Object Allocation in Distributed Databases and Mobile Computers*

Yixiu Huang, Ouri Wolfson
Electrical Engineering and Computer Science Department
University of Illinois
Chicago, Illinois 60680

Abstract

This paper makes two contributions. First, we introduce a model for evaluating the performance of data allocation and replication algorithms in distributed databases. The model is comprehensive in the sense that it accounts for I/O cost, for communication cost, and for limits on the minimum number of copies of the object (to ensure availability).

The second contribution of this paper is the introduction and analysis of an algorithm for automatic dynamic allocation of replicas to processors. Using the new model, we compare the performance of the traditional read-one-write-all static allocation algorithm, to the performance of the dynamic allocation algorithm. As a result, we obtain the relationship between the communication cost and I/O cost for which static allocation is superior to dynamic allocation, and the relationships for which dynamic allocation is superior.

1. Introduction

1.1 Motivation

We are presently moving 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 be geographically distributed world-wide. For example, in electronic publishing a document (e.g. a newspaper, an article, a book) will be co-authored by multiple users and read by many, 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 annotated by multiple hospitals and read by many. In the mobile communication environments of the future (see [7, 15]) 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, an object is usually replicated, i.e. stored in the local database of multiple processors, for performance and availability. The object is accessed as a result of requests issued by the processors in the system. Each request has a cost in terms of both, communication and I/O. For example, a write request of the object usually results in the transmission of the object to multiple processors (communication cost), and each one of these processors outputs the object to its local database that resides on disk (I/O cost).

In mobile computing, the communication costs of servicing access requests translates into significant consumption of preciously limited battery power. Additionally, the network-provider will charge a fee for each (wireless transmission) message sent to (or received from) the mobile processor. In an ethernet environment, a higher communication cost implies a higher load on the network, which, in turn, implies a higher probability of contention on the communication bus, and a higher response time; a higher I/O cost also negatively affects the response time. Therefore, it is important to optimize the cost of servicing a set of access requests.

1.2 The cost of access requests

Assume that the I/O cost of servicing an access request, i.e. the cost of inputting (outputting) of the object from (to) the local database, is c_{io}. In terms of communication, there are two types of messages associated with servicing access requests, *control-messages* and *data-messages*. An example of a control message is a request message, in which a processor p (that does not have a copy of the object) requests another processor q to transfer a copy of the object to p. A data message is a message in which the object is transmitted between processors. Control messages are usually much shorter than data messages, therefore, different costs are associated with the two types of messages. The communication cost of a control-message is c_c, and the communication costs of a data-message is c_d.

We distinguish between the *stationary-computing*

*This research was partially supported by NSF grant IRI-90-03341. It appears in the **Proceedings of the 10-th International Conference on Data Engineering**, pp. 20-29, February 1994, Houston, Texas

(SC) cost model, in which $c_{io} > 0$, and the *mobile-computing* (MC) cost model, in which $c_{io} = 0$. The reason for this is that in a mobile computing environment the user is billed for every message, whereas I/O does not carry an out-of-pocket expense.

In this paper we analyze the cost of servicing a set of read-write requests for a single object. This set is usually ordered by some concurrency-control mechanism, such that each read request accesses the most recent version of the object (i.e. the version written by the most recent write request). The total cost (communication + I/O) of servicing a read or write request depends on the *allocation scheme* of the object, namely the set of processors that store the most recent version of the object in their local databases.

Loosely speaking, the larger the allocation scheme the smaller the cost of an average read-request, and the bigger the cost of an average write request. Consider, for example, the cost of servicing a read request issued by a processor s. If s is in the allocation scheme, then the cost of servicing the read request is c_{io}, i.e. the I/O cost of inputing the latest version of the object from the local database. If s is not in the allocation scheme, then the cost of servicing the read request is $c_c+c_{io}+c_d$ (c_c is the cost of the request message sent from s to some processor y in the allocation scheme, c_{io} is the cost of inputting the object from the local database at y, and c_d is the cost of transmitting the object from y to s).

1.3 Static versus dynamic allocation

The allocation scheme of an object is either dynamic or static, namely it either changes as the read-write requests are executed, or it remains fixed.

Given a priori information on the sequence of read-write requests and on the originating processor of each request, dynamic allocation is more efficient. For example, consider the sequence of requests $r^1r^1r^2w^2r^2r^2r^2$ (r^1 denotes a read request issued by processor 1, w^2 denotes a write request issued by processor 2, etc.). Suppose that the initial allocation scheme is $\{1\}$, namely there is a single copy of the object that is stored in the local database of processor 1. Suppose that the fourth request, i.e. the write request issued by 2, invalidates the copy of the object at processor 1 (using a control-message) and outputs the object only in 2's local database, making processor 2 the single member of the allocation scheme. Then the next three read requests, issued by 2, can access the object in the local database. Clearly, with this dynamic allocation that changes the allocation scheme from $\{1\}$ to $\{2\}$, the total cost (communication + I/O) of executing the seven requests is lower than if the allocation scheme had stayed fixed at $\{1\}$.

1.4 Manual versus automatic allocation, and offline versus online allocation

Dynamic allocation of objects may be either manual, namely determined by the user, or it may be automatic, namely determined by the system.

A user may want the system (rather than itself) to determine whether or not the latest copy of the object should be allocated to its processor, for two reasons. First, she may not know the future expected frequency of reading the object at her processor (since, for example, the object may be read in response to calls from customers). Second, she may not know the frequency at which the object is updated at other processors.

In this paper we discuss automatic dynamic allocation. The allocation is performed by Distributed Object Management (DOM) algorithms. A DOM algorithm maps each request to a set of processors that execute the request, and it determines the allocation scheme of the object at any point in time. We distinguish between online and offline DOM algorithms. An offline DOM algorithm, when servicing a request, has a priori knowledge of all the future requests. An online algorithm does not do so. The example in subsection 1.3 demonstrates that dynamic allocation is more efficient for offline algorithms. In this paper we study in detail static and dynamic allocation of online algorithms.

1.5 Paper organization

The rest of the paper is organized as follows. In section 2 we introduce the dynamic allocation algorithm and present a summary of our results. In section 3 we present the model, we formalize the cost function, and we define distributed object management algorithms. In section 4 we define the static allocation algorithm (SA) and the dynamic allocation algorithm (DA) in terms of the proposed model, and we analyze them. In section 5 we compare our work with relevant literature. Finally, in section 6 we summarize our results and discuss their extension to other models.

2 Results summary

In this paper we achieve two objectives. First, we propose a model and a methodology to analyze and compare online DOM algorithms. Basically, the proposed methodology to show that DOM algorithm A is *superior* to DOM algorithm B, is as follows. First, show that A is α-competitive, i.e., there exists a constant $\alpha > 1$ such that for any sequence s of read-write requests: (the cost of A on s) $\leq \alpha \times$ (the cost of the optimal offline DOM algorithm on s). Then, show that B is not α-competitive, i.e., that there are sequences of requests for which (the cost of B) $> \alpha \times$ (the cost of the optimal offline DOM algorithm). This means that in the worst case the cost of A is lower than the cost of B. We chose this approach since, in comparing algorithms, often the worst-case performance of an algorithm is a very good indicator of its average case performance. In other words, if algorithm A is superior to algorithm B in the worst case, then it is usually superior on average. On the other hand, worst case performance is much more amenable to analysis.

The second objective of this paper is to introduce a new DOM algorithm, namely Dynamic Allocation (DA), and to compare it with the traditional read-one-write-all Static Allocation (SA) DOM algorithm. The comparison is performed using the methodology discussed above. Both the SA and the DA algorithms are on-line, in the sense that they do not know the sequence of the requests a priori.

In addition to performance, a DOM algorithm is also affected by availability considerations. We assume that the SA and DA algorithms are subject to the following constraint "the allocation scheme must be of size which is at least t"; in other words, $t > 1$ is some integer representing a threshold below which the number of copies must not drop.

Next we will precisely define the SA and DA algorithms, and then we will discuss the results of the comparison between them.

Static Allocation: At all times, SA keeps a fixed allocation scheme, Q, which is of size t, and SA performs read-one-write-all. Namely, in response to a write request issued by a processor p, SA sends the object from p to each one of the processors in Q. In turn, each processor of Q outputs the object in its local database. In response to a read request issued by a processor p, SA requests a copy of the object from some processor $y \in Q$; in turn, y retrieves the replica from its local database and sends it to p. □

Dynamic Allocation: The DA algorithm selects a priori a set of processors F, of size $(t-1)$, and a processor $p \notin F$. The initial allocation scheme consists of $F \cup p$. At any point in time the processors of F have the latest version of the object. A read request from a processor of the allocation scheme is satisfied by inputing the object from the local database. A read request from a processor q outside the allocation scheme is satisfied by requesting a copy of the object from some processor u of F; q saves the object in its local database (thus joining the allocation scheme), and u remembers that q is in the current allocation scheme by entering q in u's "join-list". A write request from some processor q outputs the object to the local database at q, and to the local database of the processors in F. If $q \in F$, then a copy of the object is also sent to p, in order to satisfy the availability constraint. Additionally, the write request results in the invalidation of the copies of the object at all the other processors (since their version is obsolete). This is done as follows. Each processor of F sends an 'invalidate' control-messages to the processors in its "join-list", except for q. □

In mobile computing, assume that the mobile processors are connected to a base station which has a processor and a local database. Then a natural choice for t is 2, with F (in DA) consisting of the base-station processor. Then each write from a mobile processor will be performed locally, as well as propagated to the base-station. The base station will invalidate the copies at all the other mobile processors.

We propose that the DA algorithm handles failures by resorting to quorum consensus with static allocation (see [14, 25]) when a processor of the set F fails. The transition occurs using the missing writes algorithm (see [12, 6]). Details are omitted due to space limitations.

The comparison results, discussed next, are obtained for the SA and DA algorithms operating in the normal mode (namely, in the absence of failures). For simplicity, in the stationary-computing model we normalize the cost by taking c_{io}=1. For high-bandwidth, high speed networks the communication cost (i.e. c_c and c_d) may be arbitrarily close to zero. We obtain the following results in this model. We show that the SA algorithm is $(1+c_c+c_d)$-competitive. Then we show that the DA algorithm is $(2+2\cdot c_c)$-competitive in general, and that the DA algorithm is $(2+c_c)$-competitive when $c_d > 1$. Interestingly, these competitiveness factors are independent of the integer t which limits the minimum number of copies in the system. Then we show that the SA algorithm is not γ-competitive for any $\gamma < 1 + c_c + c_d$; therefore, when $c_d - 1 > 0$ (i.e., when the cost of a data-message is higher than the I/O cost) the DA algorithm is superior to the SA algorithm, since in this case $1 + c_c + c_d > 2 + c_c$. Then we show that the DA algorithm is not 1.5-competitive; therefore, when $c_d + c_c < 0.5$ (i.e., when sum of the data-message cost and the control-message cost is lower than 0.5) the SA algorithm is superior to the DA algorithm, since in this case $1 + c_c + c_d < 1.5$.

The results of the comparison are summarized in figure 1. The figure indicates the area on the c_c and c_d plane for which the SA algorithm is superior, and the area for which the DA algorithm is superior. The area in which $c_c > c_d$ is marked "Cannot be true", since a data message cannot be less costly than a control message. The reason for this is that the control message includes the object-id and operation (read, write, or invalidate) fields only, whereas the data message includes the object-id and operation (read, write, or invalidate) fields, as well as the object content. The area marked "Unknown" represents the c_c and c_d values for which it is currently unknown whether the DA algorithm is superior to the SA algorithm or vice versa. The reason for this uncertainty is the gap between the upper and lower bound on the competitiveness of the DA algorithm.

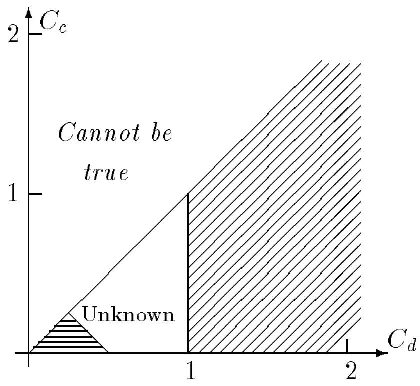

Figure 1.

(Stationary-computing cost model)

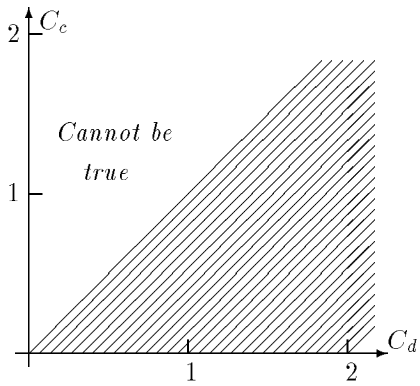

Figure 2.

(Mobile-computing cost model)

Then we consider the MC model. In this model the I/O cost is zero, the data-message cost is c_d and the control-message cost is c_c. We showed that in a mobile computing environment the DA algorithm is $(2+3\cdot\frac{c_c}{c_d})$-competitive, whereas the SA algorithm is no longer competitive. Thus, in such an environment the DA algorithm is strictly superior to the SA algorithm. Figure 2 illustrates this dominance of DA over SA.

3 The Model

3.1 Schedules and allocation schemes

A *distributed system* is a set of interconnected processors. The *local database* at a processor is a set of objects that are written on stable storage at the processor. Transactions operate on the object by issuing read and write requests. In this paper we address the allocation of a single object. A *schedule* is a finite sequence of read-write requests to the object, each of which is issued by a processor. For example, $\psi_0 = w^2r^4w^3r^1r^2$ is a schedule in which the first request is a write from processor 2, the second request is a read from processor 4, etc.

In practice, any pair of writes, or a read and a write, are totally ordered in a schedule, however, reads can execute concurrently. Our analysis using the model applies almost verbatim even if reads between two consecutive writes are partially ordered.

Each write request in a schedule creates a new *version of the object.* Given a schedule, the *latest version of the object at a request q* is the version created by the most recent write request that precedes q.

Each request is mapped to a set of processors, namely the *execution set* of the request. Intuitively, for a write request the execution set is the set of processors which output the written object to their local database. For a read request the execution set is the set of processors from which the object is retrieved to satisfy the request.[1]

An *execution schedule* is a schedule of requests, each with its associated execution set. For example, $\xi_0 = w^2\{2,3\}$, $r^4\{1,2\}$, $w^3\{2,3\}$, $r^1\{1,2\}$, $r^2\{2\}$ is an execution schedule of ψ_0, where $\{2,3\}$ is the execution set of the first request w^2, $\{1,2\}$ is the execution set of the second request r^4, etc.

A read request brings the object to main memory of the processor that issued the read request, namely the *reading processor.* In case of dynamic allocation, the reading processor, s, may also store the object in its local database, in order to enable future reads at s to be local. A read request that results in saving the object in the local database is called a *saving-read.*

An *allocation schedule* is an execution schedule in which some reads are converted into saving-reads. A saving-read is denoted by an underlined read in the allocation schedule. For example, by converting the fourth request in ξ_0 into a saving-read, we obtain an allocation schedule $\phi_0 = w^2\{2,3\}$, $r^4\{1,2\}$, $w^3\{2,3\}$, $\underline{r}^1\{1,2\}$, $r^2\{2\}$. Observe that at the end of this allocation schedule the object is stored in the local databases of processors $\{1,2,3\}$.

The *initial allocation scheme* is a set of processors.

[1] Generally, a read request does not necessarily access a single copy of the object. For example, in quorum consensus (see [14, 25]), a read request retrieves a number of copies that have a read-quorum (and then discards all of them, except the one with the most recent time-stamp).

Intuitively, it is the set of processors that have the object in their local database before the schedule begins. Given an allocation schedule, the *allocation scheme at a request* q is the set of processors that have the latest version of the object in the local database right before q is executed, but after the immediately-preceding request is executed. The allocation scheme at the first request is the initial allocation scheme. For example, consider the allocation schedule ϕ_0 above, and the initial allocation scheme $\{3,4\}$. The allocation scheme at the first request w^2 is $\{3,4\}$; the allocation scheme at the second, third, and fourth requests is $\{2,3\}$; the allocation scheme at the fifth request is $\{1,2,3\}$. We define a *data processor at a request* to be a processor which belongs to the allocation scheme at the request, and a *non-data processor at a request* to be a processor which is not a data processor at the request.

In the introduction we mentioned a t-available constraint. Formally, for an integer t, an allocation schedule satisfies the *t-available constraint* if the allocation scheme at every request is of size which is at least t. For simplicity we shall assume that t is at least 2. We shall also assume that t is smaller than the total number of processors in the network, otherwise each write must be propagated to all the processors of the network, and the problems addressed in this paper become trivial.

Next we define a legal allocation schedule. Intuitively, it is an allocation schedule in which the execution set of every read request contains at least one data processor. In other words, the read request accesses the object in some processor that has a latest version of the object in its local database. Formally, given an initial allocation scheme, a *legal allocation schedule* is an allocation schedule in which the execution set of every read request has a non-empty intersection with the allocation scheme at the read request. For example, the allocation schedule ϕ_0 above is legal. However, ϕ_0 will be illegal if we change the execution set of the last request r^2 from $\{2\}$ to $\{4\}$.

A legal allocation schedule ϕ and a schedule ψ *correspond* to each other if ψ is obtained from ϕ by eliminating the execution sets, and turning every saving-read to a read. For example, ϕ_0 and ψ_0 correspond to each other.

3.2 The cost of requests in stationary-computing

We define the cost of inputting/outputting the object to the local database at a processor to be one unit. We denote by c_d the cost of transmitting a data-message from one processor to another, and we denote by c_c the cost of transmitting a control-message from one processor to another. Observe that this definition assumes a homogeneous system, in which the data-message between every pair of processors costs c_d, the control-message between every pair of processors costs c_c, and the I/O cost is identical at all the processors.

Given an allocation schedule, the cost of a request q, denoted $COST(q)$, is defined as follows. Suppose that $q = r^i$, i.e., q is a non-saving-read from processor i, and suppose that X is its execution set. In other words, the read request retrieves the object from the local database at the processors in X. If $i \in X$, then $COST(r^i) = (|X|-1) \cdot c_c + |X| + (|X|-1) \cdot c_d$. Intuitively, $(|X|-1) \cdot c_c$ is the cost of submitting a read request to $(|X|-1)$ processors, $|X|$ is the I/O cost of inputing the object from the local database at $|X|$ processors, and $(|X|-1) \cdot c_d$ is the communication cost of sending the object from $(|X|-1)$ processors to i. Otherwise, i.e., if $i \notin X$, then $COST(r^i) = |X| \cdot (c_c + 1 + c_d)$.

Suppose that $q = \underline{r}^i$, i.e., q is a saving-read from processor i, and suppose that X is its execution set. Then,

$$COST(\underline{r}^i) = \begin{cases} (|X|-1) \cdot c_c + |X| + \\ \qquad (|X|-1) \cdot c_d + 1 & \text{if } i \in X \\ |X| \cdot (c_c + 1 + c_d) + 1 & \\ & \text{otherwise} \end{cases}$$

The cost of a saving-read is higher (by 1) than that of a non-saving read, to account for the extra I/O cost to save the object in the local database at i.

Suppose that $q = w^i$, i.e., q is a write request from processor i, suppose that X is its execution set, and suppose that Y is the allocation scheme at w^i. Then,

$$COST(w^i) = \begin{cases} |Y \setminus X| \cdot c_c + (|X|-1) \cdot c_d + |X| \\ \qquad \text{if } i \in X, \\ |Y \setminus X \setminus \{i\}| \cdot c_c + |X| \cdot (c_d + 1) \\ \qquad \text{otherwise} \end{cases}$$

The explanation for the write cost is as follows. As part of servicing the write request that creates a new allocation scheme of the object, an 'invalidate' control message has to be sent to all the processors of $Y \setminus X$, i.e. the processors at which the copy of the object is obsolete; these are the processors of Y (the old allocation scheme) that are not in X (the new allocation scheme). Thus, the first term in the write-cost. The other terms are the cost of communicating the object to the processors of the new allocation scheme, and the cost of outputting the object to the local database at these processors.

For an allocation schedule $\xi = o_1 X_1 \ldots o_n X_n$ and an initial allocation scheme I, where o_i is either a read or a saving-read or a write, X_i is the execution set of o_i, we define the *cost of the allocation schedule* ξ, denoted by $COST(I, \xi)$, to be the sum of all costs of the read-write requests in the schedule, i.e.,

$$COST(I, \xi) = \sum_{i=1}^{n} COST(o_i)$$

3.3 The cost of requests in mobile-computing

Assume that the cost of inputting/outputting the object to/from the local database is zero. We still assume that the data message cost is c_d, and the control message cost is c_c.

Given an allocation schedule, suppose that o^i is a request in the schedule, Y is the allocation scheme at o^i, and X is its execution set. The cost of request o^i, denoted by $COST(o^i)$, is defined as follows.

$$COST(o^i) = \begin{cases} (|X|-1)\cdot(c_c+c_d) & \text{if } o \text{ is a read, and } i \in X \\ |X|\cdot(c_c+c_d) & \text{if } o \text{ is a read, and } i \notin X \\ |Y\setminus X|\cdot c_c + (|X|-1)\cdot c_d & \text{if } o \text{ is a write, and } i \in X \\ |Y\setminus X\setminus\{i\}|\cdot c_c + |X|\cdot c_d & \text{if } o \text{ is a write, and } i \notin X \end{cases}$$

The reasoning for this cost function is identical to the one in the previous subsection. Observe that the cost of a read request executed only locally is zero. Observe also that the cost of a saving-read does not differ from that of a non-saving read.

3.4 Distributed object management algorithms

A *distributed object management* (DOM) algorithm (say A) is an algorithm which, given a schedule ψ and an initial allocation scheme, produces a corresponding legal allocation schedule. We call this legal allocation schedule the *A-allocated schedule* of ψ and denote it by $las_A(\psi)$. For an integer $t > 1$, a DOM algorithm is *t-available constrained* if every legal allocation schedule that it produces satisfies the t-available constraint.

We define the *cost of the algorithm A on the schedule ψ* with initial allocation scheme I to be the cost of the A-allocated schedule, i.e., $COST(I, las_A(\psi))$, and sometime we denote it by $COST_A(I,\psi)$. A t-available constrained DOM algorithm is *cost-optimal* if, for every schedule and for every initial allocation scheme, the cost of the legal allocation schedule that it produces is minimum among all the corresponding legal allocation schedules that satisfy the t-available constraint.

Now we define a special type of DOM algorithm, called an on-line DOM algorithm. An online DOM algorithm consists of a sequence of online steps. An *online step* takes as input an initial allocation scheme I, a legal allocation schedule ϕ, and a request q; it produces a new legal allocation schedule η that consists of the prefix ϕ followed by q (that could be underlined) with an execution set associated with q. Intuitively, an online step appends q to ϕ, associates an execution set with q, and, if q is a read, possibly turns it into a saving-read.

An *online DOM* algorithm is one that produces the legal allocation schedule by "feeding" the requests of the schedule, sequentially, into an online step. Specifically, it provides the online step with the initial allocation scheme, the *las* produced in the previous invocation of the online step, and the next request in the sequence. For example, given an initial allocation scheme I, the SA online algorithm produces a legal allocation schedule by feeding the requests of a schedule into the following online step, that we call *Static Allocation Online Step* (SAOS). SAOS ignores the input legal allocation schedule, it associates with every write the execution set I, and it associates with every read a singleton that is some member in I. Intuitively, every read request accesses the object at some processor of I, and every write request is propagated to all processors of I.

An *offline DOM* algorithm is one that is not online, namely, it can consider future requests when determining the execution set of the current one.

4 Analysis of on-line algorithms

In this section we define competitiveness as a performance measure for online DOM algorithms, then we present the static and dynamic on-line DOM algorithms, and then we analyze them in terms of competitiveness. The SA and DA algorithms defined in the introduction are t-available constrained, where t is the size of the initial allocation scheme. In other words, the SA and DA algorithms defined in the introduction become t-available constrained DOM's by providing them with an initial allocation scheme of size t.

4.1 Competitiveness

Competitiveness is a widely accepted way to measure the performance of an on-line algorithm (see [3, 13, 16, 23]). Intuitively, an α-competitive online DOM algorithm is one which, for any schedule, costs at most α times as much as the minimum cost. Formally, a t-available constrained DOM algorithm, A, is α-competitive if there are two constants β and α, such that: for an arbitrary initial allocation scheme I and an arbitrary schedule ψ, $COST_A(I,\psi) \le \alpha \cdot COST_{OPT}(I,\psi) + \beta$, where OPT is an offline t-available constrained DOM algorithm that produces the minimum cost legal allocation schedule for any input. We call α the *competitiveness factor* of the algorithm A.

4.2 Static and dynamic allocation in stationary computing

4.2.1 The Static Allocation Algorithm

The SA algorithm was presented in the introduction. In terms of our model, SA maps a schedule into the following legal allocation schedule. Suppose that the initial allocation scheme is Q (of size t).

SA Algorithm

1. For a read request r^i, if $i \in Q$, then SA associates with it the execution set $\{i\}$. Otherwise, i.e., if $i \notin Q$, SA associates with it the execution set that consists of an arbitrary processor in Q.
2. SA associates with every write the execution set Q.

Theorem 1: For any integer t, the SA algorithm is $(1 + c_c + c_d)$-competitive. [2]

The competitiveness factor of SA is *tight* in the following sense. For any constants γ $(< 1 + c_c + c_d)$, and $\beta \geq 0$, we can construct a schedule ψ, such that $COST_{SA}(Q, \psi) > \gamma \cdot COST_{OPT}(Q, \psi) + \beta$. Thus, we have the following.

Proposition 1: For any constant γ and for any integer t, if $\gamma < 1 + c_c + c_d$, then the algorithm SA is not γ-competitive.

4.2.2 The dynamic allocation algorithm

Here we describe the DA algorithm presented in the introduction in terms of our model. DA selects a set of processors F of size $(t - 1)$ and a processor $p \notin F$, such that $F \cup \{p\}$ is the initial allocation scheme. DA maps a schedule to an legal allocation schedule in the online steps as follows.

DA Algorithm

1. For a read request r^i, if i is a data processor, then DA associates with it the execution set $\{i\}$. Otherwise, i.e., if i is a non-data processor, then DA associates with it the execution set that consists of a processor in F, and DA turns this read into a saving read $\underline{r}^i$.
2. For a write request w^j, if $j \in F \cup \{p\}$, then DA associates with it the execution set $F \cup \{p\}$. Otherwise, i.e., if $j \notin F \cup \{p\}$, then DA associates with it the execution set $F \cup \{j\}$.

Theorem 2: For any integer t, the DA algorithm is $(2 + 2 \cdot c_c)$-competitive.

This theorem gives an upper bound of the competitiveness factor of DA, namely the bound $(2 + 2 \cdot c_c)$ may not be tight. We say so because we did not find a schedule for which the cost of DA is exactly $(2+2 \cdot c_c)$ $\times$ (the optimal cost) and we did not find a constant α $(\alpha < 2 + 2 \cdot c_c)$ so that DA is α-competitive. Though we did not lower the competitiveness factor in general, we obtained that this factor can be lowered by c_c under the condition $c_d > 1$, i.e., the following.

Theorem 3: For any integer t, if $c_d > 1$, then the DA algorithm is $(2 + c_c)$-competitive.

Next, we obtain a lower bound of 1.5 for the competitiveness factor of DA, namely:

Proposition 2: For any constant γ and for any integer t, if $\gamma < 1.5$, then the algorithm DA is not γ-competitive.

4.3 Static and dynamic allocation in mobile computing

Remember that in mobile computing with wireless communication the cost of servicing requests is dominated by the communication cost, i.e. the I/O cost is zero. We will first show that the static allocation algorithm, SA, is not competitive. Then we show that the dynamic allocation algorithm, DA, remains competitive.

Proposition 3: The SA algorithm is not competitive.

Theorem 4: The DA algorithm is $(2 + 3 \cdot \frac{c_c}{c_d})$-competitive.

Observe that since $c_c \leq c_d$, the competitiveness factor of DA is at most 5.

5 Relevant work

5.1 Dynamic allocation in distributed systems

In [27, 17] we considered the cost and time of dynamic allocation. However, the model there ignores the I/O cost and availability constraints (i.e., they consider only communication). Also, the model in [27] is dependent on the communication network having a tree topology. The present paper removes this dependency.

Additionally, the algorithms developed in [27] are convergent rather than competitive. This means the following. Assume that the pattern of access to each object is generally regular. For example, during the first two hours processor x executes three reads and one write per second, processor y executes five reads and two writes per second, etc.; during the next four hour period processor x executes one read and one write per second, processor y executes two reads and two writes, and so on. Then, a convergent algorith-

[2] Proofs of the theorems (1-4), and of propositions (1-3) are omitted due to space limitations

m will move to the optimal allocation scheme for the global read-write pattern during the first two hours, then it will converge to the optimal allocation scheme for the global read-write pattern during the next four hours, etc.

Competitiveness and convergence are two criteria for evaluating online algorithms. A competitive algorithm may not converge to the optimal allocation scheme when the read-write pattern stabilizes, and a convergent algorithm may unboundedly diverge from the optimum when the read-write pattern is irregular. A competitive online algorithm is more appropriate for chaotic read-write patterns, in which the past access pattern does not provide any indication to the future read-write pattern. In contrast, a convergent online algorithm is more appropriate for regular read-write patterns (although it is not guaranteed that a convergent algorithm will always outperform a competitive one, even for regular read-write patterns).

In [28] we developed a different algorithm that is also convergent rather than competitive. In [17] we developed a competitive algorithm, called CDDR, for a model that ignores the I/O cost and the availability constraints. The DA algorithm presented here is totally different than CDDR. Furthermore, the CDDR algorithm is not competitive when the I/O cost and the availability constraints are taken into consideration.

There has also been work addressing dynamic object allocation algorithms in [3]. However, the model there does not allow concurrent requests, and it requires centralized decision making by a processor that is aware of all the requests in the network. In contrast, our algorithms are distributed, and allow concurrent read-write requests. Additionally, the model in [3] also concentrates exclusively on communication, and it does not consider I/O cost and availability constraints.

The two main purposes of replicated data are increased availability and performance. No other work of which we are aware, quantitatively compares static and dynamic allocation, while considering both purposes of data replication and treating I/O and communication costs in a unified model.

Static allocation was studied in [26, 9], however not from an online-algorithmic point of view. In other words, these works address the following file-allocation problem. They assume that the read-write pattern at each processor is known a priori and they find the optimal static allocation scheme. However, works on file-allocation problem do not quantify the cost penalty if the read-write pattern is not known. In contrast, in this paper we do so.

5.2 Caching and virtual memory

In the computer architecture and operating systems literature there are studies of two subjects related to dynamic allocation, namely caching and distributed virtual memory (see [2, 1, 4, 5, 8, 10, 11, 19, 16, 18, 20, 21, 22, 24, 29]). In these methods, when a processor issues a read to a shared page that is not in its memory, a read-page-fault is triggered and the page-fault interrupt handler requests the page from another processor; when received, the page is cached locally. There are various ways of handling writes, but one, called write-invalidation, is closely related to our DA algorithm. The write-invalidation method calls for a processor that issues a write to a cached page to invalidate all other cached copies of the page before updating its own.

However, there are several important differences between Caching and Distributed Virtual Memory (CDVM) on one hand, and replicated data in distributed systems on the other. Therefore, our results have not been obtained previously. First, CDVM methods have been devised for performance enhancement only, whereas in databases replication has two objectives, enhanced performance and availability. Consequently, CDVM methods do not have a threshold on the minimum number of copies, whereas in this paper we assumed such a threshold.

Second, the size of the cache is limited. Thus important issues in CDVM literature are cache utilization, and page replacement strategy (e.g. LRU, MRU, etc.), namely which page to replace in the cache when the cache is full and a new page has to be brought in. In other words, in contrast to replicated data in distributed systems, which may reside on secondary (and even tertiary) storage, in CDVM literature a page is "uncached" not only as a result of writes, but also as a results of limited storage. One may argue whether or not limited storage is a major issue in distributed databases, however, in this paper we assumed that storage at a processor is abundant.

Third, the appropriate cost and performance measures for replicated data are different from cost and performance measures for CDVM. For example, when a page that is read is found in the cache, no I/O cost is incurred. On the other hand, even when an object is replicated at a processor, it may reside in secondary storage, leading to an I/O cost incurred at the time of read.

Fourth, the architecture assumed in most CDVM methods is bus-based (e.g. [2, 10, 11, 21]). This architecture supports broadcast at the same cost as a single-cast, and on the other hand incurs contention. In contrast, in this paper we assumed point-to-point communication.

The present work is also related to replicated file systems, such as CODA (in [18, 24]). However, these works also assume that in the absence of failures the allocation scheme is fixed.

6 Discussion

6.1 Conclusion

In this paper we addressed the problem of automatic allocation and replication of a single object in a distributed system. Particularly, we considered distributed-object-management (DOM) algorithms. Such an algorithm services read-write requests by mapping the request to a set of processors that will execute it. Additionally, the algorithm determines the allocation scheme of the object, namely the set of processors that store the object in their local database. The allocation scheme may change over time, as read-write requests are processed.

We introduced a model for analyzing the cost of distributed object management algorithms in stationary and mobile computing environments. The difference between the two cost models is that in mobile computing, because of wireless communication charges, the I/O cost becomes insignificant. On the other hand, in stationary computing the I/O may dominate the cost of servicing read-write requests.

We discussed distributed-object-management algorithms on two dimensions. First, the knowledge level of the algorithm. Offline algorithms have a priori knowledge of all read-write requests, whereas online algorithms do not do so; they have to service a request and determine the allocation scheme without knowing the future requests. Offline algorithms were used as a yardstick to evaluate the performance of online algorithms. The second dimension is variation of the allocation scheme. A static algorithm does not vary the allocation scheme while processing reads and writes, whereas a dynamic algorithm does so.

We also introduced the dynamic allocation (DA) algorithm. In DA, whenever a processor reads a copy of the object, it saves the copy in its local database. Then we compared the DA algorithm with the traditional, read-one-write-all, static allocation (SA) algorithm. We showed that in the stationary computing model, the SA algorithm is tightly $(1+c_c+c_d)$-competitive, the DA algorithm is $(2+2 \cdot c_c)$-competitive, and when $c_d > 1$ the DA algorithm is $(2+c_c)$-competitive; c_c is the ratio of the cost of transmitting a control message to the cost of inputting/outputting the object to the local database on secondary storage, and c_d is the ratio of the cost of transmitting the object between two processors to the I/O cost. A DOM algorithm is α-competitive if the ratio of the cost of the algorithm to the optimal cost is at most α, for an arbitrary sequence of read-write requests. These results are summarized in figure 1 in the section 2. The area of figure 1 that is marked "Unknown" represents the c_c and c_d values for which it is currently unknown whether the DA algorithm is superior to the SA algorithm or vice versa. The reason for this uncertainty is that there is a gap between the upper and lower bound on the competitiveness of the DA algorithm. This gap is the subject of future research.

Then we showed that in mobile computing the SA algorithm is not competitive, while the DA algorithm remains competitive. Hence the DA algorithm is strictly superior to the SA algorithm, as indicated in figure 2 of the section 2.

6.2 Application to versioning and other types of requests

The results obtained in this paper are applicable verbatim to the following append-only distributed-database model. Consider a set S of processors, and a sequence of objects generated by these processors, each by a processor. For example, the objects are images transmitted, one per minute, by a satellite. Each one of the images is received at (generated by) some earth-station, and all the stations comprise the set S. Alternatively, the sequence of objects is the subsequence of images that satisfy a certain predicate (e.g., an aircraft is identified in the image).

For reliability, each object must be stored at t or more processors. Each processor of S is also reading the latest object in the sequence at arbitrary points in time.

SA and DA are two possible algorithms for managing the sequence of objects. SA means that there is a fixed set of t processors with a permanent standing-order to receive the latest object; The other processors issue on-demand read requests. DA means that $t-1$ processors have permanent standing-orders; whenever another processor needs the latest version it issues a temporary standing-order. The standing order is invalidated when the next object in the sequence is received.

Acknowledgement

We would like to thank the anonymous referees for their valuable suggestions to improve the presentation of the paper.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence", *Proc. of the 15-th Int'l Symp. on Computer Architecture*, June 1988

[2] J. Archibald and J. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multi-

processors", *ACM Trans. on Computer Systems*, 4:(4), Nov. 1986, pages 273-298

[3] Y. Bartal, A. Fiat, Y. Rabani, "Competitive Algorithms for Distributed Data Management", *24th ACM STOC*, 5/92, Victoria, B.C. Canada.

[4] J.K. Bennett, J.B. Carter and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures", *Proc. of the 17-th Int'l Symp. on Comp. Arch.*, 5/90

[5] J.K. Bennett, J.B. Carter and W. Zwaenepoel, "Munin : Distributed shared memory based on type-specific memory coherence", *Proc. of the 1990 Conference on Principles and Practice of Parallel Programming*, March 1990

[6] P. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems", *Addison-Wesley* (1987)

[7] B. R. Badrinath and T. Imielinski, "Replication and Mobility", *WMRD-II*, Monterey, CA.

[8] M.J. Carey, M.J. Franklin, M. Livny, E.J. Shekita, "Data Caching Tradeoffs in Client-Server B-DMS Architectures", *ACM-SIGMOD '91*

[9] L. W. Dowdy and D. V. Foster, "Comparative Models of the File Assignment Problem', *ACM Computing Surveys*, 14 (2), 1982.

[10] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation", *Proc. of the 15-th Int'l Symp. on Comp. Architecture*, Pages 373-382, June 1988

[11] Susan J. Eggers and Randy H. Katz, "Evaluating the Performance of Four Snooping Cache Coherency Protocols", *Proc. of the 16-th Int'l Symp. on Computer Architecture*, June 1989

[12] Eager, D.L. and K.C. Sevick, "Achieving Robustness in Distributed Database Systems", *ACM-TODS*, 8(3), 354-381, Sept. 1983

[13] A. Fiat, R, Karp, M.Luby, L.A. McGeoch, D.Sleator, N.E. Yong, "Competitive paging algorithms", *Journal of Algorithms*, 12, 1991

[14] D. Gifford. "Weighted Voting for Replicated Data", *Proc. of 7th ACM Symposium on Operation System Principles*, pages 150-162, 1979

[15] T. Imielinski and B. R. Badrinath, "Querying in highly mobile distributed environments", Proc. of the 18th Int'l Conf. on VLDB '92, pp. 41-52

[16] A.R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive Snoopy Caching", *Algorithmica* (1988) 3:79-119

[17] Yixiu Huang, Ouri Wolfson, "A Competitive Dynamic Data Replication Algorithm", *IEEE Proc. of 9th Int'l Conf. on Data Engineering*, 4/93

[18] J.J. Kistler, and M. Satyanarayanan, "Disconnected Operation in the Coda File System", *ACM Trans. on Computer Systems*, 10(1), 2/92

[19] Kai Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors", *Ph. D. thesis*, 9/86, Dept. of Computer Science, Yale University

[20] Kai Li and Paul Hudak, "Memory Coherence in Shared Virtual memory systems", *ACM Trans. on Computer Systems*, 7(4):321-359, Nov. 1989

[21] J. Lee and U. Ramachandram, "Synchronization with Multiprocessor Caches", *Proc. of the 17-th Int'l Symp. on Comp. Architecture*, 5/90

[22] D.J. Makaroff and D.L. Eager, "Disk Cache Performance for Distributed Systems" *Proc. of the 10-th Int'l Conf. on Dist. Comp. Sys.*, 5/90

[23] M. Manasse, L.A. McGeoch, and D.Sleator, "Competitive algorithms for online problems", *Proc. 20th ACM STOC*, pp. 322-333, ACM 1988

[24] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment", *IEEE Trans. on Computers*, 39(4) April 1990, Pages 447-459

[25] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Database", *ACM-TODS*, 4(2):180-209, 1979

[26] O. Wolfson and A. Milo, "The Multicast Policy and Its Relationship to Replicated Data Placement", *ACM TODS*, 16 (1), 1991.

[27] Ouri Wolfson and Sushil Jajodia, "Distributed Algorithms for Dynamic Replication of Data", *Proc. of ACM-PODS*, 1992

[28] Ouri Wolfson and Sushil Jajodia, "An Algorithm for Dynamic Data Distribution", *Proc. of the 2nd Workshop on Management of Replicated Data (WMRD-II)*, 1992, pp. 62-65

[29] Y. Wang, L.A. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", *ACM-SIGMOD '91*, pp. 367-376