of $b(o,p)$ to the minimum benefit value of all the objects that p stores. Denote by o' the object for which this minimum benefit value is achieved. If $b(o,p) \leq b(o',p)$, then the expansion or switch order is denied. Otherwise o replaces o' at p; p is contracted out of the replication scheme of o', or o' becomes a pseudo replica, depending on whether or not p is a fringe processor in the replication scheme of o'.

When different objects have different sizes this method can be adapted by multiplying the benefit of an object by its size, however we will not elaborate on this here.

One copy serializability. In a transaction processing system, the ADR algorithm can be combined with two-phase-locking to ensure one-copy-serializability. Since the ADR algorithm obeys the read-one-write-all policy, this may seem obvi-

[2]Strictly speaking, o can be switched to another processor, but for the sake of simplicity we will not consider here this marginal optimization.

[3]These two numbers represent the amount by which reads will become more expensive and writes will become less expensive, respectively, if the copy of o is deleted from p. The difference between the two numbers is positive, otherwise p would not store a copy of o.

ous. However, care must be taken, since the meaning of "all" changes dynamically, and the replication scheme during one transaction may be different from the replication scheme during another transaction. Therefore it may be possible that a transaction writes to all the copies of the replication scheme while the replication scheme is expanding to a processor i, and the write is not propagated to i.

This problem can be handled as follows. Each processor, j, maintains a directory-record for each object. The record indicates whether or not j is in the replication scheme, R; if it is then the record also indicates which of j's neighbors are in R, and if it is not then the record indicates the direction of R. When j expands, contracts, or switches it executes a replication-scheme-change transaction. Such a transaction modifies the directory record for the object in order to indicate the change.

Now consider a transaction that writes the replica at processor j. It must also read j's directory record in order to determine to which of j's neighbors the write should be propagated. This read of the directory record conflicts with any write of the directory record that is executed by a replication-scheme-change transaction. Therefore a transaction that writes the object conflicts with a replication-scheme-change transaction. Consequently, a concurrency control mechanism that ensures serializability of transactions in a static replication environment will also do so in this dynamic environment.

Other replica consistency protocols. One copy serializability in transaction-oriented systems ensures that all replicas of an object appear consistent at all times. However, dynamic replication using the ADR algorithm can be combined with other replica management protocols, namely protocols that provide weaker consistency guarantees.

Consider for example the lazy-replication protocol proposed in [Ladin et al. 1992]. In lazy-replication, the operations (described at a higher semantic level than reads and writes) interleave correctly (i.e. according to a causal specification) although replicas are not necessarily consistent at all times. The protocol also distinguishes between update operations, that have to be eventually performed by all the replicas, and query operations that are serviced by a single replica. Updates are propagated to all replicas using background gossip messages.

Clearly, the performance of the lazy-replication protocol can also benefit from dynamic replication, i.e. from changing the number of replicas as a direct function of the ratio of queries to updates, and moving the replicas closer to the locations which initiate the operations. Moreover, consider any 2-type application, i.e. an application in which there are two types of operations, one that is serviced by one replica, and another, that is serviced by all the replicas. For example, a transaction processing system ensuring 1-copy-serializability is one such application, and lazy-replication is another. In the former the updates are propagated synchronously, and in the latter they are asynchronous. In [Fischer and Michael 1992] there is a proposal of another 2-type protocol, namely one that maintains a dictionary database using three type of operations. Two update operations, INSERT and DELETE, and a query operation, LIST.

In principle, the performance of any 2-type application can be improved by dynamic replication. The 2-type application may have to be adapted to dynamic replication. The adaptation is straight-forward, as explained above, for transaction processing systems that ensure 1-copy serializability. However, adaptation of lazy-replication is more complicated, since lazy-replication uses a multipart timestamp that has a component for every replica (so it will not work if the number of replicas varies dynamically).

The time period t. Now we discuss a variant of the ADR algorithm in which the the length of the time period t changes with the rate of read-write requests. Specifically, consider the variant in which the tests of the ADR algorithm are executed at a processor p every k read-write requests received by p. For example, the expansion-test would be executed when p (assuming it is an $\overline{R}$-*neighbor* processor) receives k read-write requests. The effect of this change would be to execute the replication-scheme changes more frequently when the read-write request load is heavy, and less frequently when the load decreases. Furthermore, since the request frequency may differ in different parts of the network, the tests of the ADR algorithm will be executed more frequently in some parts of the network than in others, and consequently, replication scheme adaptation will be faster in parts of the network that execute more requests.

What will be the effect of an expansion of the replication scheme from p to q under the request-based version of the ADR algorithm? The answer is the following. Consider the set of processors P that issued the k requests based on which p made its expansion decision. Suppose that these requests were issued during t seconds. Suppose further that in the t seconds following the expansion each processor q in the set P issues the same number of reads (writes) as the number of reads that q issued in the t seconds before the expansion. In other words, the read-write-pattern A (see subsection 4.1 for the formal definition of a read-write-pattern) representing the set of requests issued by processors of P is identical before and after the expansion. Then the total communication cost of executing the requests in A is higher before the expansion than after it. In other words, if each processor in the set P issues the same set of requests in the t-second periods before and after the expansion, then the total cost of these requests will be lower after the expansion.

Furthermore, for any two replication schemes that differ only in the fact that in one p has expanded to q, A has a lower cost in the expanded scheme. Thus, assuming that A remains the read-write pattern, the expansion from p to q will reduce cost, regardless of other changes that will occur in the replication scheme. Moreover, since the expansion clearly does not affect the communication cost of requests issued by any other processor, then the effect of the expansion is to reduce the overall cost of servicing read-write requests. A similar claim holds for a replication scheme change resulting from a contraction or switch test. The problem with a formal analysis of the so revised ADR algorithm is that it is hard to prove a global property of the form of theorem 3. The reason for this is that the replication scheme changes (expansion, contraction, switch) would occur in a totally asynchronous fashion, even if the clocks were synchronous.

Discrepancy between the read-write unit and the replication unit. The ADR algorithm assumes that the read-write unit is identical to the replication unit. This is often the case when data is replicated in logical units (e.g. a text file) but not when data is allocated in physical units. For example, suppose that the replication unit is a block, i.e., data is replicated in full blocks. Furthermore, suppose that as a result of a read, the unit transferred between two processors is a set of tuples.

The ADR algorithm can be adapted to handle this situation as follows. Each $\overline{R}$-

neighbor i in the replication scheme of a block b has two counters for each neighbor j that is not in the replication scheme. These are the write-counter and the read-counter of j. Whenever j reads some tuples from b, j's read-counter increases by a fraction that is the ratio of the total size of the tuples read, to the size of b. For example, suppose that the read issued by processor j retrieves 20 tuples from block b, and these tuples constitute 1/10th of the size of b. Then the read-counter for j increases by 1/10th of its value. Similarly, whenever i writes some tuples in b, j's write-counter increases by a fraction that represents the total size of the tuples written. Other than that nothing changes in the ADR algorithm. At the end of the time period the expansion test compares the read-counter and the write-counter, and if the former is larger then j is given a copy of b, thus joining the replication scheme. In other words, the only difference in the ADR algorithm is that the counters may be incremented by a fraction rather than an integer.

Methods. The ADR algorithm can be generalized to systems in which processes do not issue reads and writes to objects, but invoke methods that operate on these objects. The generalization can work as follows. Each method invoked by processor j on object O consists of both a read and a write. The data returned by the method is "read from the object" by j, and the parameters passed by j to the method are "written to the object" by j. The read and write counters are incremented accordingly.

The tree topology. The ADR algorithm works on a tree-network. The tree may represent a logical (in addition to physical) interconnection. An example of a logical tree network (possibly, over a physical network of arbitrary topology) is a management hierarchy, in which some of the processors in the communication network are designated as managers, and they form a hierarchy. Net-mate, a network management system that we are currently developing (see [Dupuy et al. 1991a; Dupuy et al. 1991b; Sengupta et al. 1990; Wolfson et al. 1991]), employs such a management hierarchy. Net-mate's purpose is to provide software tools for the detection of faults and recovery from them in very large communication networks. In Net-mate, an object may be transferred from a manager to its superior, or from the superior to the manager, and its read-write pattern varies over time. For example, consider an object that stores the identification of the overloaded processors. During distributed diagnosis of a global problem in the network, this object is read extensively, whereas at other times it is mainly written. Other examples of logical interconnections represented by a tree can be found in [Goodman 1991; Imielinski and Badrinath 1992].

Complex-object distribution. Observe that adaptive replication can also be used for determining complex-object distribution. Complex-object distribution is the problem of establishing how a complex-object is partitioned/replicated in a computer network, i.e., where each subobject is replicated. This problem seems to be important in many applications, e.g., computer supported collaborative work (see [Grudin 1991]). The ADR algorithm can be used for dynamic complex-object distribution as follows. A complex-object, o, includes all its subobjects, and a read or a write of o is treated by the adaptive replication algorithm as a read or write of all its subobjects. Additionally, each subobject is read and written individually, i.e., it has its own read-write pattern. Thus, when using the ADR algorithm, the distribution of o in the network is determined automatically, depending on the read-write pattern of o and on the read-write pattern of each one of its subobjects.

Incorporating a priori information about read-write activity. As it stands, the ADR algorithm does not use any a priori information about read-write activity. It changes the replication scheme to improve cost under the assumption that the activity in the last time period is indicative of the expected activity in the next time period.

When a priori information is available, i.e. when it is known that processor 1 issues two reads and one write per time period, processor 2 issues five reads and one write per time period, processor 3 issues four reads and four writes per time period, etc. one can use a static replication scheme to optimize communication cost. The static optimal replication scheme can be found using a linear time algorithm provided in [Wolfson and Milo 1991].

However, sometimes we may want to combine a priori read-write information with dynamic allocation. For example, suppose that the a priori information is uncertain, and the read-write activity in a time period is a random variable that has the a priori value with probability x, and has a value identical to the latest time period with probability $1-x$. The ADR algorithm can be adapted to incorporate this type of uncertain a priori information by "adjusting" the number of reads and writes compared in each test of the algorithm. For example, in the expansion test of processor i, the number of read requests from j, ER, is taken to be $AR \cdot x + LR \cdot (1-x)$; where AR is the number of read requests that are received from j if all the processors issued their prespecified number of reads, and LR is the number of reads received from j in the last time period. Similarly, the number of writes from other processors is computed as $EW = AW \cdot x + LW \cdot (1-x)$; where AW is the number of write requests that are received from i or a processor other than j if all the processors issued their prespecified number of writes, and LW is the number of writes received from i or a processor other than j in the last time period. The expansion test succeeds if $ER > EW$.

4. FAILURE AND RECOVERY

In this section we discuss a method by which the ADR algorithm can handle failures. Actually, there may be quite a few methods of handling failures, depending on the level of consistency required among the replicas, and depending on the assumptions about the network topology. Specifically, a method that deals with failures in a 1-copy-serializable transaction-oriented environment may be too restrictive when replicas are allowed to diverge from the most up to date version.

Similarly, a method that works for a physical tree-network may be too restrictive for a logical tree network (see section 3) superimposed on the physical network. The reason is that failure in a physical tree implies network partition, whereas in a logical tree network it does not necessarily do so. In the latter case, suppose that a processor p of the logical tree fails. Denote one of its neighbors by q. If all the neighbors of p can communicate, then a tree can be reconstructed by logically connecting to q all the neighbors of p, except q.

The method of this section deals with the most restrictive case, namely a 1-copy-serializable transaction-oriented system on a physical tree-network. Assume that the ADR algorithm is incorporated in a transaction processing system using a concurrency control mechanism such as two-phase-locking. As with any other read-one-write-all scheme, in case of a failure that prevents a writing processor from reaching all replicas, the writing transaction cannot commit. For a fixed number of replicas there is a protocol called Missing Writes ([Bernstein et al. 1987]) that enables at least some of the writes to be committed, while 1-copy-serializability is ensured for any type of failure, including network partition.

The Missing Writes protocol uses read-one-write-all during the normal mode of operation, and it switches to Quorum Consensus (see [Gifford 1979; Thomas 1979]) when a failure is detected. However, the Missing Writes protocol uses a priori knowledge of the total number of copies in order to determine which partition has a majority of the copies; this will be the only partition that is allowed to write the object. In a static replication environment this works, but when the number of copies varies dynamically, a partition cannot determine what constitutes a majority of the copies. Hence, the Missing Writes protocol does not work for dynamic replication. We have devised a protocol called the Primary Missing Write (PMW) that solves this problem by substituting the primary copy protocol for the quorum consensus.

4.1 Overview of the PMW Protocol

Generally speaking, the PMW protocol works as follows. At any point in time a unique processor of the replication scheme is designated to be the *primary processor*. In the normal (non-failure) mode of operation PMW uses the ADR algorithm. When the primary processor exits from the replication scheme of an object, it assigns the primary role to the processor from which it requests the exit. In case of a switch, the primary role is also switched.

Now consider the failure mode. We assume that the failures are clean, namely they can be detected, and a failed processor will not process any request (no Byzantine failures). If a processor or a communication link fails, then the PMW protocol changes the replication scheme of each object to a singleton consisting of the primary processor alone. The replication scheme may differ from object to object. This replication scheme remains fixed until all the failures are repaired; at that time the system switches to normal mode. In failure mode, the transactions in the partition which contains the primary processor access the primary copy of the object. The object is inaccessible by transactions in the other partitions.

A failure is detected in one of the following cases. First, a transaction may not be able to read an object, i.e., reach a processor at which the object is replicated. In this case the transaction has to be aborted, and it is resumed when the failure is repaired. Second, the transaction may not be able to propagate a write of an object to some processor at which the object is replicated. Then we say that a "missing write" is detected. Once a "missing write" is detected, the transaction that issues the write is aborted and requeued for execution.

Each processor has a status bit indicating whether the processor is running in normal or failure mode. Each transaction, at initiation, reads the status bit and runs either in failure mode or normal mode. In normal mode the transaction will use the ADR algorithm for all the objects that it accesses, and in failure mode it will use the primary copy for all objects it accesses. The status bit is regarded as a database item, i.e., it is locked when read or written. The status bit is read by every transaction; it is written by the failure and recovery transactions, discussed next.

4.2 Conversion to failure mode – When and How ?

A partition converts to failure mode when a "missing write" is detected, i.e., if some of the replicas of the written object cannot be reached. The processor that detects the "missing write" (we call it the *pioneer processor*) initiates a *failure-transaction* that performs the following operations. First, it notifies all the processors in the partition that they are in failure mode and they should discard all non-primary replicas of any object. Each such processor p replies with the identification of the objects for which p is a primary. The pioneer processor constructs the *objects-reachable* list, i.e. the list of objects that are reachable and the (single) processor of the partition that stores each object. Finally, the pioneer processor sends the objects-reachable list to all the processors in the partition. The above operations are executed in an atomic transaction. At the end of the failure transaction all the processors in the partition will have converted to failure mode, and each one will know which objects are reachable in the partition and which processor stores each object.

4.3 Recovery and conversion to normal mode

When a processor recovers from a failure it executes the *recovery-transaction* that performs the following operations. First it checks whether or not there exist any failures in the tree network. [4] If there exist some failures, then the recovering processor tells all processors to be in failure mode; otherwise it tells all processors to convert to normal mode. In either case the recovering processor constructs the objects-reachable list as explained in the previous subsection, and sends it to all the reachable processors. (Observe that even if failures still exist in the system, the reachable-objects list has to be updated since the recovery may reconnect disconnected components, and thus it increases the list objects that are reachable in the partition.) This concludes the description of the recovery transaction.

Observe that when the system converts back to normal mode, each object will have a replication scheme consisting of a single node, the primary processor, and adaptive replication using the ADR algorithm will begin.

Recovery from the failure of a communication link is also handled by running the recovery transaction. This transaction is run by one of the processors connected by the recovering link, say the one with a lower processor identification number.

4.4 Reliability constraints

Observe that if the replication scheme consists of a single processor, then failure of that processor results in the object being inaccessible for reads and writes. In order to avoid this situation, the ADR algorithm may be presented with a reliability

[4]In order to check the up/down status of all processors in the network, a processor can simply send a status message to all its neighbors. When a processor i receives such a message from processor j, i will send a status message to all its neighbors (except j) and wait for a certain period of time. If i receives the "up" replies from all its neighbors in response to the status message, then i sends an "up" status message to processor j. Otherwise, namely if i receives a "fail" status message, or it does not receive any message within a given time period, then i sends to j a "fail" status message.

constraint of the form "at any time there must be at least two copies of an object". The ADR algorithm can enforce this constraint by preventing the contraction test from resulting in a singleton-set replication scheme; it is done as follows. If a processor p has only one neighbor q that has a replica (i.e. p is a leaf of the subtree induced by the replication scheme), then p denies any exit request issued by q. This incorporation of reliability constraints in ADR is independent of the replica consistency protocol, i.e. it can be used in transaction oriented one-copy-serializable systems, as well as other replica-management protocols.

5. ANALYSIS OF THE ADR ALGORITHM

In this section we first prove in subsection 5.1 that the ADR algorithm preserves the connectivity of the replication scheme. Then in subsection 5.2 we introduce a formal model for analyzing adaptive replication algorithms, and we prove that when the read-write pattern of an object becomes regular the ADR algorithm converges to the replication scheme that is optimal for the pattern.

5.1 Connectivity of the replication scheme

A *network* is an undirected tree, $T = (V, E)$. The set V represents a set of processors, and each edge in E represents a bidirectional communication link between two processors. We consider an object replicated at some of the processors in the network. The *replication scheme* of the object is the (nonempty) set of processors at which the object is replicated.

THEOREM 1. *Suppose that an object is replicated using the ADR algorithm. If the replication scheme R in one time period is connected, then the replication scheme R' in the immediately following time period is also connected. Furthermore, either R and R' have at least one common processor, or they are adjacent singletons.*

PROOF. From the definition of the ADR algorithm we see that for a processor $i \in R$ only one test can be successfully executed at the end of a time period. If the expansion test succeeds, then the test maps the replication scheme R to $R' = R \cup \{\textit{some neighbors of } i\}$. It is easy to see in this case that R' is a connected scheme, and that R and R' have the common processor i. If a contraction test succeeds, then the resulting scheme is $R' = R \setminus \{i\}$ where i is an R-fringe processor with a single neighbor, say j, of R. Since i and j cannot exit simultaneously, R' is not empty. Therefore R' is connected, and R and R' have a common processor, j. If the switch test succeeds, then the test maps the replication scheme from $R = \{i\}$ to a neighbor $R' = \{n\}$. In this case, R' is a connected singleton scheme adjacent to R. $\square$

The processors in the network issue read and write requests for the object. A set of pairs $A = \{(\#R_i, \#W_i) \mid i \text{ is a processor in the network, and } \#R_i \text{ and } \#W_i \text{ are nonnegative integers}\}$, is called a *read-write-pattern*. Intuitively, $\#R_i$ represents the number of reads issued by processor i and $\#W_i$ represents the number of writes issued by processor i. The replication scheme *associated* with a request is the replication scheme that exists when the request is issued. Observe that in practice it is possible that by the time the request is serviced the replication scheme has changed.

We assume that a communication cost $c(i,j)$ is associated with each edge (i,j) of the network; it represents the cost of the edge-traversal by the object. The costs are symmetric and positive, i.e. for each pair of processors i and j, $c(i,j) = c(j,i) > 0$. The cost r_i of a read issued at processor i is the total cost of edges on the shortest path between i and a processor of the associated replication scheme. Intuitively, the cost of a read is the total cost of the communication links traversed by the object to satisfy the read. Obviously, if i is in the associated replication scheme, then the read cost is zero.

Assume now that processor $i \in V$ issues a write of the object, and let R be the replication scheme associated with the write. The cost w_i of the write is the total cost of edges in the minimum-cost subtree of T that contains the set $\{i\} \cup R$.

Given a replication scheme, R, and a read-write pattern, A, the *replication scheme cost* for A, denoted $cost(R, A)$, is defined as $\sum_{i \in V} \#R_i \cdot r_i + \sum_{i \in V} \#W_i \cdot w_i$. Intuitively, $cost(R, A)$ represents the total cost of messages sent in order to service the requests in the read-write pattern, assuming that R is the replication scheme associated with every request in A. A message is the transmission of the object over one communication link (edge). A replication scheme is *optimal* for a read-write pattern A if it has the minimum (among all the replication schemes) cost for A. Obviously, optimality of the replication scheme implies that the average cost of a request in A is minimum.

The following theorem justifies the fact that the ADR algorithm keeps the replication scheme connected at all times.

THEOREM 2. *Let A be an arbitrary read-write pattern. For every disconnected replication scheme R there is a connected replication scheme R' such that $cost(R', A) \leq cost(R, A)$, and $R \subset R'$.*

PROOF. Suppose that the graph induced by R consists of two separate connected components. We will construct R' as follows. Observe that there must be two processors of R, i and j, such that if we denote the unique path between them by $i, b_1, \ldots, b_k, j$ for $k \geq 1$, then the $b_i^{'}$s do not belong to R. To obtain R', we add to R all the processors on the path between i and j. It is clear that the cost of any read request does not increase by associating with it the scheme R' rather than R. Furthermore, the cost of each read request from a processor b_i decreases to zero. It is also easy to see that the cost of any write w_l for the associated scheme R' is equal to the cost of w_l for the associated scheme R. Thus $cost(R', A) \leq cost(R, A)$.

If the graph induced by R consists of more than two separate connected components, then we can repeat the above process, each time connecting two disconnected components, until we obtain a connected replication scheme of lower or equal cost than that of R. □

5.2 Convergence to optimal replication scheme

Now we will show that when the read-write pattern becomes regular the ADR algorithm moves the replication scheme towards the optimal one, and when reaching it the ADR algorithm stabilizes the replication scheme. This is quite easy to see for simple read-write patterns.

For example, suppose that starting at some point in time, t, all the processors become quiescent (i.e. stop issuing requests), except for one, i; and, suppose that i issues only reads. Then it is clear intuitively that the ADR algorithm will stabilize the replication scheme, and that the stability scheme will include i (i.e. it will be cost-optimal for any read-write pattern consisting only of reads from i). Specifically, if at time t processor i is in the replication scheme R, then as long as it issues read requests and all the other processors are quiescent, the replication scheme will not change. If processor i is not in R, then R will expand towards i until it reaches i, and from then on the ADR algorithm will become stable. Each expansion step will take one time period, and therefore, if we define the *diameter* of a network to be the maximum number of edges on a path, then the convergence to the optimal replication scheme will occur after a number of time periods that is bounded by the diameter of the network.

Now suppose that starting from point in time t on, processor i issues only writes. Then the ADR algorithm will stabilize the replication scheme, and the stability scheme will be optimal for any read-write pattern consisting only of writes from i (i.e. the stability scheme will be the singleton set, $\{i\}$). Specifically, if at time t processor i is in the replication scheme R, then as long as it issues write requests and all the other processors are quiescent, the replication scheme will contract until it consists of the single processor, i. If processor i is not in R, then denote by j the processor of R that is closest to i. R will contract until it consists of the singleton set $\{j\}$, and then it will switch until it consists of the singleton set $\{i\}$. In both cases, convergence to the optimal replication scheme will occur after a number of time periods that is bounded by the diameter of the network.

Now we will show that this convergence property holds for any regular schedule. A *schedule* is a set of time-stamped requests $o_{t_1}^{j_1}, o_{t_2}^{j_2}, \ldots, o_{t_n}^{j_n}$. Each o is a read or write request, j_i is the processor at which the request o originated and t_i is an integer time stamp at which o was issued and serviced (for the purpose of algorithm analysis we assume that requests are executed instantaneously). Two read requests can have the same time stamp, but two write requests, and a read and a write request cannot do so.

Consider an integer t representing the number of time units in a period, and a schedule S. Then we can refer to the requests of S in the first time period, the requests of S in the second time period, etc. For example, suppose that the time-stamp of the first request of S is 0. Then the requests of S in the first time period are all the requests with a time-stamp which is not higher than t; the requests of S in the second time period are all the requests with a time-stamp s, such that $t+1 \leq s \leq 2 \cdot t$, etc. Informally, we say that a schedule is regular if each time period has the same read-write pattern. Formally, a schedule S is *t-regular* if for each processor p there are two integers r_p and w_p, such that in every time period processor p issues r_p read requests and w_p write requests. For a t-regular schedule S we say that $\{(r_p, w_p) \mid p \textit{ is a processor } \}$ is the *read-write pattern of the time period.*

Consider a schedule S whose first time stamp is 0, and consider the execution of the ADR algorithm. Assume that all the processors execute the tests of the ADR algorithm and they change the replication scheme instantaneously. The first time period tests and the replication scheme changes occur after all the requests with time stamp t, and before all the requests with time stamp $(t+1)$; all the second time period tests and the replication scheme changes occur after all the requests with time stamp $2t$ and before all the requests with time stamp $(2t+1)$; etc. This implies in particular that the clocks at the various processors run "approximately" at the same rate, since for all of them expiration of the time period t, although not necessarily simultaneous, always occurs between the same pair of requests in the schedule. We say that the ADR algorithm *stabilizes* in the qth time period if all the tests of all processors fail starting from the qth time period. The replication scheme at time period q is called the *stability scheme*. Recall that in example 1 of section 2 the replication scheme stabilizes in the third time period.

THEOREM 3. *Let d be the diameter of a tree network, and suppose that a schedule S is t-regular, for an integer t. Then, starting with any connected replication scheme, the ADR algorithm with time period t stabilizes on S after at most $(d+1)$ time periods; furthermore, the stability scheme is optimal for the read-write pattern of the period.*

PROOF. See Appendix A. □

6. EXPERIMENTAL ANALYSIS OF THE ADR ALGORITHM

In this section we report on experimental comparison of the performance of the ADR algorithm with that of static replication. In subsection 6.1 we describe how the experiments were conducted. In subsection 6.2 we compare the ADR algorithm with all static replication schemes, using a fixed read-write pattern. The read-write pattern was generated using Poisson processes with parameters that were randomly chosen. In subsection 6.3 we compare the ADR algorithm with each static replication scheme. For each comparison we used a different randomly generated read-write pattern. Whereas subsections 6.1, 6.2 and 6.3 consider the network of figure 1, subsection 6.4 considers other network topologies.

6.1 Experiment preliminaries

For the experimental analysis we used SUN workstations interconnected as a tree network. The communication between two neighboring processors is via stream-socket.

Each experiment counts the number of messages that are used by an algorithm in order to execute a given read-write pattern. The algorithm is either ADR, or static replication with a particular replication scheme. We only consider connected replication schemes (see theorem 2). For the network of figure 1 we consider 8 different replication schemes of size 1; 7 different schemes of size 2; 10 schemes of size 3; 10 schemes of size 4; 11 schemes of size 5; 10 schemes of size 6; 5 schemes of size 7; and only one scheme of size 8.

We assume that each processor p in the tree network generates reads and writes of the object independently of other processors, following a Poisson distribution with parameters λ_r^p, and λ_w^p respectively. Namely, in each time period, the expected number of reads issued by p is λ_r^p, and the expected number of writes issued by p is λ_w^p writes. Furthermore, we assume that the parameters λ_r^p and λ_w^p of each processor p change over time. Namely, p has parameters $\{\lambda_r^{p1}, \lambda_w^{p1}\}$ for the first

L_1 periods of time, and then has parameters $\{\lambda_r^{p_2}, \lambda_w^{p_2}\}$ for the next L_2 periods of time, and so on. The L's are also generated at each processor randomly and independently, and due to independence, the read-write patterns are not regular in the sense of subsection 5.2. On the other hand they are not totally chaotic, since the same λ_r and λ_w persist for more than one time period at each processor.

In our experiments the clocks of the different processors are not synchronized, and a request issued in one time period is not guaranteed to be serviced within the same time period due to communication delays. In other words, the ideal assumptions made in the last section do not hold.

As mentioned, the cost of an algorithm is the total cost of messages. A message is sent between neighbors in the tree, and we distinguish between two types of messages, data messages and control messages. Data messages are messages that carry the object, and control messages are messages that do not do so, i.e. read requests and exit-replication-scheme requests (the latter type is used only in the ADR algorithm, and it is sent by a processor that relinquishes its copy). In the experiments we assume that the costs of all edges in the tree network are identical, i.e. there is no difference between the costs of two data messages or the costs of two control messages.

6.2 A fixed read-write pattern

In this subsection, we consider a read-write pattern that was generated by certain Poisson processes. The parameters of the processes are given in the following table 1. Each column of the table corresponds to a processor, and an entry L_p: λ_r^p–λ_w^p in column p indicates that for L_p time periods processor p generated, in each time period, reads and writes using Poisson processes with parameters λ_r^p and λ_w^p, respectively. For example, column 7 indicates that processor 7 generated, on average, 2 reads and 0 writes per time period for the first 31 periods; then it generated, on average, 15 reads and 5 writes per time period for the next 33 periods, etc. The three values L, λ_r, and λ_w in each entry were generated randomly, such that $\lambda_r > \lambda_w$. Each processor generated requests for 200 time periods. Observe that the read-write pattern of table 1 is not regular, thus the ADR algorithm does not necessarily converge.

①	②	③	④	⑤	⑥	⑦	⑧
47:6—2	17:4—0	29:1—0	16:17—2	66:0—0	48:16—4	31:2—0	57:0—0
71:2—1	47:1—0	70:8—2	67:1—0	45:12—4	77:1—0	33:15—5	22:4—2
33:0—0	49:10—3	53:0—0	79:0—0	89:0—0	53:0—0	33:7—3	30:0—0
49:14—3	87:1—0	48:2—1	38:5—1		22:4—1	50:2—0	63:18—5
						53:0—0	28:1—0

Table 1 : The Fixed Access Pattern

We executed 14 runs of the ADR algorithm using Poisson processes with the above parameters. Each run started with a different initial replication scheme. The total number of requests generated in different runs varied by at most 0.1%. The total cost of different runs varied by at most 2%. Thus the initial replication scheme does not affect the cost of the ADR algorithm significantly.

On average, the ADR algorithm used 5833 data messages and 2005 control messages in order to service a read-write pattern generated by Poisson processes with the above parameters. Using these two numbers, the cost of the ADR algorithm can be computed for any ω, where ω is the ratio of the control message cost to the data message cost.

Then we chose one read-write pattern P of the 14 read-write patterns, and we computed the number of data messages D and the number of control messages C used to service the requests in P, for each static replication scheme. For $0 \leq \omega \leq 1$, using D and C, we computed the cost of servicing the read-write pattern P. For $\omega = 0$, the optimal static replication scheme [5] is {1,2,3,8}; the cost of the ADR algorithm on P is 33.47% lower than {1,2,3,8}'s cost. For $\omega = 1$, the optimal static replication scheme is {1,2,3,6,7,8}; the cost of the ADR algorithm on P is 28.82% lower than {1,2,3,6,7,8}'s cost. For any other value of ω the cost of the ADR algorithm is lower than the cost of the optimal replication scheme by a percentage c, $28.82 < c < 33.47$.

The conclusion from the above discussion is the following. Suppose that it is known in advance that the read-write pattern in table 1 occurs in the network. Then, by using the ADR algorithm the database administrator can save at least 28% compared to the optimal (for the pattern) static replication scheme.

Now assume the read-write pattern of table 1 occurs in the network, but this fact is not known a priori and/or the pattern changes from day to day or week to week. In this case, in the absence of a dynamic replication algorithm, the DBA cannot select an optimal replication scheme, and s/he may choose an arbitrary static scheme. Thus, we compared the performance of the ADR algorithm with such arbitrary static replication scheme, assuming the above read-write pattern. The results of this comparison are illustrated in figure 2. It shows the cost-saving percentage [6] of the ADR algorithm over the static replication schemes, for the case where $\omega = 0$. The dark bars represent the average (over all connected schemes of the same size) savings in communication cost obtained by the ADR algorithm; the lighter bars stand for the standard deviation. For example, compared with all the connected replication schemes consisting of three processors, the ADR algorithm saves an average of 47.3% with a standard deviation of 10.2%. Figure 3 illustrates the communication cost savings for $\omega = 1$. For an intermediate value of ω, namely for $0 < \omega < 1$, the ADR algorithm's average communication cost saving is at least the minimum of the two bar charts.

6.3 Varying read-write patterns

In this subsection we compare the performance of the ADR algorithm with that of each connected static replication scheme. We used a randomly generated read-write pattern for each comparison. More precisely, to compare the ADR algorithm with

[5] See section 5.1 for the definition of an optimal replication scheme.

[6] The cost-saving percentage of algorithm X compared to algorithm Y is (1 - x/y) * 100, where x is the cost of algorithm X and y is the cost of Y.

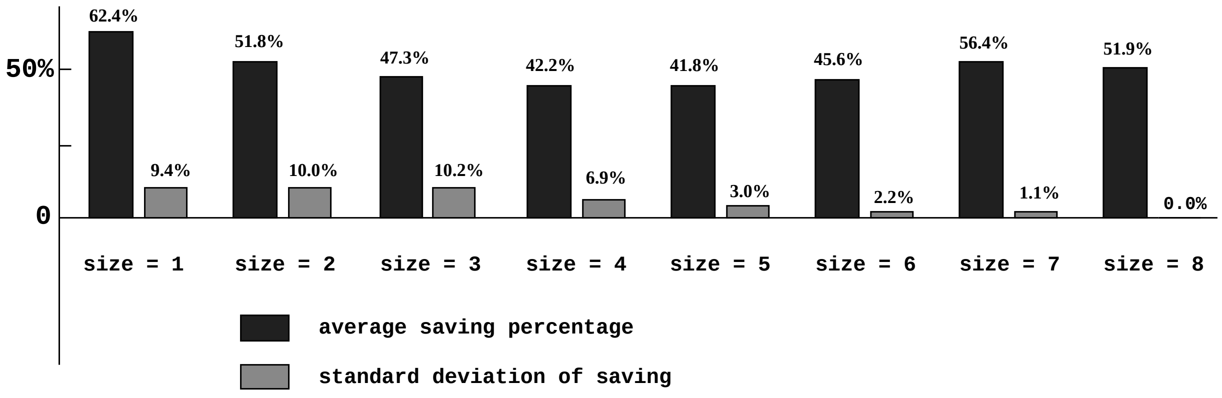

Fig. 2. Average communication cost savings of the ADR algorithm compared with static replication schemes of a given size, for $\omega = 0$

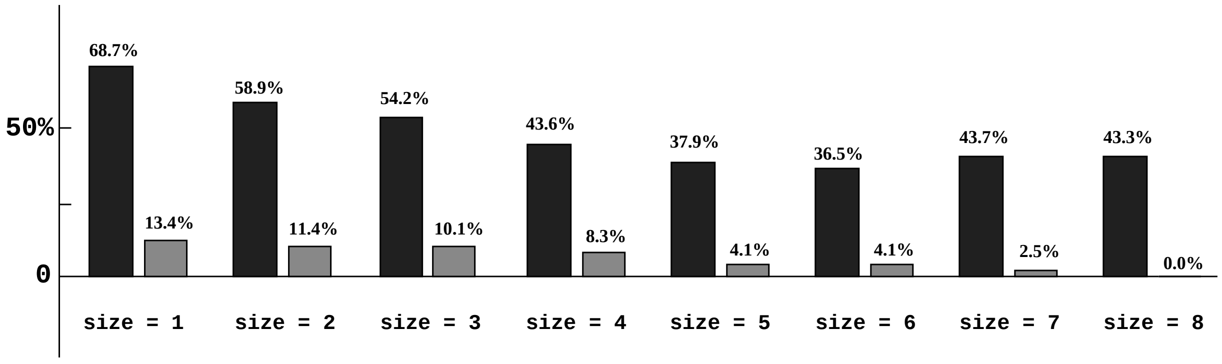

Fig. 3. Average communication cost savings of the ADR algorithm compared with static replication schemes of a given size, for $\omega = 1$

a specific static replication scheme, say Q, we started the ADR algorithm with the initial replication scheme Q. For each Q we executed 8 runs, where a run is a set of read/write requests generated as follows. Each processor p generates the parameters, L_p (which is randomly selected between 1 and 11), λ_r^p (which is randomly selected between 0 and 20) and λ_w^p (which is randomly selected between 0 and λ_r^p). In each time period the processor p issues reads and writes using Poisson processes with parameters λ_r^p and λ_w^p, respectively. After L_p time periods, processor p randomly selects another set of three parameters, L_p λ_r^p and λ_w^p; processor p continues the run with the new set, and so on. Observe that, in contrast to the previous section, a different set of read-write patterns is used for every static replication scheme.

For each run r the cost comparison was executed as follows. During the execution of the ADR algorithm on the run r we recorded all the requests generated from all processors, and we computed the communication cost c by counting the messages. Then we computed the communication cost d of the static replication scheme Q using the same request set, where Q is the initial replication scheme of the ADR algorithm. By comparing c and d we obtained the cost saving of the ADR algorithm for run r. The cost of the ADR algorithm was compared with that of Q using 8 different runs, and we computed the average cost saving (over the 8 runs) obtained by the ADR algorithm. Call this average $a(Q)$.

Then we computed the average of all $a(Q)$'s for all the replication schemes Q of a fixed size. The results of these calculations for $\omega = 0$ are given in figure 4. The results for $\omega = 1$ are given in figure 5.

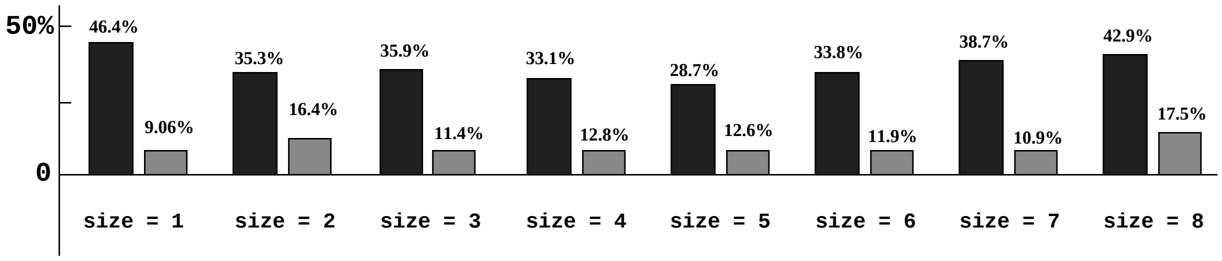

Fig. 4. Average communication cost savings of the ADR algorithm compared with static replication schemes of a given size (varying read-write patterns), for $\omega = 0$.

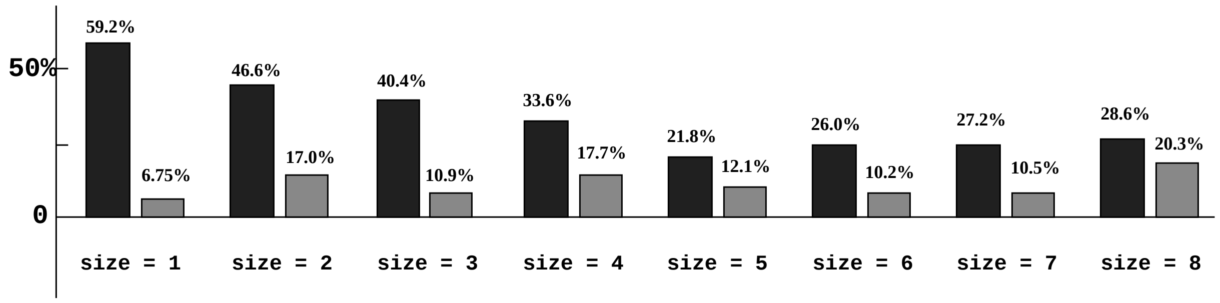

Fig. 5. Average communication cost savings of the ADR algorithm compared with static replication schemes of a given size (varying read-write patterns), for $\omega = 1$.

6.4 Other network topologies

To verify that the ADR algorithm is superior to static replication in other network topologies we ran two additional sets of experiments. In each one of them the network topology was generated in a random fashion, and the initial replication scheme consisted of all the processors. In the first set of experiments we used the read-write pattern of table 1 on network topologies of eight processors. The results of these experiments are summarized in figure 6. The first column of the table in this figure shows the tree, the second column shows the number of messages used by the ADR algorithm, and the third column shows the number of messages used by optimal static replication, i.e. static replication with the optimal (in the sense of subsection 5.1) replication scheme. On average, the cost savings of the ADR algorithm are 27.86% for $\omega = 0$, 24.51% for $\omega = 0.5$, and 21.84% for $\omega = 1$.

Tree Structure	ADR		Optimal Static		
				Messages	
	Data Messages	Control Messages	Replication Scheme	Data	Control
8-3, 8-1, 8-2, 3-4, 2-5, 2-7, 4-6	5234	2527	2, 3, 4, 8	9659	3471
1-4, 4-2, 1-3, 2-5, 3-6, 2-7, 1-8	6680	2665	1, 2, 3, 4	9846	3611
6-1, 6-7, 6-8, 8-5, 1-4, 4-3, 4-2	7157	2100	1, 4, 6, 8	8798	2805
7-2, 2-8, 8-6, 7-1, 8-5, 1-4, 6-3	7780	4168	1, 2, 6, 7, 8	9160	1770
8-3, 8-1, 8-2, 3-4, 8-7, 7-5, 2-6	4722	1781	2, 7, 8	8872	4358
3-6, 6-4, 6-5, 3-7, 3-8, 3-2, 6-1	5905	3435	3, 6	8095	5025
3-6, 6-4, 6-5, 3-7, 6-8, 3-2, 4-1	7852	5194	3, 4, 6	9218	4480
2-5, 2-3, 3-4, 4-6, 5-7, 5-8, 5-1	6025	3054	2, 3, 4, 5	10548	4181
7-2, 2-8, 8-6, 7-1, 1-5, 1-4, 1-3	6320	2552	1, 2, 7, 8	8573	2703
2-5, 2-3, 2-4, 4-6, 6-7, 7-8, 7-1	7119	3732	2, 4,6, 7	9838	3585
2-5, 2-3, 5-4, 4-6, 6-7, 4-8, 8-1	8473	5184	4, 5, 6, 8	10373	4060
7-2, 2-8, 2-6, 7-1, 8-5, 7-4, 8-3	6702	4029	2, 7, 8	8257	3813

Fig. 6. Comparison of the performance of the ADR algorithm versus that of static optimal replication, for various network topologies. The read write pattern that we used is given in table 1.

The second set of experiments is similar to the first one, except that the number of processors in the network, the static replication scheme, and the read write pattern are all randomly generated. In other words, the experiments are similar to the ones described in subsection 6.3, except that the network and the static replication scheme used for comparison are generated in a random fashion. The results of these experiments are summarized in figure 7. On average, the cost savings of the ADR algorithm are 50.58% for $\omega = 0$, 50.06% for $\omega = 0.5$, and 49.40% for $\omega = 1$.

7. A LOWER BOUND FOR DYNAMIC REPLICATION ALGORITHMS

In this section we devise the lower bound on the cost of a given schedule. Then we use the lower bound as a yardstick to experimentally evaluate the performance of the ADR algorithm.

What is the lower bound? The ADR algorithm is "online" in the the sense that after each request the new replication scheme has to be determined, without knowledge of what are the subsequent requests. Furthermore, the new replication scheme is determined in a distributed fashion, and the read-write requests may occur concurrently. In this section we present the optimal algorithm, called LOWER-BOUND (LB). It is "offline" in the sense that its input consists of a schedule given a priori. In other words, the algorithm is not presented with "new" requests. LB is a centralized algorithm. The input of LB is a schedule, and the algorithm LB associates a replication scheme with each request such that the total communication cost is minimum.

Tree Structure	ADR		Static replication scheme		
	Data Messages	Control Messages	Replication Scheme	Messages	
				Data	Control
7-> 2-5, 2-3, 2-4, 4-6, 6-7, 7-1	2460	690	2, 3	6564	4412
8-> 6-1, 6-7, 7-8, 8-5, 7-4, 1-3, 7-2	2342	744	2, 4, 7	6144	3541
7-> 1-4, 4-2, 4-3, 1-5, 2-6, 1-7	2277	652	1, 4, 5	4400	1764
8-> 3-6, 6-4, 6-5, 3-7, 4-8, 3-2, 8-1	2102	905	2, 3, 6, 7	5361	2052
9-> 9-4, 4-1, 9-2, 1-3, 4-5, 1-6, 1-8, 4-7	3132	1864	1, 4, 6	5452	2313
7-> 2-4, 2-3, 4-5, 4-6, 4-7, 5-1	1790	992	1, 4, 5	3357	1061
9-> 2-6, 2-3, 3-4, 6-5, 4-1, 2-8, 2-9, 1-4	2468	907	1, 4	7596	5097
8-> 1-4, 4-2, 4-3, 2-5, 3-6, 1-7, 7-8	2503	1520	1, 3, 4	4714	2411
7-> 3-5, 5-4, 4-6, 4-7, 6-2, 7-1	3142	1121	2, 4, 5, 6	3839	584

Fig. 7. Comparison of the performance of the ADR algorithm versus that of static replication, for various network topologies. The first column shows the number of processors in the network, and the network topology. The read write pattern and the static replication scheme were randomly generated.

Suppose that $S = o_{t_1}^{j_1}, o_{t_2}^{j_2}, \ldots, o_{t_n}^{j_n}$ is a schedule; recall that a pair of requests can have the same time stamp only if both are reads. A *configured-schedule* is a schedule in which each request $o_{t_i}^{j_i}$ is mapped to its associated replication scheme R_i. It is denoted $o_{t_1}^{j_1}(R_1)$, $o_{t_2}^{j_2}(R_2)$, …, $o_{t_n}^{j_n}(R_n)$, and we require that requests with the same time stamp are associated with the same replication scheme (since if the requests are issued simultaneously the replication scheme at that time is unique). An *adaptive replication algorithm* (*ARA*) is a function that maps each schedule to a configured schedule.

For convenience, we assume in this section that a schedule is a sequence of requests (rather than a partial order), in which the requests are given in increasing time stamp order. Two reads that occur simultaneously may appear in any order in the sequence. It will be clear that the results of this section are not affected by this assumption.

7.1 The cost of a configured schedule

Intuitively, the cost of a configured schedule is the cost of all read-write requests, plus the cost of changing the associated replication scheme from one request to the next. Formally, we define the unit $o_i^{j_i}(R_i)$, $o_{i+1}^{j_{i+1}}(R_{i+1})$ in the configured schedule to be a *transition*. The *cost of the transition* is defined as follows. If $o_{i+1}^{j_{i+1}}$ is a write, then the cost of this transition is the cost of the write request $o_{i+1}^{j_{i+1}}$ to the replication scheme R_{i+1} (the cost of a write to a replication scheme is defined in section 5.1). If $o_{i+1}^{j_{i+1}}$ is a read, and $o_i^{j_i}$ is a write, then the cost of the transition is the cost of the read of j_{i+1} from R_{i+1}, plus the minimum cost of writing the object from the processors of R_i to the processors of R_{i+1}. Intuitively, this represents the cost of executing o_{i+1}, plus the cost of moving the replication scheme from R_i to R_{i+1}. If

$o_{i+1}^{j_{i+1}}$ is a read, and $o_i^{j_i}$ is also a read, then the cost of the transition is: the cost of the read of j_{i+1} from R_{i+1}, plus the minimum cost of writing the object from the processors in $R_i \cup \{$all the processors on the shortest path that connects j_i *to* R_i $\}$ to the processors of R_{i+1}. Intuitively, this case is slightly more complicated than the previous one for the following reason. Since the object is read at j_i, j_i and all the the processors between it and R_i can be used to minimize the cost of copying the object from R_i to R_{i+1}.

Finally, the *cost of a configured schedule* is the cost of the first request, plus the cost of all the consecutive transitions of the configured schedule.

For example, consider the tree network of Figure 1 in section 1, and the configured schedule: $r_1^6(\{3,6,7,8\}), r_2^1(\{3,6\}), r_3^2(\{1,3\})$. Its cost, assuming a unit cost for each edge, is the following:

$$\text{[the cost of a read by 6 from the replication scheme } \{3,6,7,8\}] \ (=0)$$
$$+$$
$$\text{[the cost of a read by 1 from the replication scheme } \{3,6\}] \ (=1)$$
$$+$$
$$\text{[the cost of moving from } \{6 \cup \{3,6,7,8\}\}\text{to}\{3,6\}] \ (=0)$$
$$+$$
$$\text{[the cost of a read by 2 from the replication scheme } \{1,3\}] \ (=1)$$
$$+$$
$$\text{[the cost of moving from } \{1 \cup \{3,6\}\}\text{to}\{1,3\}] \ (=0)$$

The first addend represents the cost of the first read. The second and third addends represent the cost of the first transition. The fourth and fifth addends represent the cost of the second transition. Notice that a replication scheme contraction, as in the move from $\{3,6,7,8\}$ to $\{3,6\}$, does not move the object, hence it has zero communication cost.

As another example for the same tree, consider the configured schedule: $r_1^6(\{1,2\})$, $w_2^7(\{1,2,3,6\})$, $r_3^3(\{3,7,8\})$. Its cost is the following:

$$\text{[the cost of a read by 6 from the replication scheme } \{1,2\}] \ (=2)$$
$$+$$
$$\text{[the cost of a write by 7 to the replication scheme } \{1,2,3,6\}] \ (=4)$$
$$+$$
$$\text{[the cost of a read by 3 from the replication scheme } \{3,7,8\}] \ (=0)$$
$$+$$
$$\text{[the cost of moving from } \{1,2,3,6\}\text{to}\{3,7,8\}] \ (=2)$$

The first addend represents the cost of the first read. The second addend represents the cost of the first transition. The third and fourth addends represent the cost of the second transition.

Given a schedule ψ, its *cost-optimal* configured schedule is the one with minimum cost among all configured schedules that have the sequence of requests ψ (but different associated replication schemes).

7.2 The lower bound algorithm

The algorithm LB, defined next, is given as input a schedule ψ. It configures ψ to create a cost-optimal configured schedule.

Intuitively, when using LB, each write is sent to the processors that will read it. In other words, write w is sent to the set of processors that read after w, but before the first write that succeeds w.

Precisely, the algorithm LB works as follows. If the first request of ψ is a read, then LB associates the replication scheme { all processors that read before the first write } with all the reads up to the first write. Whether or not the first request is a read, LB associates with each write, w_i^j, and with each read executed between w_i^j and the first write that succeeds it, the following replication scheme: $\{j \cup$ the processors that read the object between w_i^j and the first write that succeeds $w_i^j\}$. Observe that LB is an adaptive replication algorithm.

For example, consider the tree network in Figure 1, and the schedule: $r_1^1, r_2^6, r_3^3, w_4^3, r_5^6, r_6^2$. The configured schedule devised by LB is as follows. $r_1^1(\{1,3,6\})$, $r_2^6(\{1,3,6\})$, $r_3^3(\{1,3,6\})$, $w_4^3(\{1,2,3,6\})$, $r_5^6(\{1,2,3,6\})$, $r_6^2(\{1,2,3,6\})$. Assuming a unit cost on each edge, the cost of the configured schedule is:

$$\begin{gathered}
[\text{The cost of the first read by 1 from the scheme } \{1,3,6\}]\ (=0) \\
+ \\
[\text{The cost of the first transition }]\ (=0) \\
+ \\
[\text{The cost of the second transition }]\ (=0) \\
+ \\
[\text{The cost of the write by 3 to the scheme } \{1,2,3,6\}]\ (=3) \\
+ \\
[\text{The cost of the fourth transition }]\ (=0) \\
+ \\
[\text{The cost of the fifth transition }]\ (=0)
\end{gathered}$$

Notice that all the read requests in this schedule are local, hence their cost is zero. The only non-zero addend is the third transition. The total cost of the configured schedule is 3, and it is minimum, since processors 2 and 6 have to read the replica written by processor 3; the minimal cost of transmitting the object from processor 3 to processors 2 and 6 is 2+1.

For a schedule ψ and an adaptive replication algorithm A, denote by $c_A(\psi)$ the configured schedule obtained from ψ, by A.

Theorem 4. *For a schedule ψ, $c_{LB}(\psi)$ is cost-optimal.*

Proof. First we will subdivide the schedule ψ in a convenient manner. We denote by π an arbitrary schedule that consists of zero or more read requests. We denote by σ an arbitrary schedule that consists of a write followed by zero or more read requests. Then we can rewrite the schedule ψ as $\psi = \pi_0, \sigma_1, \ldots, \sigma_k$. Also, for an arbitrary ARA, A, we can rewrite the schedule $c_A(\psi)$ as $c_A(\pi_0)$, $c_A(\sigma_1)$, …,

$c_A(\sigma_k)$. Furthermore, the cost of $c_A(\psi)$ is the sum: the cost of $c_A(\pi_0)$ plus the total cost of all $c_A(\sigma_i)'$s for $i = 1, \ldots, k$ (see the definition of the cost of a configured schedule).

Next, consider the cost of the configured schedule devised by LB. Suppose $\pi_0 = r_1^{i_1} \ldots r_n^{i_n}$ $(n \geq 0)$. Denote $R = \{i_1, \ldots, i_n\}$. LB associates the replication scheme R with all the reads in π_0. Thus, the cost of read requests are 0. Furthermore, there is no replication scheme change in $c_{LB}(\pi_0)$. Therefore, the cost of $c_{LB}(\pi_0)$ is 0.

Now consider schedules of the form σ_i. Suppose that $\sigma_i = w_0^{j_0}, r_1^{j_1}, \ldots, r_m^{j_m}$ $(m \geq 0)$. LB associates the replication scheme $\{j_0, j_1, \ldots, j_m\}$ with all the requests in σ_i. Thus, the cost of the reads in σ_i is zero, and also the cost of the transitions is 0; the cost of the write is the cost of edges in T_{σ_i}, where T_{σ_i} is the minimum-cost subtree of the network that spans over $\{j_0, j_1, \ldots, j_m\}$.

Now, for an arbitrary adaptive replication algorithm A, consider $c_A(\sigma_i)$. Since the processors in $\{j_1, \ldots, j_m\}$ read the replica written by j_0, the set of edges traversed by the object as a result of requests in σ_i is a connected subtree of the network that contains the set of processors $\{j_0, j_1, \ldots, j_m\}$. Denote this subtree by Q_{σ_i}. Since T_{σ_i} is the minimum cost subtree that contains $\{j_0, j_1, \ldots, j_m\}$, the cost of the edges in T_{σ_i} is not higher than that in Q_{σ_i}. $\square$

7.3 Experimental comparison of the ADR algorithm vs. lower bound

For each run executed in section 6.3 we recorded the request sequence from each processor. Then we merged the eight sequences (for eight processors) in a random fashion, to create one input to the off-line lower bound algorithm. For each input we computed the off-line lower bound communication cost using the LB algorithm, and compared it to the cost of the run reported in section 6.3. Overall we compared the performance of the ADR and LB algorithms on 496 inputs (runs). From those experiments we concluded that, on the average, for $\omega = 0$ the ADR algorithm incurs 163.6% of the communication cost of the lower bound algorithm; the standard deviation is 8.6%. For $\omega = 1$, the ADR algorithm incurs on average 223% of the communication cost of the lower bound algorithm; the standard deviation is 17%. However, remember that since the LB algorithm knows the future, it does not use any control messages.

8. DYNAMIC ALLOCATION IN GENERAL NETWORKS

In this section we discuss dynamic data allocation in a general-graph network topology. First, let us observe that since the problem of finding the static optimal replication scheme is NP-complete for a network modeled as a general graph (see [Wolfson and Milo 1991]), it is unlikely to find an efficient and convergent-optimal dynamic allocation algorithm. However, one obvious way to extend the ADR algorithm to an arbitrary network is first to find a spanning tree of the network, and then to execute the ADR algorithm on this tree. The drawback of this approach is that the path in the spanning tree between two processors is not necessarily the shortest path between them. Consequently, suppose that a processor i issues a read request for the object, and the request is propagated along the tree edges until it reaches a processor j of the replication scheme. If there is a shorter path between i and j, it would be wasteful for j to send the object to i along the tree path; it should be transmitted along the shortest path. In other words, in contrast to the ADR algorithm, the read request and the propagation of the object in response should proceed along different paths. Furthermore, expanding the replication scheme towards i should mean an expansion from j to k, where k is the first processor that does not have a replica on the shortest path from j to i.

Modifying the ADR algorithm to use the shortest path for a general graph network, as explained above, suggests the following algorithm, called ADR-G. Basically, this algorithm still consists of the three tests of the ADR algorithm. At any point in time there is a spanning tree of the network, in which the replication scheme induces a connected subtree. However, the tree structure changes dynamically over time, as explained next.

A changing spanning tree. The edges of the current tree are divided into two subsets, edges of the subtree induced by the replication scheme, and the rest. Edges of the subtree induced by the replication scheme are not directed, in the sense that a processor that has a replica does not have a tree-father. These edges are used to propagate writes received by a processor of the replication scheme to all the other processors of the scheme. It is important that a subtree is defined to connect the processors of the replication scheme in order to prevent a write from being propagated to a processor from two different neighbors, thus increasing the communication cost. This may be the case if processors of the replication scheme are interconnected by a graph that has cycles, and, as in ADR, a write is propagated by a processor p to all its neighbors, except the one from which it received the write.

The rest of the tree edges are directed edges. They are used by a processor k outside the replication scheme to propagate read and write requests to the replication scheme. k has a unique tree-father, and zero or more tree-sons. The tree-father of k is the first processor on the shortest path in the graph to the replication scheme, as it is currently known to k. As the replication scheme changes, and the changes become known to k through servicing of k's read requests (see below for details), k's tree-father changes. This is the sense in which the tree structure changes dynamically.

When a processor p joins the replication scheme as a result of the expansion test, the directed edge between p and its neighbor in the replication scheme becomes undirected. When a processor p relinquishes its replica as a result of the contraction test, p's tree-father becomes its single neighbor in the replication scheme. When the replica switches from i to n, n becomes i's tree-father.

Processing of read and write requests. A read request issued by a processor i that does not have a replica is sent to i's tree-father, and is propagated along the edges of the current tree until it reaches a processor j of the replication scheme. In response, the object is propagated along the shortest path in the graph as follows. When a processor of the replication scheme (j in this case) receives a request to send the object to i, j first determines if there is a shortest path to i that goes through a tree-neighbor k that has a replica. If so, j propagates the request to k. [7] Otherwise, i.e. if there is no shortest path that goes through a tree-neighbor

[7]In other words, j "prefers" to propagate the request to another processor of the replication scheme which is closer to i, rather than service the request by sending the object. Intuitively, the reason for this is that it is cheaper to propagate the request rather than the object; also, this

that has a replica, j *services* the read-request, i.e. j sends the object to i through some shortest path in the graph. Notice that, since there may be more than one shortest path between two processors, the path-choice for j in either one of the two alternatives above is nondeterministic. Suppose that given the same set of possible neighbors through which to propagate a read request (or the object), a processor always chooses the same processor. Then we say that read-processing is *deterministic.* For simplicity we will assume that read-processing is deterministic.

The *tree-father* of a processor p that does not have a replica is defined to be the last graph-neighbor from which p received the object. In other words, when p receives the object from its graph-neighbor q, the tree-father of p becomes q (since the object was sent via the shortest path from the replication scheme, thus q is the first processor on the shortest path from p to the replication scheme).

A write from a processor that does not have a replica is propagated to the replication scheme along the tree-father path. When a write is received by a processor k of the replication scheme it is processed as in ADR, i.e., it is propagated to all of k's tree-neighbors except the one from which k received the write.

The ADR-G algorithm. At any point in time a processor knows its neighbors in the tree, the general network topology, [8] and a processor of the replication scheme is also aware which of its tree neighbors have a replica. At the outset, a connected replication scheme and spanning tree of the network are selected.

The expansion, contraction and switch tests of the ADR-G algorithm are similar to those of the ADR algorithm. However, some generalizations of the tests are necessary. For example, in a general network, due to the fact that a processor is not cognizant of the shortest path to the replication scheme, it is possible for a processor of the replication scheme to receive a read request from some neighbor i and respond to it by sending the object to another neighbor j. In this case, for the purpose of replication scheme expansion, what matters is the direction in which the object is sent (since it represents the current shortest-path information), rather than the direction from which the request is received (which represents outdated shortest path information). Therefore, the language of the tests has to be modified to account for this subtlety. This language change is actually a generalization, and the revised tests hold for a tree network as well.

An *$\overline{R}$-neighbor* is a processor of the replication scheme that has a graph- (rather than tree-) neighbor that does not have a replica. The expansion test is executed by an *$\overline{R}$-neighbor* for each graph-neighbor that does not have a a replica.

(Expansion-Test). For each neighbor j that is not in R compare two integers denoted x and y. x is the number of times i sent the object to j (to service read requests) during the last time period; y is the total number of write requests issued by i or received by i from a neighbor different than j during the last time period. If $x > y$, then i sends to j a copy of the object with an indication to save the copy in its local database. Thus, j joins R.

Suppose that at the end of a time period processor i is an *R-fringe* processor,

simplifies the replication-scheme change tests.

[8]Actually, instead of the whole network topology, the processor only needs to know how to propagate a message to each of the other processors through a shortest path, i.e. to which of its graph-neighbors to transmit a message destined for a processor x.

i.e. it has exactly one graph-neighbor j that is in R. Then i executes the following contraction test in one of two cases: (1) i is not an $\overline{R}$*-neighbor*, or (2) i is an $\overline{R}$*-neighbor* and its expansion test failed.

(Contraction-Test). Compare two integers denoted x and y. x is the number of writes that i received from j during the last time period; y is the number of reads in the last time period that either have been issued by i itself or that resulted in i sending the object to some neighbor. If $x > y$, then i requests permission from j to exit R, i.e. to cease keeping a copy.

Suppose that processor i constitutes the whole replication scheme. If the expansion test fails, then i executes the following test at the end of the time period.

(Switch-Test). For each neighbor n compare two integers denoted x and y. x is the number of times i sends or receives the object to/from n during the last time period; y the number of times i sends or receives the object to/from a neighbor different than n during the last time period. If $x > y$, then i sends a copy of the object to n with an indication that n becomes the new singleton processor in the replication scheme; and i discards its own copy.

Example 2. In this example we demonstrate the operation of the ADR-G algorithm. We will consider again the network of Figure 1, but we will add to the graph the set of edges $\{(1,8),(5,8),(2,6)\}$ (see figure 8).

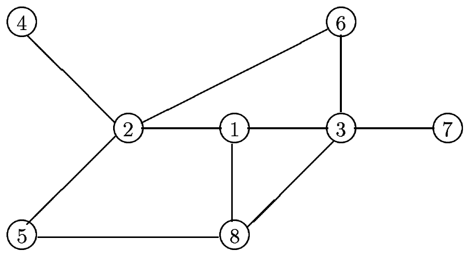

Fig. 8. A general-graph network

As in example 1, we suppose the following. First, the initial replication scheme R is $\{1\}$. Second, we suppose that each processor, except processor 8, issues 4 reads and 2 writes in each time period; processor 8 issues 20 reads and 12 writes in each time period. Third, we suppose that the requests are serviced in the time period in which they are issued. Additionally, we assume that the cost of each edge is one, and that initially requests and responses from processor 6 are routed through processor 3 (note that 6, 2, 1 is also a shortest path between 6 and 1). In other words, the tree-father of 6 is 3, and the tree-father of 3 is 1. Also, requests and responses from processor 5 are routed through processor 2.

At the end of the first time period processor 1 (as an $\overline{R}$-neighbor of processors 2, 3, and 8) executes the Expansion-test. Since the number of reads requested by processor 2 is 12 and the number of writes requested by processors 1, 3, and 8 is 20, processor 2 does not enter the replication scheme. The number of reads requested by processor 3 is 12, and the number of writes requested by processors 1, 2 and 8 is 20. Thus, processor 3 does not enter the replication scheme as a result of the test. The number of reads requested by processor 8 is 20, and the number of writes requested by processors 1, 2 , and 3 is 14. Thus processor 8 enters the replication scheme, which now becomes $\{1, 8\}$.

Assume that the first request issued by processor 5 during the second time period is a read. This read is received by 1 and is propagated to 8, since 1, 8, 5 is a shortest path from 1 to 5 that goes through a processor that has a replica, namely 8. 8 replies by sending the object directly to 5. Thus the tree-father of 5 changes from 2 to 8.

At the end of the second time period processors 1 and 8 will execute the following tests. Processor 1 performs the Expansion-test (towards processors 2 and 3), and it fails. Then processor 1 performs the Contraction-test, and it fails since the processor 1 receives 14 writes from processor 8, and 24 reads from processors 1, 2 and 3. At the same time processor 8 executes the Expansion-test towards processor 5 and it fails. The contraction test executed by processor 8 fails since 8 receives 12 write requests from processor 1, and 24 read requests from processors 5 and 8. Thus, starting from the second time period the replication scheme will stabilize on $\{1, 8\}$.

The total communication cost per time period is 80 for the replication scheme $\{1, 8\}$, and it is 92 for the initial replication scheme $\{1\}$. The optimal replication scheme is $\{8\}$, and its total communication cost is 72.

It is easy to see that Theorem 1 still holds for the ADR-G algorithm, thus the algorithm preserves replication scheme connectivity.

Next we will show that for a regular read-write pattern the communication cost of the requests in each time period decreases each time the replication scheme changes. This also implies that the replication scheme will stabilize. In other words, for a regular read-write pattern, starting with any connected replication scheme the communication cost decreases in each time period, until a time period in which the replication scheme does not change. ¿From that point the communication cost will be fixed in each time period.

Unless stated otherwise, for the statement and proof of the next theorem we use the definitions and assumptions of section 5. We say that a t-regular schedule has no *blind writes* if every write issued by a processor is preceded by a read from the same processor issued in the same time period.

THEOREM 5. *Suppose that for an integer t a schedule S is t-regular and has no blind writes. Suppose further that the ADR-G algorithm changes the replication scheme at the end of the x'th time period. Then the communication cost of requests in the time period $x + 1$ is lower than the communication cost of requests in the time period x.*

PROOF. We will prove the theorem by considering an expansion, contraction, and switch, and show that each one of these changes decreases the cost of requests in a time period. For a switch this is clearly the case.

Expansion. Suppose that in the x'th time period processor j's tree-father is processor i which has a replica, and at the end of the x'th time period the replication

scheme expanded from i to j. First let us observe that by the way the ADR-G algorithm operates, if a read issued by a processor p during a time period is serviced by a processor q, then all the reads issued by p during the time period are serviced by q. Furthermore, for the writes issued by p in the same time period, the first processor of the replication scheme that they reach is q. We will use the following notation.
Rx is the set of processors for which the reads that they issue in time period x are serviced by i sending the object to j. [9]
Wx is the set of processors for which the writes that they issue in time period x are sent by j to i.
$Rx1$ is the set of processors for which the reads that they issue in time period $x+1$ are serviced by j.
$Wx1$ is the set of processors for which the writes that they issue in time period $x+1$ are sent by j to i.

Let us consider how the expansion affects the cost of requests. The cost of every write issued by a processor that is not in $Wx1$ is higher in time period $x+1$ than in time period x by at most $c(i,j)$. The cost of every read issued by a processor that is in both, Rx and $Rx1$, is lower in time period $x+1$ than in time period x by $c(i,j)$. We will show that $Rx \subseteq Rx1$ and $Wx \subseteq Wx1$. Since the ADR-G algorithm expanded from i to j we know that the total number of reads issued by processors in Rx is higher than the total number of writes issued by processors that are not in Wx. From this we conclude that the expansion from i to j reduces the cost of requests in a time period.

Left to show is that $Rx \subseteq Rx1$ and $Wx \subseteq Wx1$. For the first containment, suppose that a processor p is in Rx. It means that there is a shortest path from i to p that goes through j. Furthermore, it means that the tree-father path from p to the replication scheme goes to i through j. Then a request issued by p in time period $x+1$ must be serviced by j. For the second containment, suppose that a processor p is in Wx. Since there are no blind writes, p is also in Rx, and by previous containment p is in $Rx1$. Thus, by the way writes are processed and by definition of $Wx1$, p must be in $Wx1$.

Contraction. Suppose that in the x'th time period processor j is in the replication scheme, and has only one tree-neighbor in the replication scheme, namely processor i. Suppose further that at the end of in the x'th time period j contracts out of the replication scheme. We will use the following notation.
Rx is the set of processors for which the reads that they issue in time period x are serviced by j.
Wx is the set of processors for which the writes that they issue in time period x are sent by j to i.
$Wx1$ is the set of processors for which the writes that they issue in time period $x+1$ are sent by j to i.

Let us consider how the contraction affects the cost of requests. The cost of every read issued by a processor in Rx is higher in time period $x+1$ than in time period x by at most $c(i,j)$. Observe that the fact that j contracted out implies that the

[9]In other words, for every processor p in Rx, there is a shortest path between i and p that goes through j.

number of writes issued by processors that are not in Wx is higher than the number of reads issued by processors in Rx. Assume that the replication scheme in time period x is identical to that in time period $x+1$, except that processor j has a replica. It is easy to see that the cost of a write issued by a processor in Wx does not increase in time period $x+1$ compared to time period x.

To complete the proof that the contraction reduces cost, we will show that the cost of a write w of a processor p that is not in Wx is lower by at least $c(i,j)$ in time period $x+1$ compared to time period x. If p is not in $Wx1$ then clearly the cost of w is lower by $c(i,j)$ after the contraction. Thus, suppose now that p is in $Wx1$, and consider the write w in time period x. The cost of w is $a+T$, where a is the cost of the tree-father path from p to the replication scheme, and T is the cost of the tree spanning the replication scheme. Observe that since p is in $Wx1$, p is not in the replication scheme in time period $x+1$. The cost of a write issued by p in time period $x+1$ is is $b+T'$, where b is the cost of the tree-father path from p to i (through j), and T' is the cost of the tree spanning the replication scheme. Notice that the difference in cost between T and T' is $c(i,j)$, and b is not higher than a since the tree-father path from p to the replication scheme changed to go through j after the first read issued by p in time period $x+1$. Thus the cost of w is lower by at least $c(i,j)$ in time period $x+1$ compared to time period x. $\square$

We conjecture that the performance of the ADR-G algorithm can be further improved if every write being propagated carries with it the identification of all the processors of the replication scheme traversed. In other words, whenever a processor of the replication scheme propagates a write to its tree-neighbors, it also appends its identification to the message being propagated. Thus, when a processor p of the replication scheme receives a write, it knows the path in the replication scheme traversed by the write. This means that p knows more about the whole replication scheme then which tree-neighbors have a replica. However, the study of this conjecture is beyond the scope of this paper.

9. RELEVANT WORK

Generally, there are two main purposes for data-replication: performance and reliability. In this paper we address the performance issue. Most existing performance-oriented works on replicated data consider the problem of static replication, namely establishing a priori a replication scheme that will optimize performance but will remain fixed at runtime. This is also called the file-allocation problem, and it has been studied extensively in the literature (see [Dowdy and Foster 1982] for a survey, and [Wolfson and Milo 1991; Humenik et al. 1992; Ozsu and Valduriez 1991] for recent work on this problem). In contrast to the approach taken in this paper, works on static replication assume that the read write pattern is given a priori. For example, in [Wolfson and Milo 1991] we have shown that finding the (static) optimal replication scheme for a given arbitrary network (modeled as a general graph), and for a given read-write pattern is an NP-complete problem. We have shown however that the problem can be solved efficiently for tree-, ring-, and clique-network topologies. The present paper was motivated by the realization that the algorithm for tree networks lends itself naturally to distribution. However, the results in [Wolfson and Milo 1991] also indicate that a comparable algorithm (distributed, efficient, and convergent-optimal) is unlikely to exist for general networks.

Works on Quorum Consensus (such as [Agrawal and Bernstein 1991; Gifford 1979; Kumar 1991; Thomas 1979; Triantafillou and Taylor 1991]), Voting and Coterie (such as [Agrawal and El-Abbadi 1990; Adam and Tewari 1993; Garcia-Molina and Barbara 1985; Herlihy 1987; Jajodia and Mutchler 1990; Paris 1986; Spasojevic and Berman 1994]) refer to performance in the presence of failures. They address the issue of how to dynamically adjust the read/write quorums and votes in order to minimize the data accesses in case of site failures and network partition.

Another approach to improve the performance in a replicated database is to relax the serializability requirement. Works on quasi-copies ([Alonso et al. 1988; Alonso et al. 1990; Barbara and Garcia-Molina 1990]), lazy replication ([Ladin et al. 1988; Ladin et al. 1992; Ladin et al. 1990]), and bounded ignorance ([Krishnakumar and Bernstein 1991]) fall in this category. In contrast, as we show in section 3, the adaptive replication algorithms that we propose here can be combined with a concurrency control algorithm to preserve 1-copy-serializability.

Recently, a few works that address the problem of dynamic (vs. static) data replication have been published ([Awerbuch et al. 1993; Bartal et al. 1992]). The need for dynamic replication was pointed out in [Gavish and Sheng 1990; Barbara and Garcia-Molina 1993]. The algorithms in [Bartal et al. 1992] are randomized, and they require centralized decision-making by a processor that is aware of all the requests in the network. Furthermore, they assume that all requests are serial (even two reads cannot occur concurrently), but it is not clear if and how the algorithms can be combined with a concurrency control mechanism. In summary, we do not think that the [Bartal et al. 1992] algorithms are applicable to a distributed database environment.

Reference [Awerbuch et al. 1993] presents a competitive distributed deterministic algorithm for dynamic data allocation. The algorithm is presented in a very sketchy manner, but basically it operates as follows. For each write, a copy is created at the writing processor, and all the other copies are deleted. For each read, the object is replicated along a shortest path from the replication scheme to the reading processor. The algorithm has an intricate data-tracking component that enables a processor to efficiently locate the processors of the replication scheme. This data-tracking component is used in read and write operations. In [Huang and Wolfson 1993; Huang and Wolfson 1994] we have discussed similar algorithms, and have proven that they are competitive in slightly different models.

The [Awerbuch et al. 1993] algorithm is competitive, which means that there exists a constant c such that for any sequence s of read-write requests: (the cost of the [Awerbuch et al. 1993]-algorithm on s) $\leq c \times$ (the cost of the lower-bound offline algorithm on s). Thus, the communication cost of the [Awerbuch et al. 1993]-algorithm may be worse than that of the lower bound by a constant factor, i.e. a factor that is independent of the number of requests; in the [Awerbuch et al. 1993]-algorithm case the factor is $O(log^4 n)$, where n is the number of processors in the network. In other words, a competitive algorithm provides a guarantee on the performance of the algorithm for the worst case input. The worst case input is a sequence of requests in which at any point in time the next request is the worst for the current configuration (i.e. replication scheme in our case). And intuitively, this is the reason that the [Awerbuch et al. 1993] algorithm erases all copies except the one at the writing processor for each write. Creating or updating replicas requires communication, and in the worst case the next request does not take advantage of this communication; in the worst case the next request is either a write or a read from a processor farthest from a replica, depending on which one maximizes communication cost for the current replication scheme.

However, in this paper we assume that most of the time, the next request is not the one that maximizes cost. In the schedules that we assume, the read-write pattern of a processor in a time period is repeated for several time periods. In other words, the read-write pattern in a time period is in most cases predictable based on the read-write pattern in the immediately-preceding time period. Indeed, this is the case in which the ADR algorithm performs best. On the other hand, a competitive algorithm does not take advantage of schedule regularity.

It can be shown that for regular schedules the ADR algorithm is superior. For example, assume that the initial replication scheme consists of the whole set of processors, and consider the regular sequence of requests in which in each time period there is a write from a processor p, followed by a read from each other processor. Then the communication cost of the ADR algorithm in each time period is the cost of the write, i.e. the cost of propagating the object to all the processors (the reads are free). On the other hand, the [Awerbuch et al. 1993] algorithm in each time period first incurs the communication cost of deleting all the copies. Then it incurs the cost of propagating the read requests. Finally, it incurs the cost of propagating the object to all the processors. Only this last cost is incurred by the ADR algorithm.

Since the replication scheme oscillates between p and the whole set of processors, the above example also demonstrates that the [Awerbuch et al. 1993] algorithm is not convergent. Suppose now that in the above example the schedule is not perfectly regular, but in some of the time periods a read is missing. Clearly in this case the ADR algorithm still outperforms the [Awerbuch et al. 1993] algorithm. Thus, strict regularity of the schedule is not necessary for superiority of ADR.

In [Wolfson and Jajodia 1992], we proposed an earlier version of the ADR algorithm, called CAR. In contrast to the ADR algorithm, in CAR a processor examines the replication scheme, and may change it, as a result of every read-write request. This increases the overhead of adaptive replication. In the present paper we also report on experimental performance analysis, we devise a protocol to handle failures, we extend the analysis to weighted communication links, and we extend the ADR algorithm to networks modeled as general graphs; these are new contributions compared to [Wolfson and Jajodia 1992].

10. CONCLUSION

The purpose of adaptive replication algorithms is to improve the performance of distributed systems by adjusting the replication scheme of an object (i.e. the number of copies of the object, and their location), to the current access-pattern in the network. In this paper we proposed an Adaptive Data Replication (ADR) algorithm, which is executed distributively by all the processors in a tree network. The execution of the ADR algorithm is integrated with the processing of reads and writes of the object.

We discussed various issues related to the incorporation of the ADR algorithm in distributed systems. Specifically, we proposed a method of coping with storage space limitations at the various processors in the network, we discussed incorporation of ADR in various replica consistency protocols (e.g. two-phase-locking), and we discussed a method of adjusting the ADR algorithm to consider a priori information about the read-write activity in the network.

We also addressed issues of failure and recovery in adaptive replication. In particular, we showed that methods of handling failures and recovery in static replication do not necessarily carry over to the dynamic case. We proposed a mechanism by which write activity may continue even when failures occur in the network.

We analyzed (theoretically and experimentally) the communication-cost of the ADR algorithm, i.e., the average number of messages necessary for servicing a read-write request. The theoretical analysis of the algorithm was performed using a new model we introduced in this paper. It showed that in steady state [10] ADR converges the replication scheme to the optimal one, regardless of the initial scheme. Convergence occurs within a number of time-periods (for example, a time period may be a minute) that is bounded by the diameter of the network.

Our experimental results showed the following. For a fixed, randomly generated read-write pattern the communication cost of the ADR algorithm is 21% lower than that of the optimal static replication scheme. This optimal replication scheme can be used if the read-write pattern is fixed and known a priori. For a read-write pattern that is not known a priori or varies over time the communication cost of the ADR algorithm is 28% to 50% lower than that of static replication.

We also compared the performance of the ADR algorithm against that of an ideal, unrealistic algorithm that knows all future read-write requests. In other words, the ADR algorithm is "online", in the sense that it has to adjust the replication scheme based on knowledge of the past read-write requests, but not of future ones. However, the optimal way of adjusting the replication scheme clearly depends on future requests, since ideally, a processor that writes the object should send the write to the processors that will read the object in the future. Thus, the lower bound is obtained by an "offline" algorithm. Therefore, we devised a lower bound algorithm, and experimentally compared its performance to that of the ADR algorithm. The experiments have shown that on average, the ADR algorithm incurs a communication cost that is only 1.636 times that of the lower bound algorithm.

Finally, we extended the ADR algorithm to operate in a network topology modeled by a general graph. We have shown that in steady state, starting with an initial replication scheme, the ADR algorithm will change the scheme in each time period as long as it can improve performance. When this is not possible the replication scheme will stabilize, with the communication cost of this final replication scheme being at least as low as that of the initial one.

ACKNOWLEDGMENTS

We wish to thank Jeff Ullman and Moti Yung for helpful discussions. We thank the referees for their valuable suggestions. We thank Kailash Narayanan and Ramya

[10]A steady state is one in which the read-write pattern of the object is regular, namely each processor i performs $\#R_i$ reads and $\#W_i$ writes per time period. This regular read-write pattern may not be known a priori; furthermore, it may change over time (e.g. from day to day).

Raj for developing the software to run some of the experiments.

REFERENCES

ADAM, N. R. AND TEWARI, R. 1993. Regeneration with virtual copies for distributed computing systems. *IEEE Trans. Softw. Eng. 19,* 6 (June), 594–602.

AGRAWAL, D. AND BERNSTEIN, A. J. 1991. A nonblocking quorum consensus protocol for replicated data. *IEEE Trans. Parall. Distrib. Syst. 2,* 2 (April), 171–179.

AGRAWAL, D. AND EL-ABBADI, A. 1990. The tree quorum protocol: An efficient approach for managing replicated data. In *Proceedings of the Sixteenth International Conference on Very Large Databases.*

ALONSO, R., BARBARA, D., AND GARCIA-MOLINA, H. 1988. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proceedings of EDBT'88, LNCS 303.* Springer-Verlag.

ALONSO, R., BARBARA, D., AND GARCIA-MOLINA, H. 1990. Data caching issues in an information retrieval system. *ACM Trans. Database Syst. 15,* 3.

AWERBUCH, B., BARTAL, Y., AND FIAT, A. 1993. Optimally-competitive distributed file allocation. In *Twenty-fifth Annual ACM STOC*, Victoria, B.C., Canada, pp. 164–173.

BADRINATH, B. R. AND IMIELINSKI, T. 1992. Replication and mobility. In *Proceedings of the Second Workshop on the Management of Replicated Data (WMRD-II)*, Monterey, CA, pp. 9–12.

BADRINATH, B. R., IMIELINSKI, T., AND VIRMANI, A. 1992. Locating stategies for personal communication networks. In *Proceedings of the Workshop on Networking of Personal Communication Applications.*

BARBARA, D. AND GARCIA-MOLINA, H. 1990. The case for controlled inconsistency in replicated data. In *Proceedings of the IEEE workshop on replicated data.*

BARBARA, D. AND GARCIA-MOLINA, H. 1993. Replicated Data Management in Mobile Environments: Anything New Under the Sun ? manuscript.

BARTAL, Y., FIAT, A., AND RABANI, Y. 1992. Competitive algorithms for distributed data management. In *Twenty-fourth Annual ACM STOC*, Victoria, B.C. Canada.

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

CERI, S. AND PELAGATTI, G. 1984. *Distributed Database Principles and Systems.* McGraw-Hill.

DOWDY, L. W. AND FOSTER, D. V. 1982. Comparative models of the file assignment problem. *ACM Comput. Surv. 14,* 2.

DUPUY, A., SENGUPTA, S., WOLFSON, O., AND YEMINI, Y. 1991a. Design of the netmate network management system. In Proceedings of the Second International Symposium on Integrated Network Management, Washington D.C.

DUPUY, A., SENGUPTA, S., WOLFSON, O., AND YEMINI, Y. 1991b. Netmate: A network management environment. *IEEE Network.*

FISCHER, M. AND MICHAEL, A. 1992. Sacrificing serializability to attain high availability of data in an unreliable network. In *ACM Principles of Database Systems*, pp. 70–75.

GARCIA-MOLINA, H. AND BARBARA, D. 1985. How to assign votes in a distributed system. *J. ACM 32,* 4, 841–860.

GAVISH, B. AND SHENG, O. 1990. Dynamic file migration in distributed computer systems. *Commun. ACM 33,* 2.

GIFFORD, D. 1979. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operation System Principles*, pp. 150–162.

GOODMAN, D. 1991. Trends in cellular and cordless communications. *IEEE Communications Magazine*, 31–40.

GRUDIN, J. 1991. Special section on computer supported cooperative work. *Commun. ACM 34,* 12.

HERLIHY, M. 1987. Dynamic quorum adjustments for partitioned data. *ACM Trans. Database Syst. 12,* 2.

Huang, Y. and Wolfson, O. 1993. A competitive dynamic data replication algorithm. In *IEEE Proceedings of the Ninth International Conference on Data Engineering*, pp. 310–317.

Huang, Y. and Wolfson, O. 1994. Dynamic allocation in distributed system and mobile computers. In *IEEE Proceedings of the Tenth International Conference on Data Engineering*, pp. 20–29.

Humenik, K., Matthews, P., Stephens, A., and Yesha, Y. 1992. Minimizing message complexity in partially replicated databases on hypercube networks. Tech. Rep. TR CS-92-09 (July), Dept. of Computer Science, University of Maryland, Baltimore County.

Imielinski, T. and Badrinath, B. R. 1992. Querying in highly mobile distributed environments. In *Proceedings of the Eighteenth International Conference on Very large Databases*, pp. 41–52.

Jajodia, S. and Mutchler, D. 1990. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Syst. 15,* 2 (June), 230–280.

Krishnakumar, N. and Bernstein, A. 1991. Bounded ignorance in replicated systems. In *Proceedings of the ACM Principles of Database Systems.*

Kumar, A. 1991. Hierarchical quorum consensus : A new algorithm for managing replicated data. *IEEE Trans. Comput. 40,* 9 (September), 996–1004.

Ladin, R., Liskov, B., and Shrira, L. 1988. A technique for constructing highly available distributed services. *Algorithmica 3.*

Ladin, R., Liskov, B., and Shrira, L. 1990. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Workshop on Management of Replicated Data*, pp. 31–34.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. 1992. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10,* 4 (November).

Ozsu, M. T. and Valduriez, P. 1991. *Principles of Distributed Database Systems.* Prentice-Hall.

Paris, J. 1986. Voting with a variable number of copies. In *Fault Tolerant Computing Symposium*, pp. 50–55.

Sengupta, S., Dupuy, A., Schwartz, J., Wolfson, O., and Yemini, Y. 1990. The netmate model for network management. In *IEEE Network Operations and Management Symposium (NOMS)*, San Diego, CA, pp. 11–14.

Spasojevic, M. and Berman, P. 1994. Voting as the optimal static pessimistic scheme for managing replicated data. *IEEE Trans. Parall. Distrib. Syst. 5,* 1 (January), 64–73.

Thomas, R. H. 1979. A majority consensus approach to concurrency control for multiple copy database. *ACM Trans. Database Syst. 4,* 2 (June), 180–209.

Triantafillou, P. and Taylor, D. J. 1991. Using multiple replica classes to improve performance in distributed systems. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pp. 34–41.

Wolfson, O. and Jajodia, S. 1992. Distributed algorithms for adaptive replication of data. In *ACM Principles of Database Systems*, San-Diego, CA, pp. 149–163.

Wolfson, O. and Milo, A. 1991. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst. 16,* 1.

Wolfson, O., Sengupta, S., and Yemini, Y. 1991. Managing communication networks by monitoring databases. *IEEE Trans. Softw. Eng. 17,* 9 (September), 944–953.

Appendix A: Proof of Theorem 3

The proof proceeds in three stages, each of which corresponds to a subsection. In the first stage we prove that the ADR algorithm stabilizes. In the second stage we prove that the stabilization occurs after at most $d + 1$ time periods (where d is the diameter of the tree network). In the third stage we show that the stability scheme is optimal for the read-write pattern in each time period. Throughout this appendix we assume that S is a t-regular schedule and A is the read-write pattern of the time period t.

In order to prove the theorem, we will prove several lemmas. Throughout the proofs of the lemmas, we use the following notations. Let i and j be two adjacent processors in the tree network, and suppose that the edge between i and j is removed. We denote by $T(i,j)$ the connected component containing i; $T(j,i)$ is the other connected component. Denote the number of requests (reads or writes) issued in A from $T(i,j)$ by $NO(i,j)$; $NW(i,j)$ is the number of writes issued from $T(i,j)$; $NR(i,j)$ is the number of reads issued from $T(i,j)$.

1 The ADR algorithm stabilizes

LEMMA 1. *Consider two processors, i and j, that are in the replication scheme of the ADR algorithm at two different time periods, t_1 and t_2, respectively. Then each processor on the unique path between i and j is in the replication scheme of the ADR algorithm at some time period q, $t_1 \leq q \leq t_2$.*

PROOF. The proof follows easily from Theorem 1. □

LEMMA 2. *If a switch test succeeds, then subsequently no expansion test will succeed.*

PROOF. Suppose that in some time-period q the object is switched from processor i to processor j. Then $NO(j,i) > NO(i,j)$. Since the switch test from i to j is not executed unless the expansion test of i (to all neighbors, including j) fails, we derive $NR(j,i) \leq NW(i,j)$. Subtracting the second inequality from the first, we obtain $NW(j,i) > NR(i,j)$. Thus, the expansion test from j to i will fail.

Next, we show that the expansion test of j to any other neighbor k will fail too. Clearly, $NR(k,j) \leq NR(j,i)$ since the tree $T(k,j)$ is a subgraph of the tree $T(j,i)$. Similarly, $NW(i,j) \leq NW(j,k)$. Combining these two inequalities and the second inequality of the last paragraph we derive $NR(k,j) \leq NW(j,k)$. This inequality implies that the expansion test from j to k will fail. □

LEMMA 3. *If a processor j exits from the replication scheme as a result of the contraction test, then j will not reenter the replication scheme as a result of the expansion test.*

PROOF. Suppose that in time period t_1, j is an R-fringe processor with a single neighbor i in R, and $NW(i,j) > NR(j,i)$. Hence j is deleted from R by the contraction test. Assume by way of contradiction that afterwards, in time period t_2, j is told by some processor, i', to join the replication scheme, as a result of the expansion test executed by i'. Consider the first time this happens, namely, the first time j reenters the replication scheme as a result of the expansion test. Then $NR(j,i') > NW(i',j)$. By lemma 2, between time periods t_1 and t_2 j could not have reentered as a result of the switch test. Thus, j is not in the replication scheme between time periods t_1 and t_2. Therefore, from lemma 1, we know that there is a path from i to i' that does not go through j. Thus, unless $i = i'$, there is a cycle $j \to i \to i' \to j$ in the tree network. Therefore, $i = i'$. But then, the last two inequalities contradict each other. □

Although we do not need the result for the proof of theorem 3, let us mention that a processor can exit the replication scheme as a result of the contraction test and then reenter as a result of the switch test.

LEMMA 4. *A processor that exits from the replication scheme as a result of the switch test cannot reenter the replication scheme.*

PROOF. Suppose that the object is switched from i to j at the end of a time period. Then $NO(j,i) > NO(i,j)$. Then, by lemma 2, the singleton replication scheme cannot expand, it can only switch to a neighbor or stabilize. From the inequality we obtained, the replication scheme cannot switch from j to i. In the tree network, except for the edge $i - j$, there is no other path which connects j to i. Therefore i will not be switched into the replication scheme, and the lemma follows. □

LEMMA 5. *The ADR algorithm will stabilize in a finite number of time periods.*

PROOF. Lemmas 3 and 4 indicate that a processor k that exits from the replication scheme, can reenter only if its exit is by a contraction test, and its reentry is by a switch test. But then, once there is a successful switch test, by lemma 2, a subsequent change of the replication scheme can occur only by a switch test. By lemma 4, processor k can exit the replication scheme at most once more. Therefore, each processor can exit at most twice from the replication scheme, once by contraction and once by switch. Consequently, there must be a time period x, starting from which no processor exits from the replication scheme. Therefore starting at x the replication scheme can only expand. Since the number of processors is finite, the ADR algorithm will stabilize after a finite number of time periods. □

2 The ADR algorithm stabilizes in $d + 1$ time periods

In the proofs of lemmas 6-10 we denote by R_0 the replication scheme in the first time period, we denote by R_k the replication scheme resulting from the tests executed at the end of the k-th time period, and we denote by F the stability scheme. The *length* of a path Q between two processors is the number of communication links between them in the tree network; this length is denoted by $|Q|$. The *distance* between two processors m and n, denoted $dist(m,n)$, is the length of the path between them.

LEMMA 6. *If R_q is a singleton for some $q \geq 1$, then R_{q+1} is also a singleton.*

PROOF. Suppose that $R_q = \{x\}$. We prove the lemma in the following two cases. 1) R_{q-1} is a singleton; and 2) R_{q-1} is not a singleton.

First, suppose that R_{q-1} is a singleton and $R_{q-1} = \{z\}$. If $z = x$, then the replication scheme stabilizes on x and $R_{q+1} = \{x\}$. If $z \neq x$, then by lemma 2 R_{q+1} is also a singleton.

Second, suppose that R_{q-1} is not a singleton. Then $x \in R_{q-1}$, since by the definition of the ADR algorithm a processor cannot successfully expand to x and contract itself out of the replication scheme at the end of the same time period. Suppose that y is an arbitrary neighbor of x. If $y \in R_{q-1}$, then y must have contracted out of the replication scheme at the end of time period q, hence cannot reenter the replication scheme as a result of an expansion test (by lemma 3). Therefore, the expansion from x to y fails at the end of the time period $q + 1$. Now suppose that $y \notin R_{q-1}$. Then at the end of time period $q + 1$ the expansion from x to y fails too, since $x \in R_{q-1}$ and the expansion from x to y failed at the end of time period q. Thus, the expansion test executed by processor x fails, and the singleton replication scheme $R_q = \{x\}$ can only switch or stabilize. $\Box$

LEMMA 7. *Suppose that R_w is not a singleton for any $w < q$. Suppose that at the end of time period q, the R_{q-1}-fringe processor y (which has a single neighbor $x \in R_{q-1}$) exits the replication scheme by a contraction test. Then $R_0 \cap T(y,x) \neq \emptyset$, and $q = \max_{z \in R_0 \cap T(y,x)} dist(x,z)$. (Namely, if a processor v belonging to T(y, x) is in the replication scheme in the first time period, then it must be within distance q away from x.)*

PROOF. First we show $R_0 \cap T(y,x) \neq \emptyset$. Assume by way of contradiction that $R_0 \cap T(y,x) = \emptyset$. Since R_w is not a singleton for any $w < q$, no processor can enter the replication scheme as a result of the switch test before time period q. Since $y \in R_{q-1}$, by lemma 1, processor y must have entered the replication scheme for the first time before the q'th time period, as a result of an expansion test executed by x. Thus $NR(y,x) > NW(x,y)$. This inequality implies that the contraction test executed by processor y at the end of time period q fails. It contradicts the fact that y exits by a contraction test.

Now we prove the equality $q = \max_{z \in R_0 \cap T(y,x)} dist(x,z)$ by induction on q.

Consider first the case $q = 1$, namely y exits from the replication scheme at the end of the first time period. Then y must be an R_0-fringe processor, with x as the only neighbor in R_0. Thus, the set $R_0 \cap T(y,x)$ consists of y alone, and the equality holds.

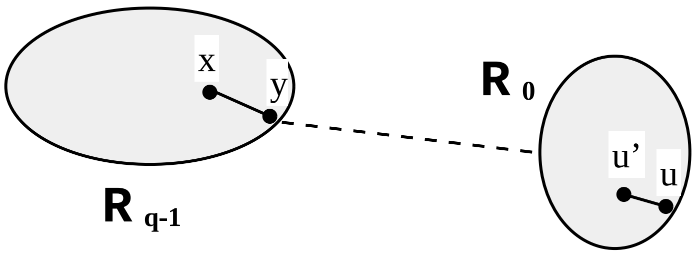

Fig. 9.

Now, suppose that for $q < n$ the equality holds. We consider the case $q = n$. Suppose that $u \in R_0 \cap T(y,x)$, and $dist(x,u) = \max_{z \in R_0 \cap T(y,x)} dist(x,z)$. Then u must be an R_0-fringe processor in the first time period, otherwise one of its neighbors in R_0 is farther away from x. Since R_0 is not a singleton replication scheme, there must exist a unique neighbor u' of u such that $u' \in R_0$. Then u' must be on the path that connects u to x and $dist(u',x) = dist(u,x) - 1$ (see figure 9). Since y executes a successful contraction test at the end of time period q, we must have $NR(y,x) < NW(x,y)$. Since $T(u,u')$ is a subtree of $T(y,x)$, $NR(u,u') \leq NR(y,x)$. Similarly, $NW(x,y) \leq NW(u',u)$. These three inequalities combined imply that $NR(u,u') < NW(u',u)$. Thus, the R_0-fringe processor u contracts out of the replication scheme at the end of the first time period.

From the previous paragraph we conclude that $\max_{z\in R_1\cap T(y,x)} dist(x,z) = \max_{z\in R_0\cap T(y,x)} dist(x,z) - 1$. Starting from the second time period (with the non-singleton replication scheme R_1), processor y exits the replication scheme in another $q-1$ time periods. By the induction hypothesis, we obtain $q-1 = \max_{z\in R_1\cap T(y,x)}$. Thus, $q = \max_{z\in R_0\cap T(y,x)} dist(x,z)$. □

For any pair of connected replication schemes R_1 and R_2, we denote by $dist(R_1, R_2)$ the length of the shortest path from R_1 to R_2, i.e., $dist(R_1, R_2) = \min_{m\in R_1, n\in R_2} dist(m,n)$. Suppose that P is a longest path from a processor $i \in R_0$ to a processor $j \in F$, and $|P| = p$. Namely, $dist(i,j) = p = \max_{m\in R_0, n\in F} dist(m,n)$.

Lemma 8. *If R_0 is not a singleton and $F = \{j\}$ (i.e. F is a singleton), then after at most d time periods the ADR algorithm stabilizes.*

Proof. Suppose that q is the first time period at which the replication scheme R_{q-1} is a singleton. In other words, R_w is not a singleton for each $w < q-1$. Let $R_{q-1} = \{k\}$. Since R_0 is not a singleton, $q > 1$.

claim 1. *$k \in P$ (where P is the longest path from a processor $i \in R_0$ to j).*
Proof: We prove this claim by way of contradiction. Suppose that $k \notin P$. By lemma 6 the replication scheme switches starting from time period q. The replication scheme moves from k to j one step in a time period, and reaches j in a total of $dist(k,j)$ time periods (based on lemmas 2, 4 and 6 the replication scheme cannot diverge from the path from k to j and then come back to it). Let $R_q = \{s\}$. Denote by t the closest processor to k that is on P (see figure 10). It is possible that $s = t$.

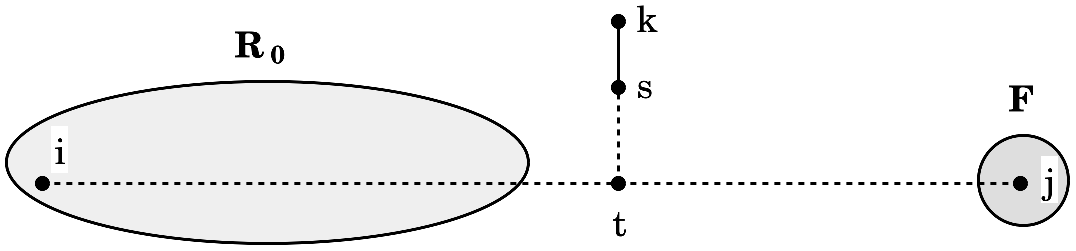

Fig. 10.

Now we will draw a contradiction in each one of the following cases.

Case I) $k \in R_0$.. We will demonstrate that there is a processor $v \in R_0$ that is farther away from j than i, contradicting the definition of the path P.

Since R_0 is connected, all processors (including s and t) in the path connecting i to k must all belong to R_0. Since q is the time period at which the replication scheme becomes a singleton for the first time, by lemma 3, k must be in the replication scheme at all the time periods before q. Furthermore, processor s must exit the replication scheme by a contraction test before time period q. Thus, we obtain $NR(s,k) < NW(k,s)$. Since at the end of time period q the switch test (from k to s) succeeds, we obtain $NO(s,k) > NO(k,s)$. Subtracting the first inequality from the second, we obtain $NW(s,k) > NR(k,s)$. This inequality implies that **if k is an R-fringe processor while s is in the replication scheme, then the contraction**

test of k will succeed. But we know that k does not exit the replication scheme in the first q time periods. Therefore k does not become an R-fringe before s does so. Furthermore, observe that before time period q there will be at most one processor on the path between i and k that exits from the replication scheme in each time period (remember that the whole path is in R_0). Therefore s will not become an R-fringe processor before $dist(i,s)$ time periods. This implies that k does not become an R-fringe processor before $dist(i,s)$ time periods. Thus, there must exist a processor $v \in R_0 \cap T(k,s)$ such that $dist(v,k) \geq dist(i,s)$ (see figure 11). Clearly $dist(v,j) = dist(v,k) + 1 + dist(s,j)$, and $dist(i,j) \leq dist(i,s) + dist(s,j)$. Based on the last three inequalities we obtain $dist(v,j) > dist(i,j)$.

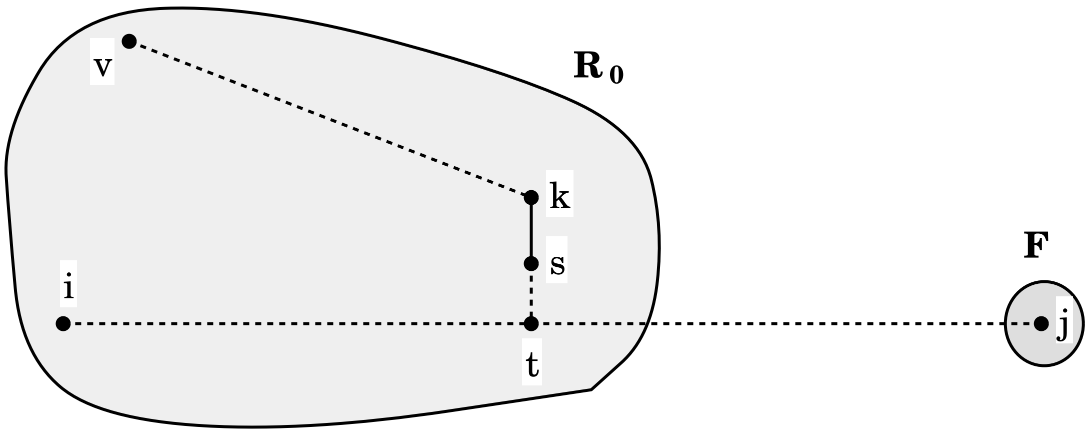

Fig. 11.

Case II) $k \notin R_0$.. Consider figure 10 again. By lemma 1 the processor s must be in the replication scheme before k enters the scheme. By the way q is defined, k must join the replication scheme as a result of an expansion test executed by s. Therefore, $NR(k,s) > NW(s,k)$. At the end of time period q the switch test from k to s succeeds, thus $NO(k,s) < NO(s,k)$. Subtracting the first inequality from the second, we obtain $NW(k,s) < NR(s,k)$. Thus, at the end of time period q, the expansion test (instead of the switch test) from k to s should have succeeded. This is a contradiction.

CLAIM 2. *$dist(i,k) = \max_{z \in R_0} dist(z,k)$, and $q = dist(i,k) + 1$.*
Proof: We prove this claim in the following cases. Both cases use claim 1.

Case I) $k \in R_0$.. We show that $dist(i,k) = \max_{z \in R_0} dist(z,k)$ by way of contradiction. Suppose that there exists a processor $v \in R_0$ such that $dist(k,v) > dist(k,i)$. By definition of P we can easily see that $k \neq j$. Suppose that x is the neighbor of k on the path between k and j. If the path between v and k does not go through x, then v is farther away from j than i in contradiction to the selection of P. Thus, it does go through x (see figure 12).

Therefore, all the processors in $T(k,x)$, except k, must exit the replication scheme before x exits. Namely k will be an R-fringe processor before x contracts out. By

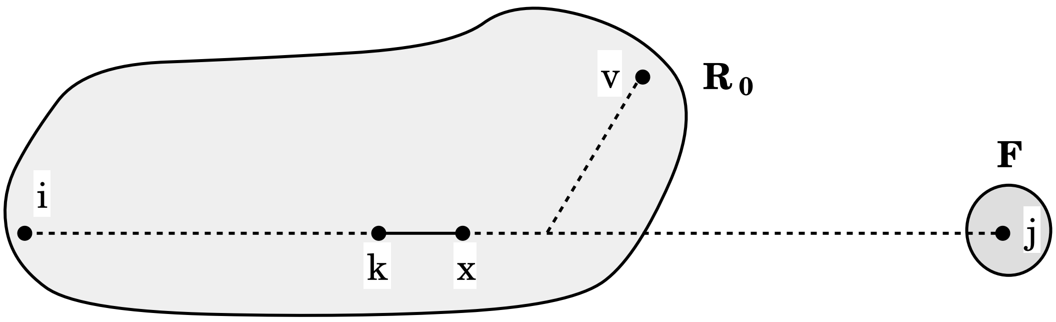

Fig. 12.

lemma 3 and the way q is defined, k does not execute a successful contraction test before time period q. Thus, $NR(k,x) \geq NW(x,k)$.

By lemma 6, starting from time period q the replication scheme consists of a single processor, and it starts to move from k to j one step in a time period. Thus, at the end of time period q the replication scheme switches from k to x. This switch implies the following two inequalities. 1) $NR(x,k) \leq NW(k,x)$ since the expansion test (from k to x) fails; and 2) $NO(x,k) > NO(k,x)$ since the switch test succeeds. Subtracting the first inequality from the second, we obtain $NW(x,k) > NR(k,x)$. But this contradicts the inequality that we obtained in the previous paragraph.

Therefore, $dist(i,k) = \max_{z \in R_0} dist(z,k)$. By lemma 7 we conclude that any neighbor $v \in R_0$ of k exits the replication scheme within $dist(i,k)$ time periods, thus $q = dist(i,k) + 1$.

Case II) $k \notin R_0$.. We know from claim 1 that k is in P, thus $dist(i,k) = \max_{z \in R_0} dist(z,k)$, which proves the first part of claim 2.

Denote by y the closest processor to k in R_0. Denote by x the neighbor of k on the path between y and k. Thus we have the situation of figure 13.

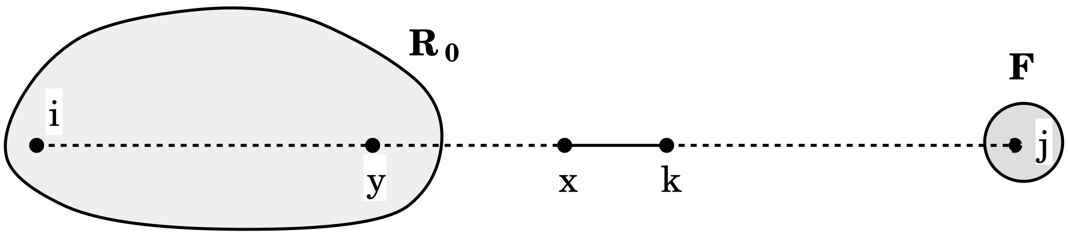

Fig. 13.

By the definition of q, k will enter the replication scheme by an expansion test executed by processor x. We claim that R_{q-2} is $\{k,x\}$. The reasons for this are the following: 1) R_{q-2} is not a singleton, i.e., it includes k and some of its neighbors, and 2) since at time period q the expansion test must fail (by lemma 6) the expansion test of k could not have succeeded to include any of its neighbors.

Since $R_{q-1} = \{k\}$, x contracts out of the replication scheme at the end of time period $q-1$, thus, from lemma 7 we conclude that $q - 1 = dist(i,k)$.

Therefore, by claim 2 the ADR algorithm takes $dist(i,k)$ time periods to shrink to the singleton replication scheme $\{k\}$. By lemmas 2 and 6 at time period q the replication scheme starts to move from $\{k\}$ to $\{j\}$, and it reaches $\{j\}$ in another $dist(k,j)$ time periods. Then it stabilizes. Observe that $p = dist(i,k) + dist(k,j)$ since $k \in P$ (by claim 1). In other words, the ADR algorithm stabilizes in p time periods. Obviously, $p \leq d$, since d is the length of the longest path in the tree. □

LEMMA 9. *If neither R_0 nor F is a singleton, then after at most d time periods the ADR algorithm stabilizes.*

PROOF. In order to prove this lemma, we state the following claims.

CLAIM 1. *R_k is not a singleton for any k.*
Proof follows easily from lemma 6.

CLAIM 2. *If a processor x is in some R_q, then $x \in R_k$ for $k = dist(x, R_0)$.*
Proof follows easily from lemma 1.

CLAIM 3. *If for some $k \geq 1$ there exists a processor $y \in R_k \setminus R_0 \setminus F$, then $R_0 \cap F = \emptyset$. Furthermore, $y \notin R_p$ (remember that p is the length of the path between i and j).*
Proof: By claim 2 processor y enters the replication scheme in $dist(y, R_0)$ time periods. Suppose that x is the neighbor of y on the path that connects y to R_0. Then all processors of R_0 must be in the subtree $T(x,y)$. Since $y \notin F$, by claim 1, proces-

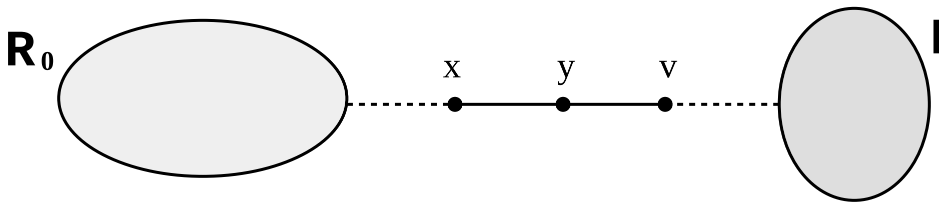

Fig. 14.

sor y will become an R-fringe processor and contract out of the replication scheme. Suppose that in time period w processor y becomes an R-fringe processor with the only neighbor v in the replication scheme, and suppose that at the end of time period w processor y executes a successful contraction test. It is easy to see that x must be different than v (see figure 14). By lemma 3, processor y will not reenter the replication scheme, and the stability scheme is in the subtree $T(v,y)$. Since $T(x,y) \cap T(v,y) = \emptyset$, $R_0 \cap F = \emptyset$. From lemma 7 we derive $w = \max_{u \in R_0} dist(v,u)$. Thus $w = dist(v,i)$. Since the path P goes through v, $dist(v,i) \leq p$. Since $y \notin R_w$, by lemma 3, $y \notin R_p$.

CLAIM 4. *If there exists a processor $y \in R_0 \setminus F$, then $y \notin R_p$.*
Proof: Suppose that x is the processor of F which is closest to y. Suppose that z is a neighbor of x on the path between x and y (see figure 15). By lemma 7, all the processors in the subtree $T(z,x)$ should contract out within $\max_{v \in R_0 \cap T(z,x)} dist(v,x)$

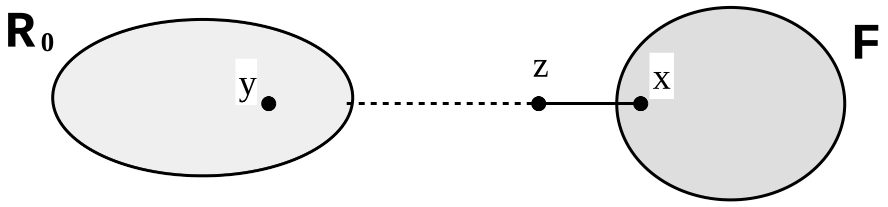

Fig. 15.

time periods. From the definition of P we know that $p \geq \max_{v \in R_0 \cap T(z,x)} dist(v,x)$. Thus, y exits the replication scheme before time period p (and, by lemma 3, never re-enters).

From claim 1 we know that only expansion and contraction tests may succeed. By claim 2, a processor x in the stability scheme F will be in the replication scheme in the time period $dist(x, R_0)$. By lemma 3 a processor of F cannot exit the replication scheme. Clearly, by definition of P, $dist(x, R_0) \leq p$. Namely, all processors of F will enter the replication scheme within p time periods and never exit. Thus, F is a subset of R_p. By claims 3 and 4, a processor which is not in the stability scheme F is not in R_p. Thus, $R_p = F$. □

LEMMA 10. *If $R_0 = \{i\}$ is a singleton, then after at most $(d+1)$ time periods the ADR algorithm stabilizes.*

PROOF. At the end of the first time period either no test succeeds, or a switch test succeeds, or an expansion test succeeds. We will prove the lemma in each one of the following three cases.

Case I) No test succeeds.. Then $F = R_0$, i.e., at the first time period the algorithm stabilizes and the lemma follows trivially.

Case II) The replication scheme switches.. By lemma 2, there will be no successful expansion tests afterwards, and the only possible successful tests would be switch tests. Since there is a unique path connecting i and j in the tree, by lemma 4 the algorithm must move the replication scheme from i to j in p time periods and stabilize. Obviously, $p \leq d$ since the diameter d of a tree is the length of the longest path in the tree.

Case III) The replication scheme expands.. Then the replication scheme in the second time period is not a singleton. By lemmas 8 and 9, starting from the second time period, in at most d time periods the algorithm stabilizes.

□

The following example shows that there are read-write patterns for which stabilization will take $d+1$ rather than d time periods. Consider a network of two processors i and j with the initial replication scheme $\{i\}$. The read-write pattern is that i issues 1 read and 1 write, j issues 3 reads and 2 writes in each time period. Then, the replication will expand to j at the end of time period 1. At the end of time period 2 the processor i will contract out, and the replication scheme will stabilize at $\{j\}$.

3 The stability scheme is optimal for the read-write pattern A

LEMMA 11. *Suppose that R is a connected replication scheme. Suppose that j is an $\overline{R}$-neighbor processor, and i ($i \notin R$) is adjacent to j. Then*

$$cost(R \cup \{i\}, A) - cost(R, A) = (NW(j,i) - NR(i,j)) \cdot c(i,j) \qquad (1)$$

PROOF. Let us consider how the cost of requests in the subtrees $T(j,i)$ and $T(i,j)$ changes when adding the processor i to the scheme R (see figure 16). The cost of

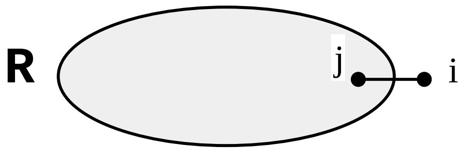

Fig. 16.

the reads in the subtree $T(j,i)$ will not change, but every write in the subtree $T(j,i)$ will cost $c(i,j)$ more (propagating from j to i), hence the requests in $T(j,i)$ will be more costly by $NW(j,i) \cdot c(i,j)$. Now consider the requests in subtree $T(i,j)$. Each read will cost $c(i,j)$ less than before, since instead of accessing the copy at j a read can access the copy at i. The cost of writes in $T(i,j)$ will not change. Thus, the requests in $T(i,j)$ become less costly by $NR(i,j) \cdot c(i,j)$. $\square$

In lemmas 12-17 we assume that F is the stability scheme, and that F is not a singleton; then we analyze the special case of the singleton stability scheme.

LEMMA 12. *Suppose that i ($\in F$) is an $\overline{F}$-neighbor processor, and j ($j \notin F$) is adjacent to i. Then $NW(j,i) \leq NR(i,j)$*

PROOF. Since F is not a singleton, there must exists an F-fringe processor $k \neq i$. Since F is connected, the processors which lie in the path from k to i must all belong to R. Suppose that n is a neighbor of k on this path between k and i (see figure 17). Obviously, we have $NR(k,n) \leq NR(i,j)$ and $NW(j,i) \leq NW(n,k)$. Since

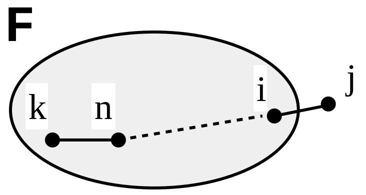

Fig. 17.

F is the stability scheme, the contraction-test of the R-fringe processor k must fail. Thus, $NW(n,k) \leq NR(k,n)$. These three inequalities imply that $NW(j,i) \leq NR(i,j)$. $\square$

LEMMA 13. *Suppose that P is a connected replication scheme such that $F \cap P = \emptyset$. Suppose that k is a processor in the path which links P to F, and k is adjacent to P. Then $cost(P, A) \geq cost(P \cup \{k\}, A)$.*

PROOF. Assume that the path which links P to F is $h - k - \ldots - j - i$, where $h \in P$, $i \in F$, and the processors on the path between them are not in F nor in P (see figure 18). Obviously, $NR(i, j) \leq NR(k, h)$, and $NW(h, k) \leq NW(j, i)$. From

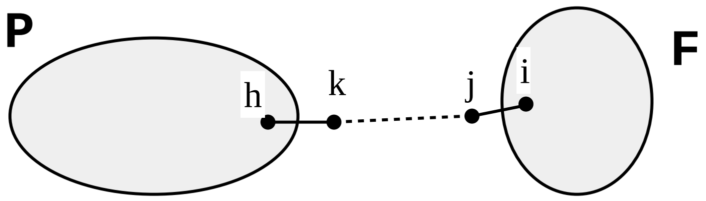

Fig. 18.

lemma 12, $NW(j, i) \leq NR(i, j)$. Combining the last three inequalities, we obtain $NW(h, k) \leq NR(k, h)$. From this inequality and lemma 11 we can easily see that $cost(P, A) \geq cost(P \cup \{k\}, A)$. $\square$

The next lemma indicates that if a connected replication scheme P contains all the fringe processors of another connected replication scheme R, then P contains all the processors of R.

LEMMA 14. *Suppose that P and R are two different connected replication schemes. Suppose that $P \cap R \neq \emptyset$, and $R \setminus P \neq \emptyset$ (i.e., R is not a subset of P). Then there must exist an R-fringe processor in the subtree of the network induced by $R \setminus P$.*

PROOF is straightforward.

LEMMA 15. *Suppose that P is a connected replication scheme such that $F \cap P \neq \emptyset$. Then $cost(P, A) \geq cost(P \cup F, A)$.*

PROOF. We prove the lemma by induction on the number of processors in $F \setminus P$. If $F \setminus P = \emptyset$ then the lemma follows trivially. Now, suppose that the lemma holds for P where $F \setminus P$ contains $n - 1$ processors. We consider the case of P where $F \setminus P$ contains n ($n \geq 1$) processors. From lemma 14 we know that there exists an F-$fringe$ processor q in $F \setminus P$. Suppose that i is the closest processor to q in $P \cap F$, and the path between q and i is $q - p - \ldots - j - i$ (see figure 19).

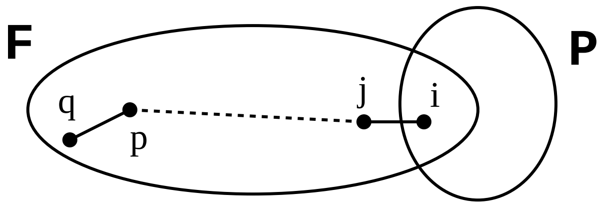

Fig. 19.

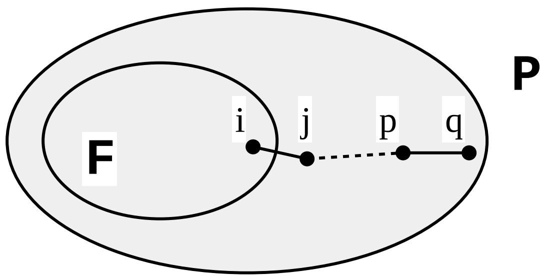

Fig. 20.

We will show that $cost(P, A) \geq cost(P \cup \{j\}, A)$, and thus the lemma will follow by the induction hypothesis. Since F is the stability scheme, the contraction-test of q must fail. Hence $NR(q, p) \geq NW(p, q)$. Obviously, $NR(j, i) \geq NR(q, p)$, and $NW(p, q) \geq NW(i, j)$. ¿From the last three inequalities we obtain $NR(j, i) \geq NW(i, j)$. From this inequality and lemma 11, $cost(P, A) \geq cost(P \cup \{j\}, A)$. □

Lemma 16. *Suppose that P is a connected replication scheme such that $F \subset P$. Then $cost(P, A) \geq cost(F, A)$.*

Proof. We prove the lemma by induction on the number of processors of $P \setminus F$. Suppose that the lemma holds for every connected replication scheme P, where $P \setminus F$ consists of $n - 1$ processors, for $n \geq 1$. Assume now that $P \setminus F$ consists of n processors. By lemma 14, there exists a P-$fringe$ processor q so that $q \in P \setminus F$. Suppose that i is the closest processor to q in F, and the path between q and i is $q - p - \ldots - j - i$, where $j \notin F$ and $i \in F$ (see figure 20).

Since F is the stability scheme, the expansion-test from i to j must fail, namely, $NR(j, i) \leq NW(i, j)$. Obviously we have $NR(q, p) \leq NR(j, i)$, and $NW(i, j) \leq NW(p, q)$. From these three inequalities, $NR(q, p) \leq NW(p, q)$. From the last inequality and lemma 11 we conclude that $cost(P, A) \geq cost(P', A)$, where $P' = P \setminus \{q\}$. Obviously, $F \subseteq P'$, P' is a connected replication scheme, and $P' \setminus F$ contains $n - 1$ processors. By the induction hypothesis we conclude that $cost(P', A) \geq cost(F, A)$. The lemma follows from the last two inequalities. □

Lemma 17. *Suppose that P is an arbitrary connected replication scheme. Then $cost(P, A) \geq cost(F, A)$.*

Proof. We prove the lemma in the following two cases:

Case I) $P \cap F \neq \emptyset$.. By lemma 15, $cost(P, A) \geq cost(P \cup F, A)$, and by lemma 16, $cost(P \cup F, A) \geq cost(F, A)$.

Case II) $P \cap F = \emptyset$.. There exists a unique path which links P to F in the tree network. Suppose that $i_0 - i_1 - \ldots - i_{n-1} - i_n$ is such a path, where $i_0 \in P$, $i_n \in F$, and the processors in between do not belong to $P \cup F$. Denote the set of all processors on this path by L. Then, by applying lemma 13 n times we will obtain $cost(P, A) \geq cost(P \cup L, A)$. Let $P' = P \cup L$. Then P' is a connected replication scheme, and $P' \cap F \neq \emptyset$. From the proof of case I), we know that $cost(P', A) \geq cost(F, A)$.

□

LEMMA 18. *Suppose that i and j are two adjacent processors in the tree network. Then $cost(\{i\},A) - cost(\{j\},A) = (NO(j,i) - NO(i,j)) \cdot c(i,j)$.*

PROOF. Obvious $\square$

In lemmas 19-20 we assume that the stability scheme F is the singleton $\{k\}$.

LEMMA 19. *Suppose that P is a connected replication scheme, and $k \in P$. Then $cost(P,A) \geq cost(F,A)$.*

PROOF. We prove the lemma by induction on the number of processors in P. If P is a singleton, then $P = F$ and the lemma follows trivially.

Suppose that the lemma holds for any set P of connected processors that has $n-1$ processors. Now consider the case where P has n processors, where $n \geq 2$. By lemma 14, there must exist a *P-fringe* processor in $P \setminus F$, say p. Suppose that the path between p and k is $p - q - \ldots - j - k$. Since F is the stability scheme, the expansion-test from k to j must have failed, namely, $NR(j,k) \leq NW(k,j)$. Obviously $NR(p,q) \leq NR(j,k)$, and $NW(k,j) \leq NW(q,p)$. These three inequalities imply that $NR(p,q) \leq NW(q,p)$. From this inequality and lemma 11 we obtain that $cost(P,A) \geq cost(P',A)$, where $P' = P \setminus \{p\}$. Then P' is a connected replication scheme, $k \in P'$, and P' has $n-1$ processors. By the induction hypothesis we obtain $cost(P',A) \geq cost(F,A)$. The lemma follows from the last two inequalities. $\square$

LEMMA 20. *Suppose that P is a connected replication scheme, and $k \notin P$. Then $cost(P,A) \geq cost(F,A)$.*

PROOF. We prove the lemma by induction on the number of processors in P. First, consider a singleton replication scheme $P = \{i\}$, where $k \neq i$. Suppose that the path between i and k is $i - q - \ldots - j - k$. Since F is the stability scheme, the switch-test from k to j must fail, namely, $NO(j,k) \leq NO(k,j)$. Obviously $NO(i,q) \leq NO(j,k)$, and $NO(k,j) \leq NO(q,i)$. These three inequalities imply that $NO(i,q) \leq NO(q,i)$. From this inequality and lemma 18, we conclude that $cost(\{i\},A) \geq cost(\{q\},A)$. Using the above technique repeatedly along the path from i to k, we can show that $cost(\{i\},A) \geq cost(\{q\},A) \geq \ldots \geq cost(\{j\},A) \geq cost(\{k\},A)$.

Suppose now that the lemma holds for every connected replication scheme P, where P has $n-1$ processors. Now we assume that P has $n \geq 2$ processors. Suppose that i is the closest processor to k in P, and the path between i and k is $L = i - q - \ldots - j - k$, where $i \in P$ and $q \notin P$. From lemma 14, we know that there exists a *P-fringe* processor in $P \setminus \{i\}$. Suppose that p is such a processor, and the path between p and i is $M = p - x - \ldots - i$. Since P is connected, every processor in M must belong to P, and the path M overlaps with path L only at i. Therefore, the path linking p to k is the concatenation of M and L, namely $p - x - \ldots - i - q - \ldots - j - k$ (see figure 21).

Since F is the stability scheme, the expansion-test from k to j must have failed, namely, $NR(j,k) \leq NW(k,j)$. Obviously $NR(p,x) \leq NR(j,k)$ and $NW(k,j) \leq NW(x,p)$. These three inequalities imply that $NR(p,x) \leq NW(x,p)$. From this inequality and lemma 11 we conclude that $cost(P,A) \geq cost(P',A)$, where $P' = P \setminus \{p\}$. Therefore P' is a connected replication scheme, $k \notin P'$, and P' has

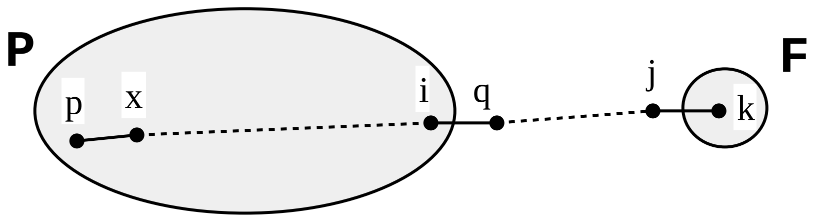

Fig. 21.

$n-1$ processors. By the induction hypothesis, $cost(P', A) \geq cost(F, A)$. Therefore, $cost(P, A) \geq cost(F, A)$. □

Proof of theorem 3. From lemma 5 and lemmas 8-10 we conclude that if the schedule S is t-regular, then the ADR algorithm will stabilize after at most $(d+1)$ time periods. Denote the stability scheme by F. Suppose that P is an arbitrary replication scheme. By theorem 2, it suffices to show that $cost(P, A) \geq cost(F, A)$, for every connected replication scheme P. But for such a replication scheme the inequality follows from lemmas 17, 19 and 20. □

Appendix B: Pseudo code of the ADR algorithm

The code uses two variables, *Req* and *Id*. *Req* is the request type. The request types are: 'read', 'write', 'time slice t expires', 'exit', 'join', and 'switch'. *Id* is the processor identification from which the request is submitted. The 'time slice t expires' request is executed only by processors of the replication scheme. The other requests are executed by all the processors in the network.

Each processor p in the network maintains a directory record *Nb->Drt* for each one of its neighbors. For each neighbor Nb, Nb->Drt = 1 if and only if the object is replicated at a processor q such that the path from p to q goes through *Nb*. If Nb has a replica, then clearly Nb->Drt = 1. If p does not have a replica, then Nb->Drt = 1 for exactly one neighbor Nb. The ADR algorithm uses this directory information for the routing of the read/write requests to the replicated object.

Each processor p that has a replica of the object maintains the counters *'My->SumR'*, *'My->SumW'*, *'Nb->NoR'*, and *'Nb->NoW'*, where

—'My->SumR' counts the total number of reads performed at p (the requests could be issued locally or from a neighbor),

—'My->SumW' counts the total number of writes performed at p,

—'Nb->NoR' counts the total number of reads submitted from a neighbor Nb; and

—'Nb->NoW' counts the total number of writes propagated from a neighbor Nb.

```
void ADR(Req, Id)
{
        switch (Req)
        {
         case 'r' :     ADR_r(Id); /* a read request */
                        break;
         case 'w' :     ADR_w(Id); /* a write request */
                        break;
```

```
          case 't' :     ADR_t(Id); /* a scheme change request (time expired) */
                         break;
          case 'x' :     ADR_x(Id); /* an exit request */
                         break;
          case 'j' :     ADR_j(Id); /* a join request */
                         break;
          case 's' :     ADR_s(Id); /* a switch request */
                         break;
          default  :     fprintf(stderr, "Parameter err in ADR(). \n");
                         break;
        }
}

void ADR_r(Id)  /* This procedure is called when a read request is issued
                   locally (Id == My->Id), or a message from a neighbor is
                   received indicating that the neighbor wants to read the
                   object (in this case, Id will be the neighbor's Id).      */
{
        if (I have a replica of the object)
          { retrieve the object from the local database;
            send the object to processor Id;
            My->SumR += 1;
            if (Id != My->Id)
                Nb->NoR += 1 (for the neighbor Nb which submitted the read);
          }
        else /* I do not have a local replica */
          { find a neighbor Nb, such that Nb->Drt == 1;
            submit the read request to Nb, and wait for the reply;
            after getting the object, send it to processor Id;
          }
}

void ADR_w(Id)
{
        if (I have a replica of the object)
          { update the local replica;
            My->SumW += 1;
            if (Id == My->Id) /* the write is issued by myself */
              { for each neighbor Nb, do
                  { if (Nb->Drt == 1)
                      propagate the write to Nb;
              }   }
            else /* the write is propagated from elsewhere */
              { find the neighbor NB such that NB has the id 'Id';
                NB->NoW += 1;
                for each neighbor Nb, do
                  { if (Nb->Drt == 1) and (Nb != NB)
```

```
                       propagate the write to Nb;
          }   }   }
        else /* I do not have a local replica */
          { find a neighbor Nb so that Nb->Drt == 1;
            submit the write request to Nb;
          }
}

void ADR_t(Id)
{
        if ( I am in the Singleton replication scheme )
        /* namely, I have a local replica of the object
                      and Nb->Drt == 0 for all neighbors */
          { if (Expansion() == 0) /* (Expansion() == 0) if the expansion test failed */
              Switch();
          }
        else if ( I am an R_bar_neighbor processor )
        /* namely, I have a local replica of the object
              and Nb->Drt == 0 for at least one neighbor Nb */
          { if ( (Expansion() == 0) && ( I am an R_fringe processor ) )
              /* I am an R_fringe processor if and only if I have a replica
                 of the object, and (Nb->Drt == 1) for exactly one neighbor Nb */
              Contraction();
          }
        else if ( I am an R_fringe processor )
                Contraction();

        My->SumR = My->SumW = 0;    /* reset all the counters */
        for each neighbor Nb, do
            Nb->NoR = Nb->NoW = 0;
}

int Expansion()   /* This procedure returns '1' if the expansion test succeeds,
                     namely at least one of the neighbor joins the replication
                     scheme in the procedure call, otherwise it returns '0' */
{ int succeed = 0;

        for each neighbor Nb
          if (Nb->Drt == 0)
            { if (Nb->NoR > My->SumW - Nb->NoW)
                { send a "join" message to Nb, along with a replica of the object;
                 /* the message is processed by the ADR_j procedure */
                  Nb->Drt = 1; succeed += 1;
            }   }
        return(succeed);
}
```

```
int Switch()       /* This procedure returns ‘1’ if the switch test succeeds,
                      namely the singleton replication scheme switches to a
                      neighbor, otherwise it returns ‘0’                  */
{ int succeed = 0;

        while ( (succeed == 0) && (there exists an unmarked neighbor Nb) )
          { if ( 2 * (Nb->NoR + Nb->NoW) > My->SumR + My->SumW )
                { Send to Nb the message "join as a singleton scheme",
                  along with a replica of the object;
                 /* the message is processed by the ADR_s procedure */
                  Nb->Drt = 1;
                  delete the local replica; deallocate counters;
                  succeed = 1;
                }
            else
                  mark Nb;
          }
        return(succeed);
}

int Contraction()  /* This procedure is called by an R_fringe processor.
                      Observe that a pair of processors which constitute
                      the whole replication scheme may call this procedure
                      at the same time for contraction. Thus, special care
                      needs to be taken to prevent the contraction of both */
{
        find a neighbor Nb such that (Nb->Drt == 1);
        if ( Nb->NoW > My->SumR )
          { send the "exit" message to Nb, and wait for a response;
            if ( the response is "Yes, you may exit" )
                delete the local replica; deallocate the counters;
            else if ( an "exit" request from Nb is received )
                { if ( My->Id < Nb->Id )
                    { delete the local replica; deallocate the counters;
                      send a message to Nb saying "No, you may not exit";
                    }
                  else
                    { send a message to Nb saying "Yes, you may exit";
                      Nb->Drt = 0;
                }   }
            else if ( the response is "No, you may not exit" )
                Nb->Drt = 0;
          }
}

void ADR_x(Id)  /* This procedure is called when an ’exit’ message is received from
                   an R_fringe neighbor ‘Id’.  A processor q will grant such request
                   if q is not executing the contraction test at the same time. */
{
        send a message to the neighbor ‘Id’ saying "Yes, you may exit";
        Id->Drt = 0;
}

void ADR_j(Id)  /* This procedure is called when a ‘join’ message is received from
                   neighbor ‘Id’ */
{
        save the object in the local database;
        allocate the counters and initialize them to 0;
}

void ADR_s(Id)  /* This procedure is called when a ‘join as a singleton scheme’
                   message is received from neighbor ‘Id’  */
{
        save the object in the local database;
        Id->Drt = 0;
        allocate the counters and initialize them to 0;
}
```