Replicated Data Management in Mobile Environments: Anything New Under the Sun?

Daniel Barbará
Matsushita Information Technology Laboratory
2 Research Way, 3rd Floor
Princeton, NJ 08540 USA
daniel@MITL.Research.Panasonic.COM

Hector Garcia-Molina
Stanford University
Department of Computer Science
Stanford, CA 94305 USA
hector@cs.stanford.edu

October 22, 1993

Abstract

The mobile wireless computing environment of the future will contain large numbers of low powered palmtop machines. Replication will be an essential technique in this environment, providing data availability to the system. In a mobile environment it is important to have *dynamic* replicated data management algorithms that allow for instance copies to migrate from one site to another or for new copies to be generated. In this paper we show that such dynamic algorithms can be obtained simply by letting transaction update the *directory* that specifies sites holding copies. Thus we argue that no fundamentally new algorithms are needed to cope with mobility. However, exisiting algorithms may have to be "tuned" for a mobile environment, and we discuss what this may entail. As an illustration, we present a variation of the primary copy algorithm, Primary By Row, that is well suited for migrating copies [1].

Keywords: Distributed Data Bases, replication, mobility, availability.

1 Introduction

The mobile wireless computing environment of the future [IB92] will contain large numbers of low powered palmtop machines, querying databases over wireless channels. The units will often be disconnected due to power limitations, inaccessible communication channels, or as units move between different cells.

The ability to replicate data objects in such an environment will be essential. Object copies are the key to high data availability: when a unit is disconnected it can continue to process objects stored locally. At the same time, replicated data can improve performance: a copy at a nearby or less congested site can be accessed. Thus, we expect copies to be common both

[1]Note to the referees:

- A much shorter version of this paper will appear in the IFIP Conference on Applications in Parallel and Distributed Computing, Caracas, Venezuela, April 1994. That version is limited to 10 pages, and contains around 40% of the words of this manuscript. Essentially, it is a condensed version of the first four sections of this manuscript.
- This paper is intended to be a survey-like paper of previous work in the area and its applicability to mobility environments. The authors understand that the VLDB journal welcomes contributions of this nature

on mobile units as well as on the servers they interact with; these copies will be dynamically created, updated, and destroyed in the course of the system's operation.

There are two fundamental problems related to replicated data in a mobile environment:

- How to manage the replicated data, providing the levels of consistency, durability and availability needed.
- How to locate objects (or copies of them) of interest. In particular, a directory that indicates the location of objects is commonly used. Should this directory may be centralized, partitioned, or replicated? Should the directory be partitioned in such a way that each of the copies knows only about a subset of the participants?

There has been a lot of work done on replicated data management (for surveys see [AGM87, Dav89, BHG87, CP92]), addressing the above problems. In this paper we focus on two questions related to replicated data in a mobile environment:

- Do we need any "new" replicated data management algorithms for mobile computing, or will existing ones suffice? One could argue that in principle mobility does not introduce any fundamental differences with respect to replicated data in a conventional (non-mobile) environment. In both cases one has data copies at multiple sites and the communication network may partition. (Actually, network partitions have been studied extensively [DGMS85].) On the other hand, one can argue that in a mobile environment, parameters are different: the links have limited bandwidth, the frequency of disconnections is high and the sites might know in advance that they will "fail" (disconnect or lose power). This last fact may allow them to migrate functions to other sites, in preparation for a "failure." Thus, maybe there are new management strategies, or at least new variation appropriate for mobile environments.
- Existing replicated data management algorithms are notoriously complex, especially if one wishes to provide high data availability. So, can we describe at a high level these algorithms, giving their various components? In particular, can we identify the components that may need to be tuned or modified for a mobile environment? In [BI92], the authors identify the problem of replication in mobile environments. Some replication schemes are presented, but no solution or taxonomy of choices is offered.

To answer these questions, we start by clarifying the difference between core copies (those that can be updated by user transactions) and cached copies (Section 2). Although the distinction is rather obvious, very different types of algorithms are need to manage each type. A number of existing papers combine both types of algorithms into one (e.g., [CS92]), in our opinion yielding overly complex algorithms. To avoid this, in this paper we focus on core copy management only (Section 3).

In a mobile environment it is important to have the ability to reconfigure the set of replicas, for example, migrating one copy from one site to another, or adding a new copy to a set of copies. A number of such algorithms have been proposed, e.g., [ET86, DB85, Her87, JM87a, JM87b, JM88, P$\hat{8}$6, P$\hat{8}$4]. The key to understanding these algorithms is to explicitly represent the *directory* that specifies the sites holding copies. Then, a reconfiguration is simply a transaction that modifies the directory. Reconfiguration transaction must be processed using the usual concurrency control mechanisms, serializing them with other transactions. This will be discussed in Sections 4 and 5. Mechanisms that do not guarantee serializability are briefly discussed in Section 6.

Viewing reconfigurations in this fashion shows that essentially no new algorithms are needed. Furthermore, we will argue that mobility does not add any new fundamental differences. However, we will argue that the selection of a replication algorithm from the existing menu of choices should be driven by the characteristics of the mobile environment (Section 7). As an illustration we present a variation (or adaptation) of the Primary Copy algorithm that is well suited for migrating copies (see Section 4.5).

2 Core versus cached copies

In many articles in the replicated data literature, researchers have suggested the use of a large number of copies (e.g. [CGP81, WJ92]), and management algorithms that can scale to these large numbers. Before going any further, it is important to clarify that this is not entirely correct. Specifically, we must distinguish between two types of replicas:

- *Updateable copies*: changes to the object may be initiated at the site holding the copy. We will call the updateable copies the *core* copies, and the set of all updateable copies the *core set*.
- *Read-only*: these are cached copies that cannot be modified locally. We will also call the read-only copies the *cached* copies, for reasons that will become apparent soon.

For example, consider an object that represents the position of a taxi cab. This object may have many copies; for instance all dispatching sites may want to be informed of the whereabouts of this cab. However, it only makes sense to have a single updateable copy, mainly the one at the taxicab itself. All updates to this object will originate at the updateable copy, and be propagated to the read-only copies. As a second example, consider an object representing a patient's medical record and medications in use. In this case, we may wish to make only two copies updateable: one at the patient's hospital floor and the other on his or her physician's palmtop machine. Other copies may exist at other locations, e.g., in the laboratory or at insurance companies. But we do not wish to let all holders of these copies initiate updates (e.g., give medications to the patient!).

In the medical records example, note that we may periodically wish to change a read-only copy to a core one. For example, the patient's physician may wish to use a fixed computer in his or her office. In this case, a special protocol (see Section 4) needs to be run to add a new updateable copy to the core set (or to remove a copy from the core set). The key point is that to execute an update, one only has to check (for consistency violations) with other *current* core copies, not with all copies.

The distinction between core and read-only copies is important because of performance. If one wishes to produce a new version of the C compiler, for example, it does not make sense to request an authorization from the thousands of sites that have a copy. Instead, one requests permission (e.g., locks) from a small set of core copies, performs the update, and then propagates the new version of the compiler to the rest of the sites. Since the read-only copies are never used for generating other updates, then their updates can be done asynchronously, using different and much more efficient protocols. This is why we call read-only copies *cached*.

We feel that no real system will have (or need) many updateable copies. We believe the number of copies that can be updated in a system will be small (e.g., between 1 and 5) for a simple reason: it becomes too expensive to update a large number of core copies. (As an example of the study of the cost of data replication, see [BGM82].) Besides, it is difficult to envision an application that needs more than the update availability that 5 copies offer. The number of read-only copies may be much larger, but these copies may not be current, as noted above.

The management of cached copies is orthogonal to that of core copies. To illustrate, assume that the core copies are managed with a read-one-write-all algorithm (see Section 3). In this case, an update transaction that needs to read data first requests read locks (for the objects it will read) at any one of the core copies. When the transaction is ready to update, it requests write locks, for the objects it wishes to modify, at all the core sites. (We will review other core management strategies in Section 3; read-one-write-all is just an example we use here to illustrate the interactions with the cached copies.) Let us assume that the core copies reside at sites S_a, S_b, S_c. Any of the S_i can act as a "server," giving out copies to "clients" that wish a cached copy. (As illustrated in Figure 1.) If, for instance, S_b gives a copy to site C_1, it can use a variety of standard cache management techniques. For instance, S_b can keep a read lock on the object while C_1 has a copy. This ensures that the copy at C_1 is current, since no updates can take place while any of the S_i is read locked. If S_b does wish to allow an update, it can then invalidate the cached copy at C_1. (If we are worried about C_1 and S_b disconnecting, it may be wise for S_b to grant to C_1 a read lock with an expiration time. Site C_1 would have to renew the lock periodically.) As an alternative, the C_1 copy can be kept without a read lock at S_b. In this case, if a transaction at C_1 wishes to read the data and wishes to make sure it is current, then it would set the lock at S_b at that point in time. As a third alternative, the C_1 copy can be read without any lock. This does not guarantee currency, but will probably be acceptable in many cases. (If the data were to be used for an update, the transaction would not be running at C_1 in the first place; it would have gone to one of S_a, S_b, S_c for the update.)

Of course, S_b can issue multiple copies, say at C_1 and C_2. The read lock at S_b would have to be managed accordingly. For example, if we want the copies to be current, then a lock is held at S_b as long as there is one copy that needs to be current.

As a second example, consider a majority voting scheme for the core, where two of the three sites are needed for a read or for a write. In this case, a single core site cannot issue a cached copy (unless it does not need to be current). In this case, a cached copy C_1 would have to have locks at 2 of the three core sites, say S_a, S_b. If either one of these sites informs C_1 (via invalidate message or a lock timeout) that lock is not valid, then copy cannot be considered current.

Incidentally, there are a variety of ways for servers to send new data values to clients. In particular, the server can ship a new copy of the entire database, or only the values that have changed since the cached copy was updated last. Also, as the new values are installed at the client, it is necessary to use some type of local concurrency control to ensure the installation does not interfere with read-only transactions there (i.e., to guarantee consistency of the cached copy).

Since core and cached copy management is relatively independent, it is possible to study algorithms for each separately. Actually, one of the reasons replicated data papers are confusing is that they describe management of both types of data at once. To avoid this, in the rest of this paper we ignore cached copies.

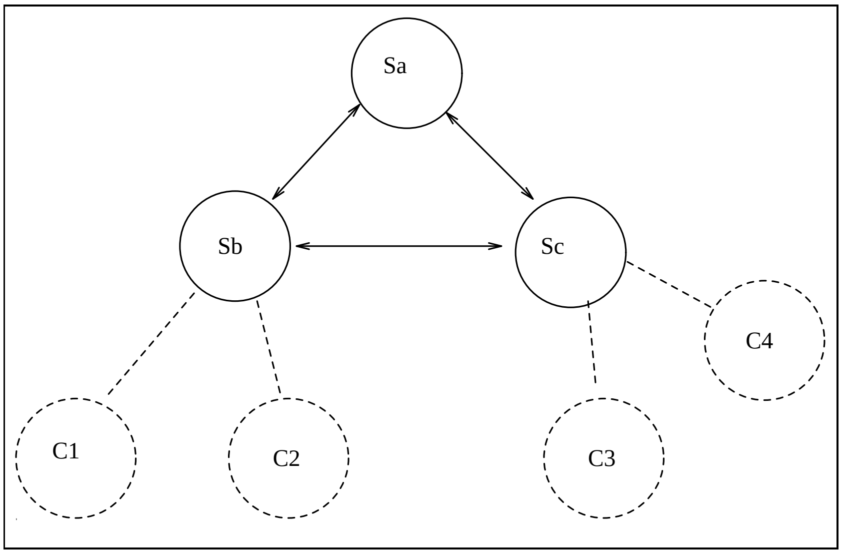

Figure 1: Core and cached copies.

3 Managing the core

Before we discuss dynamic reconfigurations of the core set and the impact of mobility, we briefly review the basic choices for managing a set of replicas. This is not intended to be a survey of existing replicated data management algorithms (there are a large number of them). Instead we try to distill the key ingredients in such algorithms.

3.1 Failure and fragmentation model

The first step in defining a replicated data management algorithm should always be to specify the undesired, expected failures [LS76], i.e., the failures that the algorithm will cope with. For the processor (and memory) the most common models are the fail-stop [SS83] and the Byzantine one [LSP82]. Here we will assume fail-stop processors: when a failure occurs the processor simply halts; the contents of main memory are lost but stable storage is unaffected. For the network, one can assume a reliable network [HS80] or a partitionable network. For our discussion, we assume a partitionable network that delivers messages between a pair of sites in order and that does not inject spurious messages.

Note that for each failure model, one typically uses low level protocols to increase the likelihood that the model holds. For example, if we assume messages are delivered in order, then the network may add sequence numbers to messages. Some proposed replicated data algorithms mix these low level failure management mechanisms with the replicated data management scheme. We believe it is much better to layer the protocols, so that the data management algorithms can assume a stronger failure model and not concern itself with how the probability of "noncompliance" with this model is made acceptably small.

Another initial decision is the selection of a correctness criteria for data management. The most commonly used criteria is that of *serializable schedules*, and in particular *one-copy serializability* [BHG87] for replicated data. Intuitively this means that the execution of the transactions should be equivalent to a serial execution of the same transactions where every access to a replicated object is replaced by an access to a single copy of the object. This will be our correctness criteria here, except for Section 6 where we explore other less strict ones.

If we are managing a collection of objects, each object could be replicated at a different set of sites. A *fragment* is a collection of objects that is replicated at the same set of sites. In terms of designing algorithms, we feel it is much better to first develop a *one-fragment* algorithm and then generalize it to multiple fragments (which is usually straightforward), rather than to develop a multi-fragment algorithm directly (e.g., as is done in [CS92]). Thus, for the time being we focus on one-fragment algorithms; multiple fragments are discussed in Section 6. In what follows, keep in mind that a fragment may be located at a single core site. This is a reasonable scenario in a mobile environment since (a) there may be additional cached copies for availability, and (b) this one core copy may migrate, as discussed later.

3.2 Concurrency control

The basic concurrency control strategy ensures that readers exclude writers, and that writers exclude other writers. In general this can be described by *read and write quorums* [BGM86]. We will explain quorums in terms of locking, although it is not necessary to use locks. (However, all implemented systems use locks and locks are simple and intuitive). The read quorum specifies the sites where read locks must be obtained when an object is read; the write quorum specifies where write locks need to be requested for a write. (The actual writes are also executed at the sites where write locks are held; however, they eventually have to be propagated to all sites. See below.) The locks can be requested *pessimistically* (before the read or write takes place) or *optimistically* (when the transaction is preparing to commit).

There are four main types of quorums that have been suggested in the literature. We illustrate them using a system with three sites, a, b, and c.

- Primary copy: Read quorum is $\{a\}$, write quorum is $\{a\}$. Site a is the primary site; both read and write locks are requested there.
- Read-one-write-all: Read quorum is $\{\{a\}, \{b\}, \{c\}\}$, write quorum is $\{a, b, c\}$. To read data, we get read locks at any single site; to write we must write lock at all sites ("all sites" is defined by the directory; see Section 3.4).
- General quorums [Gif79]: There are many possibilities; one example is a read quorum of $\{\{a\}, \{b, c\}\}$ and a write quorum of $\{\{a, b\}, \{a, c\}\}$. This quorum can be implemented by giving a 2 votes and b and c 1 vote each, and by requiring transactions to read lock at sites containing at least 2 votes, and to write lock at sites with at least 3 votes.
- Majority quorums: This is simply a special case of general quorums, that is referenced often in the literature. Here, each core copy is given one vote; a majority of votes is required to read and to write.

3.3 Propagation of updates

When the quorums allow transactions to write (and request write locks) at a subset of the core sites, we also need a mechanism for eventually propagating the updates to the remaining core copies. For example, in a primary copy scheme, the primary copy can broadcast the updates (including a sequence number) to the backup copies.

In addition, we also need a mechanism for preventing transactions from reading stale data at core sites that have missed updates. This is not a problem in the primary copy scheme (where all reads occur at the primary copy that is always up to date) nor in the read-one-write-all scheme (since all core copies are always up to date). However, a stale-date-prevention mechanism is needed for general quorums. In our three site example above, suppose that a transaction T_1 writes an object at a and b only. A subsequent transaction T_2 wishes to read the object, so it read locks at both b and c, say. The stale-data-prevention mechanism must ensure that the actual read occurs at b, or that site c is brought up to date (by propagating T_1's update) before T_2 reads at c.

3.4 Directory update mechanism

The *directory* is a structure that specifies which sites have copies of the fragment. The directory update mechanism allows us to change the core set (and possibly the quorums used). The mechanism must ensure that transactions that are using an old quorum (based on an old directory) do not execute under the current one. The directory update mechanism is the key to operation in a mobile environment and will be discussed in Section 4 in more detail.

4 Directory strategy and update mechanism

As pointed out in [BI92], the location of the copies becomes a dynamically changing data item. Thus, in a mobile environment it is important to be able to change number of core copies and/or the sites that hold them (i.e., change the core set). Let us start by discussing the reasons why we may wish to *move* a core site, say a site S_a to another S_b. (In this case, S_a is removed from the core set while S_b is added.)

1. S_a has failed and is expected to be down for a long time.
2. S_a is being brought down for preventive maintenance.
3. S_a is mobile and is running out of power.
4. S_a is mobile are is moving out of radio contact.
5. there has been a change of data access patterns, and it is determined that S_b is a more effective site for the core copy.
6. the user who "owns" copy at S_a moves to S_b, carrying a copy of the data in say a floppy disk.
7. S_a fails but the data is on a dual-ported disk, accessible S_b.

Note that except in cases (1) and (6), S_a is active and can participate in the movement. As we will see, this makes it easier to do migration.

Similarly, it may be desirable to change the number of core copies, either adding sites or removing sites to the core set. Increasing the number of copies can improve data availability (e.g., it may be more likely to find a quorum), but increases overhead (e.g., more copies need to be updated). Eliminating sites from the core set has the opposite effect. In addition, eliminating failed sites from the core set (and changing the quorums) makes it more likely to find a quorum.

Having argued that it is important to be able to change the core set (defined by the directory), let us discuss how the directory can be updated. The first step is to understand the directory structure. There are four aspects to consider.

4.1 Partial versus complete directory

A complete directory defines the location of all core copies, while a partial one only gives the location of some core copies. For example, say the core set is sites S_a, S_b and S_c. A complete directory for this fragment would list $\{S_a, S_b, S_c\}$. A partial directory would give, say, $\{S_a, S_b\}$ as well as a pointer to some other directory (perhaps on another site) that gave the rest of the core set. (For example, we can organize the directory into a tree, where each node only knows about one core site and the children in the tree.)

Some researchers have argued in favor of partial directories because no one structure needs to record all the core set, an advantage if the core set is very large. However, we have argued that the core set is typically very small, so recording all of the participants in one place is very reasonable. Furthermore, the replicated data management algorithms are complicated *enormously* it each transaction has to traverse a complex structure to figure out what sites need to be locked or updated. Thus, in this paper we assume complete directories.

Note that the directory also records the read and write quorums that are in currently in use (this may be implicit). Changing the core sites and/or the quorum rules requires a directory update (to be discussed in Section 4.4). Sometimes it may be useful to associate a timestamp or *version number* with a directory. Each time the directory is updated, its version number is incremented.

4.2 Core versus cached directory

The (complete) directory is just another data object, and it can be replicated (and should be, for availability). One or more of the directory copies can be core; the rest can be cached (see Section 2). To avoid confusion between replication of the directory and replication of the fragment it refers to, we will refer to the latter as the *base* data or objects.

Cached directory copies can be used as *hints* as to where to find the base data of interest to a transaction; note that the hints may be out of date. However, before a transaction can commit an update, it needs to read lock a read-quorum of the directory core sites to ensure that it has an up to date version of the directory. (Recall that the transaction uses the directory information to decide where to lock the base data and where to propagate updates to.) In the rest of this paper we will only refer to core directory copies.

4.3 Number and location of copies

The complete core directory can be centralized (1 copy) or may be replicated at a number of sites. The directory core set specifies this. The centralization strategy is simple but not very robust. Without ruling out centralization, we will assume the directory is replicated at several sites (centralization is just a special case of this).

The next question to address is the location of the directory core copies. One may be tempted to place them at an arbitrary set of sites, but this would mean we needed another level of directories! Hence, there are only two reasonable choices for locating the core directories:

1. Place them at a *fixed* set of sites that never changes (e.g., on three fixed servers on the network).

2. Place them at the (base) core sites. That is, if a directory at site S_a states that S_a, S_b, S_c have copies, then it means that these three sites have a copy of the base data as well as of this same directory.

Placing the directories at fixed sites makes it easier to find the directories. On the other hand, it makes it impossible for the core set to reconfigure if it is disconnected from the fixed directory sites, or if the fixed directory sites are unavailable. Thus, there is a tradeoff between ease of location of copies and availability. To keep things simple, in this paper we will assume Option 2 (directories at core sites). However, most of the algorithms and ideas we will discuss also apply to Option 1.

Recall that sites beyond the core set, perhaps all other sites, may have a cached copy of the directory. This makes it easier to locate the core set under Option 2. Any of the schemes for managing cached copies could be used here. The entries in the caches are, of course, only hints as to the location of the core set. We return to this issue in Section 7.

To summarize, we have N core sites, each with a directory of the form:

DIRECTORY:	VER
1:	S_{i_1}
2:	S_{i_2}
3:	S_{i_3}

Each entry of the form i: S_j means that the i^{th} core copy, as well as a copy of this directory, are at site S_j. The VER entry at the top is the directory timestamp or version number.

4.4 Directory management

The directory is a replicated object that has to be managed just like any other. There are several choices:

[a] design a special purpose algorithm for directory updates (e.g., [DS83]).

[b] Use same algorithm as used for the base data.

[c] Use a standard algorithm, but different than that used for base data.

Option [a] might have some performance advantages, although we doubt it. Furthermore, it represents a lot of additional design work, so we will not consider it here.

For the rest of the options, the key idea is to guarantee serializability of both base and directory update transactions. That is, directory plus base objects are treated as a unit, and whether a transaction modifies the directory part or the base part of the unit, it does not matter: the "standard" concurrency control rules must be followed.

To illustrate, consider a transaction T_3 that will update the base data. It needs to read the directory to know what sites hold base data that must be updated. Thus, T_3 must read lock the directory to prevent some other transaction to from changing it (assuming we use locking for concurrency control). Transaction T_3 must obtain read-locks at a read-quorum for the directory management scheme in use; it does this by first locking and reading one copy to figure out what other directory copies need to be locked. When the transaction commits, the directory read locks are released.

If we wish to enforce concurrency control in an optimistic way, we proceed as follows. Transaction T_3 reads the directory without locking it, noting the directory timestamp or version number. At commit time, T_3 sends the version number of the directory it read to all write-quorum sites; each of these in turn checks if the local directory is the same as seen by T_3. If not, T_3 is aborted (i.e., the site votes "no" in the commit protocol). If the versions match, then the sites votes "yes" in the commit.

In the two examples above we only read the directory before modifying the base data. To modify the directory itself and reconfigure the core set, we simply run an update transaction on the directory. This update must follow the usual rules. This is, if locking is used, the transaction must write lock the directory (at a write-quorum of sites). If an optimistic scheme is used, again, the directory read initially is validated when the update commits.

Notice that it is not a good idea to use a read-one-write-all concurrency control scheme for the directory (where the write quorum is all sites). If this were done, the directory could not be updated when any one site was unavailable. However, this is exactly the time when one may want to change the core set. The other quorum based concurrency control schemes would not have this problem. With primary-copy scheme, reconfigurations would be possible unless the primary directory site were unavailable.

4.5 Primary by row directory management

As we have seen, the directory concurrency control scheme determines when core set reconfigurations can be performed. Thus, it is important to select a scheme that matches the system requirements.

To illustrate this point, let us consider a scenario where copies migrate often. For example, say a person is editing a document, first from his office computer, then from a laptop on the train ride home, then from his home computer, and then back to the office. At each step, a copy of the document is copied from one machine to the next. Since we want the person to be able to update the document on each machine, each new copy should become a member of the core set and each old copy should be deleted. (There may be other core copies in use by other people; see Section 6 for a further discussion of this.) We would like to be able to migrate the core copy in this fashion, regardless of whether other core copies are reachable at the moment. This suggests that each core copy should have the ability to modify the portion of the directory that concerns it.

We call this scheme a Primary By Row directory update algorithm. This is not really a "new" concurrency control scheme; it is simply an adaptation of the primary copy scheme to suit the needs of our copy migration scenario. To explain this scheme in more detail and to illustrate its advantages, assume we have S_a, S_b, S_c in the core group. Also suppose that the base data is managed under a majority scheme (two sites are needed in read and write quorums). (But keep in mind that primary-by-row directory management can be used with any base data scheme.)

Assume that currently the 3^{rd} copy is at site S_a. (See Figure 2.) This site controls line "$3 : S_a$" of the directory and can initiate changes to it on its own (local commit). For instance, say S_a wants to migrate its copy to S_d. (Figure 3.) First, S_a ensures that S_d has a copy of the data and of all transactions that S_a knows about. Then S_a changes its entry to "$3 : S_d$",

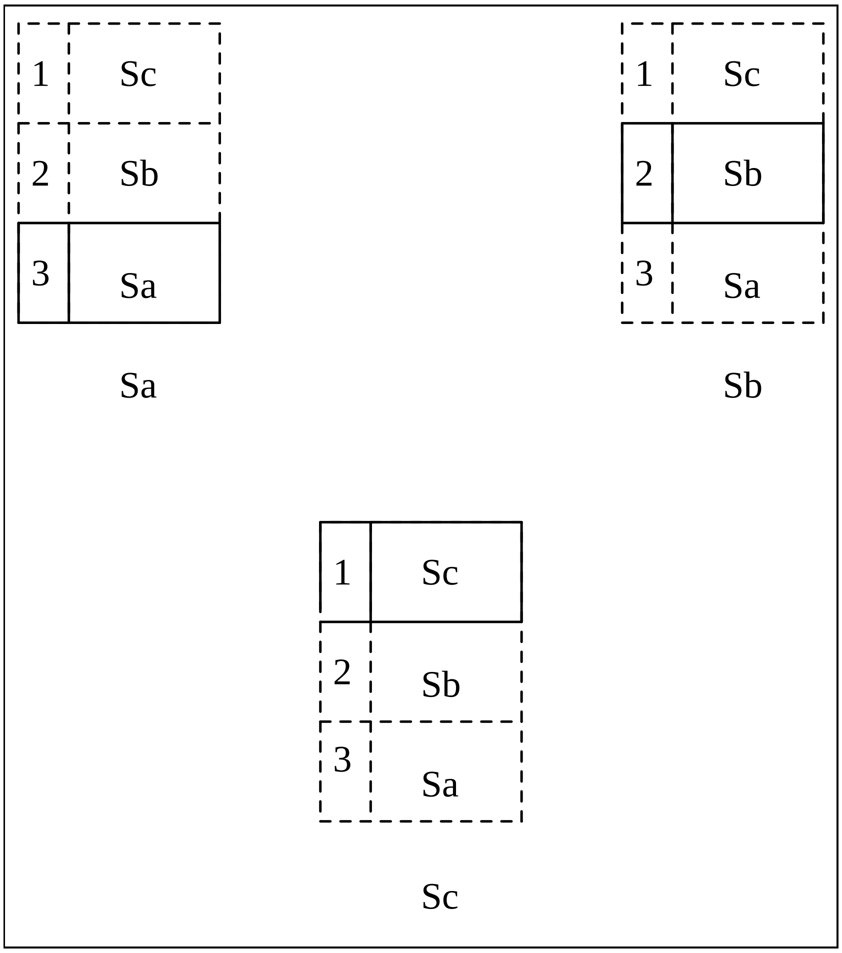

Figure 2: Primary by row directory management. (Solid lines indicate that control is at that site.)

commits the change, and propagates the new line to S_b, S_c, S_d. When these sites receive the update, they install it.

In this case, we do not need a version number for the entire directory. Instead we have a version vector, one entry per site. For instance, assume that the original directory had versions $< 1, 1, 1 >$, where the i^{th} entry in the vector is the version number for line i of the directory. The change illustrated in the previous paragraph would then yield version vector $< 1, 1, 2 >$, meaning that the 3^{rd} copy now has version number 2. If the 2^{nd} line had concurrently changed, the new vector would be $< 1, 2, 2 >$.

If a base transaction T_2 is running concurrently with the directory change, it may initially read directory $< 1, 1, 1 >$ and then at commit time be informed (by the participants of the commit) of a different vector. In this case, we can abort T_2. Or if we want to be more efficient, we can try to discover the new directory and get the missing acknowledgments. This type of optimization will be discussed in Section 5.

Since only one site at a time controls the i^{th} row in the directory, the update protocol is safe. Thus, moving the copy at S_a to S_d is like passing a token. However, the token can be lost. For instance, say S_a commits its change and fails before it propagates the new row 3 entry to other sites. Then everyone will continue to think that S_a is the row 3 site, and will try

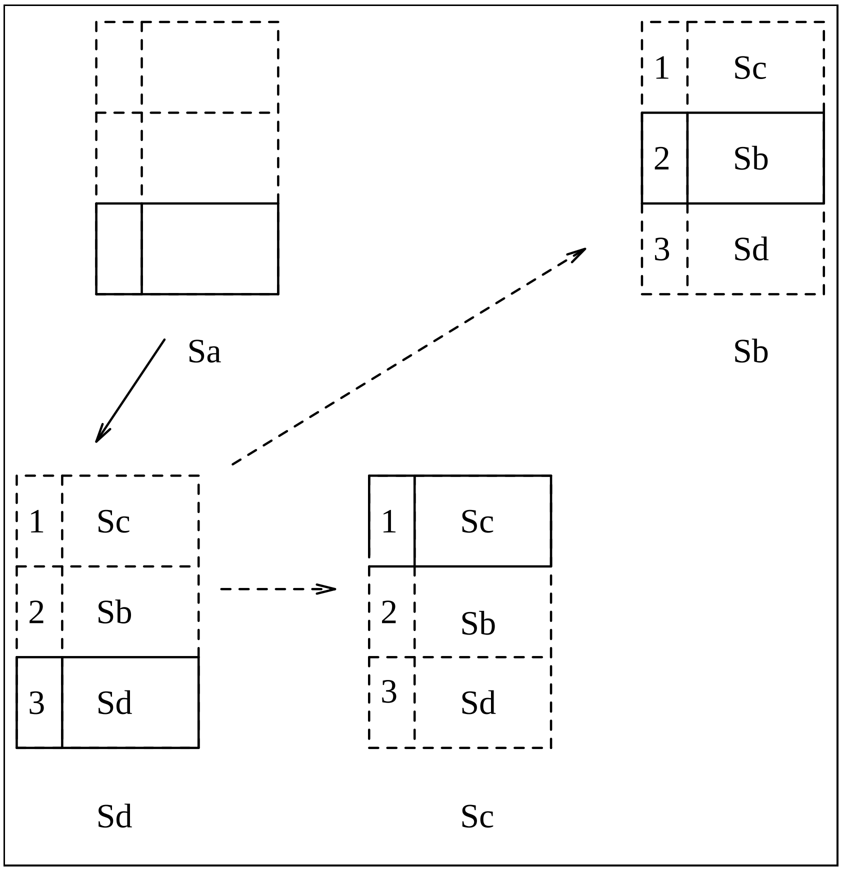

Figure 3: Migrating a copy.

to get locks form it. This will be just as if site S_a had failed. Fortunately, the base algorithm can tolerate this failure, and transactions can continue to commit if they have locks form the first and second copies.

There are many variations on this theme. For example, if votes are used to implement read and write quorums, site S_a may "split" its token and select two or more new sites to represent it. In this case, each could have half the votes that S_a had originally. Also, two sites may combine into one, with a protocol that only involves them. The new site would have the sum of their votes. There are also a variety of ways to create brand new directory entries (or to delete them). We do not discuss these here.

The main advantage of the primary-by-row directory management approach is that decisions to change the directory are localized. As we have stated, it seems especially attractive for scenarios where copies indeed migrate, e.g., because they are on a removable floppy disk or on a magnetic strip on a card. The fact that the copy has been removed from one site and installed in another represents the migration of the token, and should be allowed to happen regardless of whether the rest of the copies "authorize" the change or are aware of it.

5 Update propagation and efficiency issues

In this section we discuss some details regarding replicated data management in a dynamic environment. It is important to understand these issues as they can impact correctness and data availability in a mobile environment.

5.1 Update propagation

As discussed in Section 3.5, a replicated data management scheme must ensure that updates are eventually propagated to all core sites. For instance, assume we use a "majority scheme"[Gif79] with three core sites: i.e., any two sites constitute a read quorum, and any two sites constitute a write quorum. Say transaction T_3 commits an update at sites S_a, S_b only. Moments later, T_4, which depends on T_3, executes at S_b and tries to commit at S_b, S_c. As part of this commit, T_4 must bring S_c up to date and ensure it is made aware of T_3 (else the copy at S_c would not be valid.) For other schemes the details are different, but key principle is not: before a site can commit a transaction T_4, it must have seen all transactions that preceded T_4 in the serialization order. (Actually this may be too strong: it is only necessary to see transactions that T_4 depends on. But keeping track of what T_4 depends on is too hard, so one just ensures all previous transactions are reflected.)

Directory update transactions are no different. Before they commit, they too must bring sites up to date if they have missed transactions. Thus, if a directory update transaction adds a site, this site must have an up to date copy of the base data (and of the directory of course), before the directory update can be committed.

To illustrate, say we use a majority scheme to manage the directory. Initially, sites S_a, S_b, S_c each have a copy of directory:

DIRECTORY:	$VER = 5$
1:	S_a
2:	S_b
3:	S_c

Say site S_a fails, and site S_b wants to add another site S_d to the core set. Site S_b runs a directory transaction T_1 to update the directory. Site S_b sends messages to itself and S_c, asking for the update; the message includes the directory version number, say 5. If site S_c still has version 5, it locks the directory and goes into a prepared state for T_1. Site S_b does the same. If S_b receives 2 positive messages it has a write-quorum so it commits T_1 and sends out commit messages. Upon receipt of the commit message, S_c releases the new version of the directory (VER: 6; 1: S_a; 2: S_b; 3: S_c; 4: S_d).

Site S_d has to be brought up to date "at the same time" the directory is updated. To illustrate, say a transaction T_2 (on base data) committed before T_1 ran, and obtained votes form S_a and S_c. This means that only S_a and S_c have the updates generated by T_2. Assume now that T_1 changes the core set to S_b, S_c, S_d. In this case, knowledge of T_2 will only be at a minority of sites and this is dangerous. For example, a transaction that conflicts with T_2 could commit at S_b, S_d without any knowledge of T_2! To avoid this, T_2 must be propagated to a majority of new group members *before* the new group is open for business. This is done as part of the commit protocol that updates the directory. Thus, when the sites prepare (as described above) they acknowledge with a "list" of committed transactions they have seen. S_b, the coordinator, compares the responses and must ensure that each transaction must have been seen by a majority of new group sites. If this is not the case, it sends the transactions (like T_2) to sites that missed them (like S_b, S_d in our examples). When sites acknowledge the receipt of the missing transactions, then S_b can finally commit the directory change T_1.

To complete the example, assume another transaction T_3 (to base data) is executing concurrently with the directory update T_1. When T_3 starts it reads directory version 5 and thus tries to get at least 2 out of three votes from sites S_a, S_b, S_c. If T_3 tries to commit after directory version 6 in palace, it will be aborted (one of the two sites participating in the commit of T_3 will notice the out of date directory and will vote to abort). If T_3 commits before T_1, its update (like that of T_2) will not be lost. The key point is that updates to the directory are serialized with regular base transactions.

5.2 Lazy versus immediate propagation

Update transactions (to base data) do not necessarily have to lock and propagate their changes to all core copies. In the example in the last subsection, transaction T_2 only updated S_a and S_c. Under normal circumstances, what should the site running T_2 do about the updates to S_b? If it is *lazy* it will just not send the update out, figuring out that eventually S_b will miss it and ask for it. If it is *conscientious*, it will make a best effort to send the update as soon as possible.

In practice it makes sense to always be conscientious. There is no real advantage is postponing the transmission of T_2's updates to all sites that are active. Once the site executing T_2 knows all core sites got the update, it can delete its log records. Furthermore, updates sent conscientiously are likely to be sent while data is cached in memory; updates requested later will likely generate disk traffic. So the bottom line is that transactions should propagate updates to all available sites, regardless of what the write-quorum is.

5.3 Efficiency issues

The protocols we have presented here can be optimized in a variety of ways. Here we present two optimizations to illustrate.

Let us return to our example above where T_1 updates the directory. Many uncommitted base transactions (like T_3) may have started before T_1 and read the old directory. According to our example, they all should abort when they try to commit! However, this is not necessary. Say T_i is a transaction that straddles a directory change. When T_i starts, it reads a directory copy with version number n. (It could even be a cached directory copy.) Based on the n directory, T_i requests read and write locks. During T_i's first phase of commit, each participant can return its current directory version number. If T_i receives a version number newer than n, instead of immediately aborting, it can try to read the new directory, and simply collect additional locks to satisfy the new read and write quorums. If lucky, maybe T_i will already have enough votes under new directory too and it will be able to commit immediately.

For instance, in our S_a, S_b, S_c example above, say T_i reads directory version 5 (indicating the core set is S_a, S_b and S_c). Transaction T_i requests write locks from S_b and S_c (a write quorum). During the commit process, T_i receives acknowledgments from S_b and S_c but with version 6. After T_i reads the version 6 directory (since it is small, maybe it was included in an acknowledgment message), it discovers that the core set is S_a, S_b, S_c and S_d, and 3 out of the 4 sites are needed for a write quorum. Thus, it needs to set locks and receive an acknowledgment from an additional site. If this can be done, then T_i can successfully commit. On the other hand, say that in the version 6 directory S_a was being replaced by S_d. In this case, the new core set is S_b, S_c, S_d and T_i already has acknowledgments from 2 of these 3 sites. Hence, T_i can commit under the new directory anyway, with no further lock requests.

Another possible performance problem is that of blocking by a directory update. To illustrate, consider the following scenario. The core set consists of S_a, S_b, S_c, and site S_b is trying to add S_d to the set. Site S_b sends directory update request to all sites, and they all acknowledge. Unfortunately, site S_b fails at this time, leaving the other sites blocked in the prepared state, not knowing whether directory version 5 (set S_a, S_b, S_c) or version 6 (S_a, S_b, S_c, S_d) is valid. If we enforce concurrency rules strictly, the directory data is dirty and no other transactions can read it, so no base updates can occur in this state. However, we may get around this in some cases, when transactions can get quorums under *both* possible versions. For example, say T_2 starts reading version 5, and when it wants to commit, it tries to get acknowledgments from S_a, S_b, S_c. Say S_a does not reply and S_b replies that at this point it does not know whether version 5 (S_a, S_b, S_c) or version 6 (S_a, S_b, S_c, S_d) will be the valid one. Site S_c replies similarly. At this point, T_2 notices that it got enough acknowledgments under version 5, from S_b and S_c; thus if it turns out that version 5 is indeed valid (say T_1 aborts), then T_2 will be safe. However, if version 6 turns out to be the good one, then T_2 will not be safe. To avoid this, T_2 tries to get an acknowledgment from S_d. Say site S_d does set the required locks and acknowledges. At this point, T_2 has 3 out of 4 acknowledgments under version 6, so one way or another T_2 will be safe and can be committed, even if the directory update transaction T_1 is blocked.

6 Multiple Fragments and Non-Serializability

In Sections 3 and 4 we made two important assumptions: we only considered a single fragment, and we required that the execution schedule on the core copies was serializable. Extending the algorithms of Section 4 to multiple fragments, while still guaranteeing serializability, is relatively straightforward. The key is simply to make sure that at transaction commit time appropriate locks are held for all fragments involved in the transaction (we continue to assume a locking strategy for simplicity).

To illustrate, consider a transaction T that accesses fragments F_1 and F_2. Assume that T reads data from F_1 and updates F_2. At T's commit time, it must hold a read lock on the F_2 directory and on a write quorum of the F_2 sites. In addition, it must hold a read lock on the F_1 directory and an F_1 read quorum. This ensures that the resulting schedule is serializable.

Forcing transaction to acquire locks from multiple fragments may hurt performance. In our example, while T is committing, all other conflicting transactions on F_1 and F_2 will be blocked. If the commit process blocks (see Section 5.3), then *both* fragments will be blocked indefinitely. Also, in a mobile environment, it may take T a long time to concurrently get both sets of locks. It would be much easier, for example, to get the F_1 set of locks first, read F_1, release the locks, then get the F_2 locks and update F_2.

If T gets the sets of locks in this "independent" fashion, then performance improves but serializability is lost. Actually, not everything is lost. If transactions follow the locking rules for each fragment, then the execution schedule as far as that fragment is concerned is serializable. (More formally, if one projects out of the global schedule all actions except for those involving a single fragment, the resulting sub-schedule is serializable [GMK88, KS88].) This means that all consistency constraints involving data within each fragment (intra-fragment constraints) will be satisfied. However, global serializability does not hold and inter-fragment can be violated. This type of weaker correctness guarantee has been called either fragmentwise serializability or local serializability [GMK88, KS88].

In some applications there may be no inter-fragment constraints, so local serializability is fine. In other cases, it may be acceptable to violate inter-fragment constraints, as long as application programs try to fix them when they notice they have been violated. For example, say one fragment represents new sales made by a set of traveling salespeople on their portable computers. A second fragment could represent the actual inventory on hand. If there is an inter-fragment constraint such as "one can never sell something unless it is already sitting in the warehouse" then one probably requires global serializability. One the other hand, perhaps the application is structured in such a way that can sell items not in inventory, on the assumption that more will be manufactured if necessary. In this case, the constraint is not "hard": salespeople will check if the desired product is in inventory, and will then advise the customer. But it would be acceptable if once in a while the customer is told that something is available, only to discover after the sale is recorded that it is out-of-stock and there will be a delay in receiving the product. The fact that the constraint is not hard lets transactions first check inventory (perhaps even on a cached copy), release locks, and then record the sale. This is indeed many applications work today, even when computers are not mobile.

Local serializability still requires that transactions request read and write locks for the core fragment and its directory. If one relaxes the requirement for serializability of the fragment schedule, one can arrive at more efficient protocols. In particular, a number of algorithms have been suggested for keeping copies of a fragment approximately equal, e.g., [BGM, KB92]. In these algorithms, all core copies can be updated, but the difference between any two copies is bounded, assuming copies have a numeric value.

As a simple example, say three traveling salespeople are selling widgets. Each has a counter reflecting how many widgets have been sold, call them X_1, X_2, and X_3. Say each salesperson is allowed to sell a maximum of 3 widgets without informing the others. That is, each person can increment X_i by at most three without getting an acknowledgement that the other people have reflected those increments on their counters. (To report an increment, a site does not send the value of its counter; instead it send an "increment counter" action.) Then we could show that the counters will differ by at most 3 units (assuming no decrements) and that X_i will differ from the true total number of widgets sold by at most 6. This scheme gives the salespeople the flexibility to sell at least some widgets even when they are disconnected from the others, and to have an approximate idea of how many total widgets have been sold. The disadvantage is that applications must now understand and be able to cope with the approximate data. For instance, a program to compute sales bonuses needs to understand that the value it reads in X_i may be off by some small amount.

7 Discussion

We have argued that there are three main types of replicated data: cached, core and directory. Cached copies are read-only and can thus be managed in a more flexible way. Core copies are updateable and are expected to be few in number. The directory indicates where core copies are stored. We have argued that reconfigurations of the core copies can be treated simply as transactions that modify the directory. This implies that the same set of choices available for managing the core copies can be used to reconfigure the core, i.e., can be used to update the directory.

By allowing the directory to be updateable one achieves replicated data management algorithms that are dynamic, i.e., that can adapt to the disconnection or failure of a core copy. These dynamic algorithms attempt to continue operation (new updates to the core) even when some members of the core set are unavailable. Whether a copy copy is unavailable because it *moved* away or it simply *died* is not really critical in these algorithms. Hence our claim that mobility does not introduce any fundamental new problems or algorithms for replicated data management.

Mobility may make, however, certain choices within the available "menu" more or less desirable. In particular, there are three aspects of replicated data management that may be impacted:

- If disconnections are going to be frequent (due to travel of the copies), then one should select a directory management strategy that allows frequent reconfigurations. For example, in Section 4.5 we described a Primary By Row strategy that we think is especially well suited to frequent migration of a copy. The assumption here was that a site could orchestrate the migration of its copy before it became disconnected, as opposed to a failure case where the original site is not available for the move. If failures were the primary cause of reconfigurations, then perhaps a quorum directory strategy would be best, since a majority of the survivors can reconfigure the directory. As we have stated, this tuning of the directory strategy does not really constitute development of a "new algorithm." Even the Primary By Row strategy we advocate here is simply an application of the Primary Copy algorithm to managing each row of the directory.

- A core copy should not be placed on a low-bandwidth, limited power mobile site, unless updates originate mainly at that site or it is imperative that the mobile site be able to generate base updates when disconnected. A core copy must receive every update originating at other core copies, and must transmit all of its updates. If the communication link to a mobile workstation is low bandwidth, it will be difficult to support all this update traffic, and hence one should avoid placing a core copy there. Similarly, if the workstation has limited power, it will be turned off often, interfering with updates to the other core copies. Furthermore, storage reliability on mobile units tends to be significantly lower than on fixed servers (e.g., laptops can be dropped or stolen), again leading to interference to other copies. Thus, an arrangement where the core copies are on reliable servers, and the mobile workstations have cached copies seems much more attractive.

There are two exceptions we see to this rule. One is if the majority of the updates to the fragment originate at the mobile unit. For instance, consider a meter reader working for a utility. Clearly, he or she wishes to operate in a disconnected fashion, and most of the transactions will simply input meter readings. No other sites will be entering data into this person's readings database, so it clearly makes sense for the machine being carried by the reader to be the primary core copy for this fragment.

It is important to note, however, that while this arrangement is good for disconnected operation by the meter reader, it is vulnerable to total data loss in case of a disaster (e.g., the portable machine falls in a lake). Thus, it is important to combine the primary-copy strategy with frequent backups to a cached copy. If the mobile unit is truly disconnected, then the cached copy will have to be on removable floppy disks. Even with these local backups, transactions that commit between the last backup and the disaster will be lost. (This can be avoided by not committing a transaction until its updates have been applied at the backups, but then this would not be a primary copy strategy; it would be a read-one, write-all (backups) strategy. See Section 3.2.)

The second exception is when it is important for a disconnected station to perform updates on the base data. One example of this scenario is the traveling salespeople of Section 6. Another example may be a group of people working on a joint document while traveling in areas not served by good networks. In both of these cases, access to the data is required during partitioned operation, and there is no alternative but to place the core copies on the mobile workstations. The price to pay is either (a) poor performance (e.g., my updates may be blocked while my partners are disconnected), or (b) non-serializability. In the latter case, the application or users will have to cope with inconsistencies, e.g., approximate counters in the case of the traveling salespeople, or conflicting updates to the same portions of the document in the case of the traveling paper writers.

- A good database and application design can lead to improved data availability and performance in a mobile environment. The key is to fragment the database in such a way that each fragment can be controlled by the sites that need to control it, and in such a way that there are few inter-fragment constraints.

  To illustrate, let us return to the sales and inventory application. Say there are two relations: each tuple in the $INVENTORY$ relation records the item number, its description, and the quantity on hand in the warehouse. Each tuple in the $SALES$ relation records a particular sale, giving the date, the salesperson id, the customer, id, the item number, and the amount sold. If one considers both relations a single fragment with an inter-fragment constraint such as "quantity on hand for an item should be less than the total amount sold," one gets terrible performance. If a salesperson portable computer has a core copy of the fragment, then there will be a huge communication overhead. If the core copies are stored on servers, the salesperson will not be able to record a sale unless he is connected to the server.

As discussed in Section 6 we can improve things by splitting the database into two fragments and eliminating the inter-fragment constraint. We can do even better by partitioning the $SALES$ relation into a set of fragments S_1, S_2, ... where S_i has the sales record for salesperson i only. In this way, the machine for salesperson i can be the primary core copy for S_i. Accordingly, each salesperson can enter sales records independently of the others.

The $INVENTORY$ relation can be stored at one or more servers. Each salesperson's machine can have a cached copy of $INVENTORY$. Similarly, the servers can have cached copies of the S_i fragments. At the server, a background process can check the new sales records and update the inventory accordingly. The updated inventory is then sent to the salespeople as part of the cache management algorithm. Thus, we see that by fragmenting the data properly, each user gets to modify his critical data at all times. For an additional discussion of how to fragment a database see [GMK88].

In closing this paper, we note that mobility may impact other aspects beyond the management of the core copies themselves. Specifically, the management of cached copies needs to be tuned according the the bandwidth available and the currency requirements of the application. For instance, if the cached copies can be reached via a broadcast network, it makes sense to update all cached copies of a fragment at once. If the available bandwidth is low, then updates to the cached copies should be as infrequent as possible, and the updates should be compressed as much as possible (e.g., giving only fields that changed, as opposed to full record images).

A second aspect concerns finding the directory for a fragment. As discussed in Section 4.3, cached copies of a directory can be placed at a variety of sites, to serve as hints to the true location of the core copies. The same issues mentioned above apply here: One needs to find appropriate locations for the cached directories, and appropriate update frequencies to minimize the cost of finding the core. At one extreme, every time the directory changes, a message could be broadcast to all sites. At the other extreme, a cached directory could be changed only when it is discovered to be out of date. At that point, a query is broadcast to all sites to find the core. (Strategies to invalidate caches in mobile environments are studied in [BI93].) Finding the best strategy would depend on factors such as the available bandwidths, the availability of broadcast channels, the frequency of directory changes, and the desired response time for finding the core.

Acknowledgements. Ramon Caceres and Sharad Mehotra provided useful comments on an eralier version of this manuscript.

References

[AGM87] R. Abbott and H. Garcia-Molina. Reliable Distributed Database Management. *IEEE Proceedings*, 75(5):601–620, May 1987.

[BGM] D. Barbará and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal.* To appear.

[BGM82] D. Barbará and H. Garcia-Molina. How Expensive is Data Replication: An Example. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 263–268, February 1982.

[BGM86] D. Barbará and H. Garcia-Molina. Mutual Exclusion in Partitioned Distributed Systems. *Journal of Distributed Computing*, 1(2), June 1986.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[BI92] B.P. Badrinath and T. Imieliński. Replication and Mobility. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, November 1992.

[BI93] D. Barbará and T. Imieliński. Sleepers and Workaholics: Caching Strategies in Mobile Environments. Technical Report MITL-TR-58-93, MITL, June 1993.

[CGP81] E.G. Coffman, E. Gelenbe, and B. Plateau. Optimization of the number of coopies in a distributed system. *IEEE Transactions on Software Engineering*, 7(1):78–84, January 1981.

[CP92] S. F. Chen and C. Pu. An Analysis of Replica Control. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, November 1992.

[CS92] D.D. Chamberlain and F.B Schmuck. Dynamic Data Distribution (D3) in a Shared-Nothing Multiprocessor Data Store. In *Proceedings of the Eighteen International Conference on Very Large Databases, Vancouver*, August 1992.

[Dav89] S.B. Davidson. Replicated Data and Partition Failures. In S. Mullender, editor, *Distributed Systems*, pages 265–292. Addison-Wesley, 1989.

[DB85] D. Davcev and W. Burkhard. Consistency and Recovery Control for Replicated Files. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 87–96, December 1985.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[DS83] D. Daniels and A.Z. Spector. An Algorithm for Replicated Directories. In *Proceedings Second ACM Symposium on Principles of Distributed Computing, Montreal, Canada*, pages 104–113, August 1983.

[ET86] A. ElAbbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *Proceedings of ACM-SIGMOD 1986 International Conference on Management of Data, Washington*, pages 240–251, 1986.

[Gif79] D.K Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, December 1979.

[GMK88] H. Garcia-Molina and B. Kogan. Achieving High Availability in Distributed Databases. *IEEE Transactions on Software Engineering*, 14(7):886–896, July 1988.

[Her87] M. Herlihy. Dynamic Quorum Adjustment for Partitioned Data. *ACM Transactions on Database Systems*, 12:170–194, June 1987.

[HS80] M. Hammer and D. Shipman. Reliability Mechanisms for SDD-1: A System for Distributed Databases. *ACM Transactions on Database Systems*, 5:431–466, December 1980.

[IB92] T. Imielinski and B.R. Badrinath. Querying in Highly Mobile and Distributed Environments. In *Proceedings of the Eighteen International Conference on Very Large Databases, Vancouver*, August 1992.

[JM87a] S. Jajodia and D. Mutchler. Dynamic Voting. In *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco*, pages 227–238, 1987.

[JM87b] S. Jajodia and D. Mutchler. Enhancements to the Voting Algorithm. In *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton*, September 1987.

[JM88] S. Jajodia and D. Mutchler. Integrating Static and Dynamic Voting Protocols to Enhance File Availability. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988.

[KB92] N. Krishnakumar and A.J. Bernstein. High Throughput Escrow Algorithms for Replicated Databases. In *Proceedings of the Eighteen International Conference on Very Large Databases, Vancouver*, pages 175–186, August 1992.

[KS88] H.F. Korth and G.D. Speegle. Formal Model of Correctness Without Serializability. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 379–386, June 1988.

[LS76] B. Lampson and H. Sturgis. Crash Recovery in a Distributed System. Technical report, Computer Science Laboratory, Xerox, Palo Alto Research Center, 1976.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[P̂84] J.F. Pâris. Efficient Dynamic Voting Algorithms. In *Proceedings of the Fourth International Conference on Data Engineering*, February 1984.

[P̂86] J.F. Pâris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 606–612, May 1986.

[SS83] R.D. Schlichting and F.B Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[WJ92] O. Wolfson and S. Jajodia. Distributed Algorithm for Dynamic Replication of Data. In *Proceedings of the ACM-Principles of Database Systems Symposium*, 1992.