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

Daniel Barbará-Millá[a] and Hector Garcia-Molina[b]

[a]M.I.T.L.
2 Research Way,
Princeton, N.J. 08540. USA

[b]Stanford University
Department of Computer Science
Stanford, CA 94305. USA

The mobile wireless computing environment of the future will contain large numbers of low powered palmtop machines. 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.

1. Introduction

The mobile wireless computing environment of the future [13] 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 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 a survey see [1]), 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 [9].) 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 [2], 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., [6]), 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., [10, 14, 17]. 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 Section 4.

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 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. [7, 19]), 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 [3].) 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

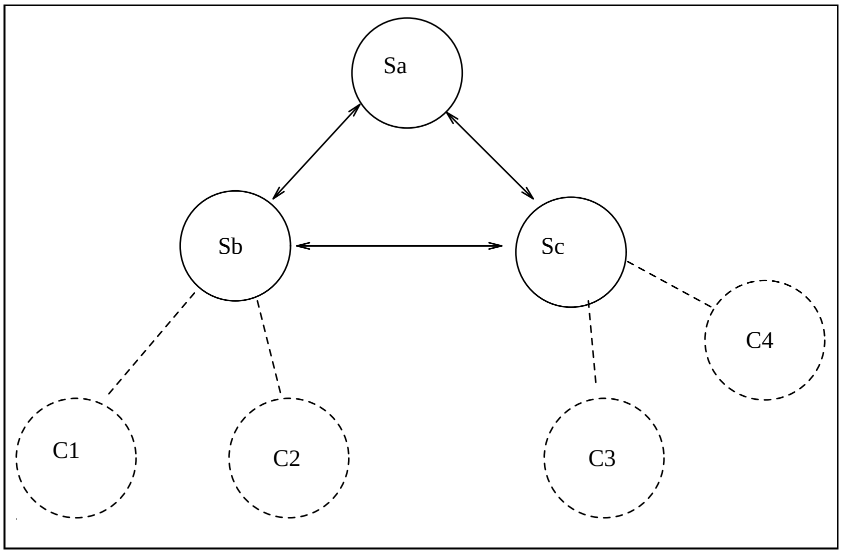

Figure 1. Core and cached copies.

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.)

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.

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 [16], i.e., the failures that the algorithm will cope with. For the processor (and memory) the most common models are the fail-stop [18] and the Byzantine one [15]. 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 [12] 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* [5] 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.

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 [6]). Thus, in this paper we focus on one-fragment algorithms.

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* [4]. 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.

- General quorums [11]: 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.

4. Directory strategy and update mechanism

As pointed out in [2], 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). There are many reasons why we may wish to *move* a core site. (For instance a site might be mobile and running out of power, or moving out of radio contact.) Sometimes the site 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 three 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. 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.

4.2. 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.

4.3. 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., [8]).

[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.

In the example 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.

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. 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.

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. Discussion

We have argued that reconfigurations of the core copies can be treated simply as transactions that modify 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 may be a group of people working on a joint document while traveling in areas not served by good networks. In this case, 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.

REFERENCES

1. R. Abbott and H. Garcia-Molina. Reliable Distributed Database Management. *IEEE Proceedings*, 75(5):601–620, May 1987.
2. B.P. Badrinath and T. Imieliński. Replication and Mobility. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, November 1992.
3. 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.

4. D. Barbará and H. Garcia-Molina. Mutual Exclusion in Partitioned Distributed Systems. *Journal of Distributed Computing*, 1(2), June 1986.
5. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.
6. 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.
7. 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.
8. 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.
9. S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.
10. 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.
11. D.K Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, December 1979.
12. 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.
13. 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.
14. 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.
15. L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
16. B. Lampson and H. Sturgis. Crash Recovery in a Distributed System. Technical report, Computer Science Laboratory, Xerox, Palo Alto Research Center, 1976.
17. J.F. Pâris. Efficient Dynamic Voting Algorithms. In *Proceedings of the Fourth International Conference on Data Engineering*, February 1984.
18. 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.
19. O. Wolfson and S. Jajodia. Distributed Algorithm for Dynamic Replication of Data. In *Proceedings of the ACM-Principles of Database Systems Symposium*, 1992.