Resumo

Nowadays, people increasingly use multiple devices to manage and share information anywhere anytime. Due to this fact, it is expected that users store different versions of same files in several devices raising a consistency problem. Most current solutions ensure data consistency across devices through central servers or Internet services. Since portable devices have intermittent network connection or no connection at all to access these services, it is essential to take advantage of proximity between devices to synchronize data between them.

This dissertation describes MultiRep, a single-user file synchronizer middleware based on optimistic approaches that ensures eventual consistency among multiple devices. MultiRep allows users to synchronize files and folders between any device reporting all relevant information about conflicts that can occur during this process. In order to achieve this, it uses version vectors for tracking changes to files and folders on several devices. Moreover, MultiRep allows users to see at any device anytime which files and folders are stored by other devices even if they are turned off or inaccessible.
MultiRep - Asynchronous Multi-Device Consistency

João Ribeiro  
INESC-ID/IST  
joao.f.ribeiro@ist.utl.pt
João Barreto  
INESC-ID/IST  
joao.barreto@ist.utl.pt
Paulo Ferreira  
INESC-ID/IST  
paulo.ferreira@inesc-id.pt

Abstract

Nowadays, people increasingly use multiple devices to manage and share information anywhere anytime. Due to this fact, it is expected that users store different versions of the same files in several devices raising a consistency problem. Most current solutions ensure data consistency across devices through central servers or Internet services. Since portable devices have intermittent network connection or no connection at all to access these services, it is essential to take advantage of proximity between devices to synchronize data between them.

This dissertation describes MultiRep, a single-user file synchronizer based on optimistic approaches that ensures eventual consistency among multiple devices. MultiRep allows users to synchronize files and folders between any device reporting all relevant information about conflicts that can occur during this process. In order to achieve this, it uses version vectors for tracking changes to files and folders on several devices.

Keywords: MultiRep, multi-device, asynchronous eventual consistency, optimistic replication, version vector, file synchronizer, distributed file system, data-sharing middleware.

1 Introduction

In the last few years, people own more and more multiple devices to store, manage and share information. At the same time, mobile devices (e.g. PDAs, smartphones, laptops) are becoming more computationally powerful with greater memory capacity. Due to this fact, people increasingly use them for entertaining purposes and to perform work that in the past was only possible with a desktop PC.

The increasing number of such devices storing a large set of different types of data, have created the requirement for having the same data in two or more devices. For instance, when working abroad, a user might find himself copying files from his desktop to his laptop in order to continue his work. Additionally, he may be interested in taking some of these files to his smartphone to perform some modifications. Also, during the trip, he may want to upload some photos taken by his digital camera to the laptop. When he returns back, all files modified abroad must be transferred to the desktop.

When the user modifies the same files concurrently, different versions of these files are created across devices raising a consistency problem. Many systems ensure data consistency among multiple devices by storing all files on a central server or online storage web service. These approaches have some limitations. For instance,
some devices (i.e. digital camera) may not have network access to the central server or Internet service to synchronize files between other devices. Additionally, when traveling long distances, often communication with a local device is easier, more efficient, and more cost effective than synchronization over long-distance links. Also, some problems of scalability and availability arise when we are using systems based on a central server. For instance, if the number of user’s devices increases too much, the central server becomes a bottleneck. Moreover, when it becomes unavailable, devices can not synchronize with each other.

The solution presented in this dissertation take advantage of proximity of devices to synchronize data among them, ensuring data consistency in a decentralized fashion. Figure 1 illustrates typical pairwise interactions that can occur at different times for synchronizing data between devices in MultiRep. The figure shows several synchronization pairs that can be created at different times between devices closest to the user. For instance, when a user walking with his tablet PC gets closer to his desktop PC at a given time, the MultiRep system creates a synchronization pair between those devices for reconciling user’s data. MultiRep will ensure data consistency in an ad-hoc way between these devices as long they can communicate with each other over time.

Figure 1: A typical set of devices and communication pathways

1.1 Objectives

The main goal of this work is to design, build and evaluate a single-user file-sharing middleware system called MultiRep that ensures the consistency of files replicated among multiple devices, relying on a single device. In order to achieve this, it uses an optimistic replication[41] scheme based on eventual consistency[52]. Additionally, MultiRep must allow users to see at any device anytime which files and folders are stored by other devices even if they are turned off or inaccessible.

In order to achieve these goals, MultiRep must have the following requirements:

- Help users maintain files replicated and consistent between several devices. Users should be able to see at any device anytime which files and folders are stored by other devices even if they are turned off or inaccessible;
- Allow users or applications to read and write replicated files on any device anytime;
- Allow asynchronous pairwise synchronization between different devices;
- Manage and merge different file versions;
• Identify and resolve conflicts (automatically when possible);
• Be independent of the operating system and user's applications (no modification to the user's application or OS's kernel should be required);
• Be transparent to the user when no conflict arises (in this case MultiRep should not be seen by the user);
• Apply efficiently synchronization mechanisms, in terms of time, memory and bandwidth usage.

1.2 Main Challenges

To fulfill the above mentioned requirements, the creation of a system like MultiRep must address some common challenges shared by other optimistic replication systems based on a pairwise synchronization model used in mobile environments. MultiRep's main challenges are the following:

• Monitoring user's behavior - MultiRep must monitor all actions done by the user when creating, renaming, deleting or modifying contents of a certain file. This is important because the system must know all relevant information of files and folders to make decisions and provide information to the user during the synchronization process. Some of the collected information regarding a given file is stored as metadata and associated with that file. Some operation-based systems, such as Bayou[9], maintain this information in a log file. Monitoring process must be efficient in time and memory;
• Synchronizing constrained mobile devices - some constrains related with mobile devices must be taken into account: battery life, low network bandwidth, intermittent connections and limited memory storage. In order to work properly in mobile environments, MultiRep must apply synchronization mechanisms to ensure the efficient use of network bandwidth, memory capacity and CPU power of these devices;
• Reconciliation of divergent replicas - divergent versions of data arise because several devices can store multiple replicas and change them multiple times without synchronizing with each other on each point of modification. MultiRep system should be able to identify which version is the latest version, which version covers the other version, and which versions can be merged;
• Dealing with conflicts - in an optimistic replication scheme based on a synchronization model without a master device or central server to control data modifications across multiple devices, conflicts are common. Therefore, the system should be able to identify conflicts correctly and resolve them if possible using the collected information taken from the monitoring process;
• Performance - evaluate a system with these characteristics is not easy. The system should be testable in order to conclude its feasibility.

1.3 Drawbacks of Current Solutions

Currently, there are many systems allowing users to synchronize files between multiple devices. These systems typically can be divided into two groups:
1. Online file synchronizers - these systems use cloud computing to store and share data among all devices. Examples: Dropbox[1], MobileMe\(^1\), SugarSync\(^2\), Box.net\(^3\).

2. Offline file synchronizers - synchronization of files can be done offline between different devices. Examples: GoodSync\(^2\), Microsoft’s SyncToy\(^27\), Microsoft’s ActiveSync\(^26\), Microsoft’s Briefcase\(^24\).

The first group of systems are the most widely used, because they offer more flexibility in a scenario where the user has several devices that can be connected to the network at any instant at all the same time. Nevertheless, these solutions only work if the user is connected to the Internet when he wants to synchronize. In addition, user’s data is stored in an unfamiliar environment that is not controlled by the user, thus raising privacy issues. Also, these systems do not take the advantage of the proximity of devices to share data among them, which is often more cost effective than synchronization over long-distance links with Internet services. The second group of systems has several drawbacks related with conflict resolution and network flexibility that are described in more detail in section 2.8. It is important to note that some systems, such as Windows Live Mesh 2011\(^4\), use an hybrid approach letting users decide if they want to synchronize data with local devices or online storage web services (e.g. Amazon S3\(^5\)).

All current solutions do not provide any information regarding the location of files and folders stored on several devices. Thus, when a user needs an object that is not stored on the device being used, he needs to manually explore the entire object collection which is spread across multiple devices.

1.4 Solution

The solution proposed in this paper is a middleware system, named MultiRep, that allows users to synchronize data asynchronously among multiple devices. This system is based on an optimist approach that uses mechanisms based on version vectors\(^34\) to deal with different files modifications across devices. With these mechanisms, MultiRep identifies conflicts and determines the set of updates to be exchanged during synchronization sessions. Techniques to reduce time and data propagation during these sessions are also implemented in MultiRep. Details of the implemented solution are presented in section 3.

1.5 Roadmap

The rest of this report is organized as follows. Section 2 describes the related work. Section 3 presents the architecture and the main algorithms of devised solution. Section 4 describes evaluation methods that will be used to evaluate the system. Finally, section 5 draws the conclusions of this paper.

---

\(^3\)http://www.box.net accessed in 28/11/2010
\(^5\)http://aws.amazon.com/s3 accessed on 28/11/2010
2 Related work

This section presents an overview of different approaches to maintain data consistency between devices. The remainder of this section is organized as follows. Section 2.1 describes the basic concepts used to explain the following sections. Section 2.2 discusses the limitations of pessimistic strategies to ensure consistent data sharing throughout multiple devices with no connection or intermittent connection to wide-area networks. Section 2.3 describes the most important architectural decisions used by many optimistic replication systems to propagate and order updates and to detect and resolve conflicts. Section 2.4 presents the trade-offs associated with different properties of the optimistic replication scheme. Section 2.5 addresses the main techniques to version timestamping for the optimistic replication approach. Finally, sections 2.6, 2.7 and 2.8 describe state-of-the-art solutions providing a comparative analysis of their most relevant features.

2.1 Basic Concepts

There are some concepts that are essential to understand the remaining sections of this article.

- **Object**: The minimal unit of replication in a replication system. It can be a file, directory, volume, database or simply a fragment of information.
- **Replica**: A copy of an object stored in a site.
- **Site**: Entity responsible for maintaining and managing replicas of one or multiple objects.
- **Reconciliation**: The process of detecting, managing and resolving conflicts, in order to produce a new consistent value.
- **Operation**: A self-contained update to an object. Operations differ from transactions (traditional databases) because they are propagated and applied in the background, often long after they were submitted by the users.

Note: Since most optimistic replication algorithms manage each object independently, the concepts of site and replica are used in an interchangeably manner.

2.2 Pessimistic Replication

Pessimistic replication model aims to ensure consistent data sharing throughout the system as if it were a single copy. The pessimistic replication algorithms cover inconsistencies between replicas by restraining their availability. They prohibit access to a replica unless it is provably up to date. In environments where network latencies are small and failures uncommon, as happens in a local-area network, these type of algorithms work well. However, this is not true when we try to apply these pessimistic strategies to a mobile environment or wide-area networks such as Internet, for three main reasons:

1. In the case of wide-area networks the main problem described by Chandra et al. is that they continue to be slow and unreliable. In addition, there are many mobile computing devices with no connection or intermittent connection to these types of networks. Furthermore, these devices can be behind firewalls or in different network partitions. In most cases, pessimistic replication algorithms do not work in such environments. For example, if we use a pessimistic replication algorithm, attempting to synchronize with an unavailable site, it would block indefinitely.
2. Pessimistic strategies don’t scale well - improving read’s availability, which requires increasing the number of replicas, implies reducing write’s availability. This happens because to update replicas contents pessimistic approaches need to coordinate these replicas in a lock-step manner.

3. Some human activities inherently demand optimistic data sharing. For example, when a team is developing software, most team members work concurrently and in relative isolation from one another. In these cases to improve collaborative work among different users, the system must allow concurrent edition of files. This last situation can not be achieved with a pessimistic approach in which multiple users have to wait to edit a file until it ceases to be used.

Despite these drawbacks, pessimistic algorithms are still used in many systems that require strong consistency. This happens because in several cases pessimistic replication is simpler to apply and sufficient to achieve system’s requirements. For example, primary-copy[45] is one of the most popular pessimistic algorithms used widely in many real-world solutions. It is based on the election of a primary replica that handles all accesses to an object. When a user submits an operation, the primary replica synchronously writes the update to other secondary replicas. Since updates are all submitted to one site, this algorithm can control data consistency in a centralized way. Nevertheless, it is a prohibitive choice for systems that require high levels of availability and scalability, due to the huge bottleneck in the primary replica regarding an increasing number of updates. Also, if the primary crashes, the time to elect a new primary may have a relevant impact on the system’s availability.

Given these limitations, we concluded that pessimistic algorithms do not meet the MultiRep’s requirements. For this reason, we will not describe in detail any pessimistic approach.

2.3 Optimistic Replication

Optimistic replication is a group of techniques for sharing data efficiently in wide-area or mobile environments [41]. It allows replicas to diverge assuming eventual consistency which means that replicas are guaranteed to converge at a certain period of time. As a result, users can access data without requiring the synchronization of all its copies, thus improving concurrency and parallelism among replicas. These benefits, however, come at a cost. Where a pessimistic algorithm waits, an optimistic one speculates [41]. The trade-off is the degree of inconsistency of data among replicas and the conflicts that may arise by the reconciliation process that is required for replicas convergence. Fortunately, in many real-world systems, especially file systems, conflicts are known to be rather rare [54, 53].

In sum, optimistic replication has several advantages over the pessimistic replication scheme, which are:

- **Availability** - Optimistic algorithms allow users to update concurrently data among replicas without having a synchronization mechanism that blocks user’s data access as happens in a pessimistic approach.
- **Networking flexibility** - Optimistic strategies support intermittent connections and network partitions that are common features in mobile environments. Many optimistic algorithms allow updates to be relayed through intermediate sites (called epidemic communication). These updates are reliably propagated to all replicas, even when the communication graph is unknown and variable.
- **Scalability** - Optimistic algorithms work well with a large number of replicas, because they require few coordination among replicas in the synchronization process. This is not true in pessimistic replication where most of the algorithms synchronously coordinate replicas during data accesses.

- **Asynchronous collaboration** - In multi-user collaboration systems, for instance in CVS[7], this type of replication allow users to work independently on the same project without the need to update their changes synchronously.

- **Site autonomy** - Optimistic algorithms often improve the autonomy of sites. For example, a new replica can be added to the system without any administrative change on existing sites (e.g. services such as FTP[30] and Usenet[19] mirroring have this feature).

- **Quick Feedback** - Optimistic algorithms can apply updates tentatively as soon as they are submitted. Thus, users can have a quick feedback of their updates, resulting in a better user experience.

Despite all these advantages, an optimistic replication does not fit in all types of systems. Applications requiring strong consistency models that do not tolerate occasional conflicts and inconsistent data can not be supplied with this model of replication. An optimistic replication algorithm consists of five phases: operation submission defines the number of sites where updates can be submitted. Also, it describes what is transferred between sites as an update; propagation defines how updates are detected and propagated by the systems; scheduling is related with the ordering of the updates among sites; conflict resolution addresses the problems related with concurrent modifications; commitment refers several algorithms to converge the state of sites. Some design choices that can be taken at these different phases are described in the following sections.

### 2.3.1 Operation Submission

In this phase, to update an object, a user submits an operation at some site. Afterward, the site locally applies that operation. This phase can be handled taking into account different requirements. For example, if the system allows users to submit updates at any site or not. This and other design choices are described below.

**Number of Writers: Single-Master vs. Multimaster**: Regarding where updates can be submitted, there are two types of systems: single-master and multimaster systems. Single-master systems are based on a master replica that handles all writes to all objects. In these systems, users can perform reads to any replica, but can only submit writes to the master. After a user's write, the master is responsible for propagating updates to other replicas. All replicas receive the update through the epidemic propagation model (pairwise interactions between replicas). These systems are simple since they can control data consistency in a centralized way. However, they have limited availability because the master replica is a single point of failure and becomes a bottleneck when users submit updates frequently. On the other hand, multimaster systems have higher availability but also greater complexity. In multimaster systems, all replicas behave as masters, so users can submit updates at any replica independently. A problem arising from the complexity of the system is its limited scalability due to the increasing rate of conflicts.

**Unit and Mode of Update Transfer: State vs. Operation**: There are two types of systems w.r.t.
what is propagated as an update: state-transfer and operation-transfer systems. In state-transfer systems, an update to an object implies the propagation of the entire object during the epidemic propagation process. In this process, when a replica receives an update from other replica, it overwrites the whole object to which the update refers. These systems are simple, because maintaining consistency only involves sending the newest replica contents to other replicas. However, they can be very inefficient, because small modifications to an object implies the propagation of the whole object which might not be necessary. Operation-transfer systems cover this problem by maintaining a history of operations performed to all objects, which can be stored as a log or database. Thus, when a replica must propagate an update it only needs to send the operations that other replicas do not have in their history log. This approach is more complex than state-transfer since all replicas must agree on the set of operations to be performed and their order. Nevertheless, it is much more flexible in conflict resolution and efficient when objects are large and operations high level.

Examples: DNS[28] is a single-master and state-transfer system. Usenet[19] and Bayou[9] are multimaster and operation-transfer systems.

2.3.2 Propagation

After the site locally apply the operation that was submitted by the user, it must propagate the associated update to other sites. The two models described below specify when each site decides to communicate and exchange updates with other sites.

**Pull vs. Push Model:** In the pull model, a replica polls other replicas for updates. The polling process can be triggered manually, for instance when a user pulls updates from a central repository, or automatically using periodic polling (e.g. FTP[30] mirroring, DNS[28] zone transfer). In the push model, the replica holding the update is responsible for pushing it to other replicas. So, when receiving an update the replica epidemically propagate it to all replicas using for instance the blindly flood technique[41]. This model can be triggered manually, for example when a user wants to push changes out to a central server, or automatically. One of the main goals of push model approach is to cover the overheads associated with the periodic polling. The choice between pulling and pushing is orthogonal to eventual consistency, but it affects the system’s scalability. It’s important to note that systems can use both models, for example DNS is a push/pull based system.

Examples: Rumor[13], Roam[39], Ficus[12] and Deno[17] are pull based systems. Usenet[19] is a push based system. DNS is both pull and push based systems.

2.3.3 Scheduling

In this phase, each site decides which order should be applied to the several operations. The main goal of scheduling is to produce equivalent states across replicas preserving the order of operations in a way expected by the users. There are two policies that can be used to schedule operations: syntactic and semantic. These polices are described below.
Syntactic vs. Semantic: Syntactic scheduling rely on general information about when, where, and by whom operations were submitted. Most systems use syntactic scheduling, ordering operations based on the time they happen. This type of ordering based on timestamps preserve the happens-before relationship defined by Lamport [20] between operations. Syntactic methods are much more simpler than semantic, but generally can cause unnecessary conflicts. For example, imagine we have a bag with just one ball and two users submitting the following operations ranked according to their temporal order: (1) user A requests a ball; (2) user B requests a ball; (3) user A adds the ball to the bag. If a replica schedules the requests syntactically then request of user B will fail, because the bag is empty. The semantic scheduling tries to cover this problem by exploiting the operation semantics. So, in the given example, if the replica now schedules the operations semantically, the order will be 1, 3, 2 and then the request 2 will now succeed. This works because operations 2 and 3 are semantically commutative (exploiting commutativity technique). When operations are not commutable, there is the possibility of using other semantic technique named operational transformation. This technique that was initially developed for collaborative editors [48] is only applicable to some systems. It is based on a set of rewriting rules (a rewrite rule for every possible pair of concurrent operations) to transform operations in order to run them in any order.

The main goal of semantic scheduling is to reduce the number of conflicts and increase the number of times that merging process of replicas is possible compared to the syntactic approach. For this reason, semantic techniques are known to tackle scheduling as an optimization problem. Icecube [18] is an example of a system that tries to optimize the scheduling of operations by applying several kinds of constrains to avoid a combinatorial explosion of all possible orderings. These semantic policies are more complex than syntactic and are only applicable to operation-transfer systems, since state-transfer systems do not work, by nature, with the semantics of operations. The choice between syntactic and semantic scheduling is related with the nature of the system and with the degree of flexibility and conflict resolution that we want.

2.3.4 Conflict Resolution

If there are any conflicts among the operations a site has scheduled, it must modify them in some way. The processes used to detect and resolve conflicts are presented below.

Detecting Conflicts: A conflict arises when a precondition of an operation is violated. In many systems this precondition is built implicitly into the replication algorithm. For example, in Coda [44] all concurrent operations are flagged to be potentially in conflict. These systems use a syntactic conflict detection approach, based on the happens-before relationship, to flag conflicts. So, a conflict is detected when two or more operations are concurrently applied. In other systems, this precondition can be explicitly defined by the user or application. For example, Bayou [9] let users explicitly assign a precondition (dependency check) to an operation. This precondition could simply be a test for a write-write conflict or something more complex. In these systems, the conflict detection is application-specific and is based on a semantic approach. State-transfer systems use both syntactic and semantic conflict detectors. On the other hand, operation-transfer systems typically use semantic conflict detectors.
*Resolving Conflicts:* Conflict resolution can be either manual or automatic. Usually, it is a very hard task and highly application-specific. Most systems are based on the manual conflict resolution by the user, so when they detect a conflict they simply flag it to the user. On the other hand, some systems can resolve conflicts automatically. In these systems, such as Bayou[9], Rumor[13] and Xmiddle[56], a *resolver* is called when a conflict occurs. The *resolver* is an application-specific component responsible for resolving conflict according to a set of rules written by the user or application. Another approach, used by many systems, to deal with conflicts is to simply ignore them; for instance, systems based on Thomas’s write rule[51]. However, this approach causes the lost-update problem.

### 2.3.5 Commitment

Scheduling and conflict resolution often make nondeterministic choice, which causes a certain divergence between sites. The main goal of the commitment process is to converge the state of replicas by letting sites agree on a final schedule and conflict resolution result. After this phase, all operations in the final scheduling are made permanent and therefore they cannot be rolled back anymore. The commitment process also acts as a space-bounding mechanism, because when the system needs space, it can safely remove the committed operations from its history logs.

There are two methods to perform the commitment phase: (1) with an agreement in the background or (2) by consensus. The first method uses some mechanisms, such as timestamp matrices[55] or TSAFE (Time-Stamped Anti Entropy)[11], to let each site learn the progress of others. These mechanisms allow sites to agree on the set of operations known to be received at all sites. The second method uses a consensus protocol. For example, the primary-based commitment protocol, used in Bayou, elects a single site as a primary which is responsible for deciding the commit order of operations. It’s important to note that there are some systems that do not need an explicit commitment algorithm, because they use totally deterministic scheduling and conflict-handling algorithms. In these systems (e.g. DNS[28], Usenet[19]), an implicit commitment process by common knowledge is done.

### 2.4 Properties of optimistic replication systems

When creating a system based on an optimistic replication scheme, there are some concerns that must be taken into account. The trade-off between availability and consistency as well as between performance and consistency, usually has a great impact on the system design. Different systems make different compromises among these requirements. Most of them implement different placement, consistency and topology policies for different environments. For example, Bayou[9] provides weak and strong consistency guarantees to applications, with sessions guarantees[50] and at the same time allows a flexible communication scheme between devices. However, it does not provide a partial replication scheme. As Bayou, most systems only meet two of the three properties presented below:

- **Partial Replication** - This property is very useful for a replication system that works in a mobile environment. It allows devices with limited memory capacity to store only a subset of all replicated data. These systems usually have a reference replica, with a large storage capacity, holding all data in the system.
• Arbitrary Consistency - Some systems allow users or applications to choose the degree of their consistency model. So, only the applications that want a strong consistency guarantee have to pay the overheads associated with it. On the other hand, several systems only provide weak consistency guarantees.

• Topology Independence - A system with a flexible communication scheme in which any node can exchange data with any other satisfies this property. Most systems restrict communication paths between devices with a client-server model.

Creating a system that provides all these properties is a very hard task. For example, imagine we have the following devices: a desktop storing files \(A\) and \(B\); a smartphone with only file \(B\); and a laptop with the same files of the desktop. At the beginning, the desktop performs a write to the file \(A\) (making it \(A'\)) followed by a write to file \(B\) (making it \(B'\)). Then, when the smartphone and desktop synchronize, for partial replication, the desktop sends only the write \(B'\) (and not the write \(A'\)) to the smartphone. Now, imagine that the smartphone and laptop synchronize. The smartphone only sends the write \(B'\) to the laptop. At this point, the laptop can present an inconsistent view of data to a user or application. This happens because a user or application can read the new version of file \(B\) (\(B'\)) and then the old version of file \(A\), thus violating the causal or even the weaker FIFO consistency. Most systems solve this problem simply by restricting one of the three properties described (full instead partial replication, strong or weak instead arbitrary consistency, client-server instead a flexible topology).

PRACTI[6] was the first system to provide all of these properties. It is important to note that a system based on optimistic replication does not have to provide all properties described above because, in most cases, it is not necessary to fulfill the application’s requirements.

2.5 Optimistic techniques

Optimistic replication systems allow multiple users to update data at different times. Due to this fact, these systems must employ mechanisms based on timestamps for tracking changes to data, in order to detect conflicts and the set of updates that must be exchanged between sites. Version vectors[34] and hash histories[16] are the two relevant mechanisms to version timestamping described in the following sections. Other mechanisms like Bayou’s version vectors[36] and dynamic version vectors[39] are only variants of the basic version vector approach and therefore they are not described in this section.

2.5.1 Version Vectors

A version vector[34] is a vector of \(n\) pairs, where \(n\) is the number of sites holding the data object associated with the vector. In this section, version vectors will be written as \(<A:5,B:2,C:10, ...>\) where letters \(A\), \(B\) and \(C\) designate site names and the numbers represent Lamport clocks[20]. A version vector contains a count of updates from each replica. Thus, a vector \(<A:5,B:2,C:10>\) indicates five updates by \(A\), two by \(B\), and ten by \(C\). Lamport clocks or logical clocks count the number of updates made to a given object \(o\) at a given site \(S\). Whenever a site \(S\) submits an update to an object, a new version vector is assigned to that object. This version vector is obtained by incrementing the entry corresponding to site \(S\) on the old vector by one.
Each time two replicas synchronize, the resulting vectors of both replicas are obtained through a *merge* operation. The *merge* operation returns a version vector which has for each entry the maximum value of the corresponding entry in the input version vectors. Conflicts are detected in the *merge* procedure when vectors associated with different updates are not *compatible*. A version vector \(vv1\) is *compatible* with \(vv2\), if \(vv1\) dominates \(vv2\) or \(vv2\) dominates \(vv1\). A vector \(vv1\) dominates \(vv2\) if all entries related to every site in \(vv1\) are greater or equal than corresponding entries in \(vv2\). This concept is closely related to the happened-before relationship[20]. For example, \(<A:2,B:5>\) dominates \(<A:1,B:5>\) and so we can guarantee that the second vector causally precedes the first, meaning that happened-before relationship links \(<A:2,B:5>\) to \(<A:1,B:5>\).

In the case of conflicts, there is not a happened-before relationship between vectors, for instance \(<A:2,B:5>\) and \(<A:1,B:6>\) are conflict versions.

Although version vectors or variants are used in many distributed file systems, such as Coda[44] and Ficus[12], to track updates, they still have some limitations. The size of version vectors is proportional to the number of sites in the system. Moreover, each vector is stored per-object at each site. Thus, a scalability problem arises when the number of sites becomes too big or the number of objects increases too much. This happens, because the storage and communication overhead to support version vectors becomes too costly. Vector sets[23], developed by Microsoft for WinFS[23], and hash history[16] approaches aim to solve this issue. W.r.t. timestamps for logged updates, the overhead is reduced by storing only a \([\text{siteID}, \text{timestamp}]\) pair, where \(\text{siteID}\) is the identifier of the site employing the update and \(\text{timestamp}\) is the logical clock of \(\text{siteID}\) that results from that update event. This pair is used as a version identifier for each entry in the log. Although this approach reduces the storage overhead compared with the version vector when increasing the number of replicas, it still has some limitations that will be discussed in section 2.5.2. This approach is used by Bayou[9].

The major limitation of basic version vector approach is the fact that the number of replicas is considered to be static. Dynamic version vectors[3, 39] and Bayou's version vectors[36] aim to cover this limitation.

As vector clocks[10, 40], version vectors are among the common techniques used to detect conflicts. Although these mechanisms seem similar, there are important differences between the update rules for version vectors and vector clocks. For instance, vector clocks count message exchanges as events unlike version vectors. Also, version vectors are commonly used to describe the state of each object on a site, while vector clocks are commonly used to describe the global state of a site.

### 2.5.2 Hash Histories

Hash history[16] is an alternative approach to version timestamping for reconciling mutual inconsistency in optimistic replication. This mechanism uses hash of the data contents, rather than timestamps, to represent the state of a replica.

In hash history approach, each site keeps a record of the hash of each version capturing a dependency among versions as a directed graph of version hashes (i.e., hash history). These hashes are calculated by applying a cryptographic hash function, such as SHA-1[32], to the entire contents of each version. A replica version is uniquely identified by its hash value that has a relatively small fixed size. Hash history approach makes use of these hash IDs for labeling each entry in the logs to extract deltas to be exchanged for the
reconciliation process. This approach has less or equal amount of the storage consumption than the version vector approach. During reconciliation, the sites exchange their hash histories that are used to determine the lastest version of a certain replica.

Unlike version vectors, the size of a hash history is independent of the number of sites. Nevertheless, it’s proportional to the rate and number of data modifications. Also, the hash history approach does not have any overhead related to the dynamic membership change of the collection of sites as happens in the dynamic version vector scheme. Moreover, it can recognize coincidental equality. A coincidental equality occurs when two versions, originated by different paths at a version history graph[16], are equal but neither dominates the other. For instance, imagine that sites C and D merge the two version vectors \(vv1 = \langle A:2,B:3,C:0,D:0,E:0,F:0\rangle\) and \(vv2 = \langle A:3,B:2,C:0,D:0,E:0,F:0\rangle\) resulting in the corresponding vectors \(vv3 = \langle A:3,B:3,C:1,D:0,E:0,F:0\rangle\) and \(vv4 = \langle A:3,B:3,C:0,D:0,E:1,F:0\rangle\). The two new versions represented by \(vv3\) and \(vv4\) could be identical relative to the data contents of each version, however, in the version vector scheme these versions are concurrent and so a conflict is reported. In this example, the hash history approach might consider both of these versions equivalent reducing the overhead associated with conflict resolution. Nevertheless, the probability event of coincidental equality is closely related with the update patterns of an application domain. Thus, for some systems this feature may not be relevant.

Despite the advantages presented, the hash histories has two drawbacks comparing to the version vector approach. First, when an update arises at a given site a new hash value has to be calculated over the entire contents of the data, even though this update consists of just a few bytes. For objects with a large size, this represents a significant performance overhead. In the version vector approach, it’s only required to increment a value in the corresponding vector. Second, the causal relationship between two different hash versions cannot be deduced by a simple comparison of both versions. This happens because there is no explicit causal information on the hash values. To identify the happens-before relationship between two versions, it’s required an examination of the [parent,child] pairs stored in the update log. On the other hand, using version vectors, a simple comparison of both versions is required.

2.6 Distributed File Systems

The main goal of a distributed file system (DFS) is to allow users of physically distributed computers to share data and storage resources by using a common file system[22]. These systems can also be regarded as replicated systems. In this case, replicated objects are files and directories. They are often replicated from server’s disk to client caches and across multiple servers to improve performance and availability, respectively. The advantages and disadvantages of the several DFSs from the perspective of sharing data among multiple devices are presented in the following sections.

2.6.1 Coda

NFS[43] and AFS[15] are distributed file systems based on a client-server architecture. These systems employ pessimistic consistency policies; therefore, on a server or network failure clients cannot have access to data. Both systems implement different caching mechanisms in the client. Unlike NFS, AFS does not generate network traffic with read and write operations performed on a cached file. In AFS, a cache manager, called
Venus, is responsible for copying files modified in the cache to the server when they are closed.

Coda[44] file system is a descendant of AFS that aims to provide high availability despite disconnected operation. It works like AFS when clients are connected to a set of accessible servers known as the available volume storage group (AVSG). When a client becomes disconnected it employs an optimistic consistency strategy that allows clients to perform read and write operations even if they are completely isolated from the servers. This is possible by exploiting the caching mechanisms handled by a cache manager (Venus).

In Coda, Venus operates in one of three states: hoarding, emulation, and reintegration. Venus is in hoarding state, when a client is connected to a server. In this state, Venus copies all critical objects, which are essential to the user’s work, from the server to the client’s cache. Critical objects are selected based on implicit and explicit information. The implicit information consists on history of recent file usage obtained by employing a traditional least recently used (LRU) cache substitution scheme. The explicit information is based on a set of pathnames that indentify critical objects. This information is specified by the user and stored at a hoard database. Venus is in emulation state, when a client becomes disconnected from a server. In this state, all operations performed by the client are logged as tentative updates. If the client tries to access an object that is not in cache a cache miss happens. This is seen by the user or application as a failure. When the connection to the server is available again, Venus is in reintegration state. In this state, the client modification log that stores all tentative updates is synchronized with all servers in AVSG. The hoarding state is then entered.

As in AFS, Coda’s clients are notified of changes via a callback mechanism. However, clients cannot be notified by this mechanism when they are disconnected from the servers. Thus, conflicts arise when two or more clients modify the same files or directories despite disconnected operation. Coda detects these conflicts with two approaches. The first one is a syntactic approach based on version vectors. A Coda Version Vector is attached to each file and used by the servers to check concurrent updates. The semantic approach is invoked only when the syntactic one is applied.

Although Coda has improved the availability of AFS with a disconnected operation model and replication of files across multiple servers, it still has several disadvantages. The main drawback that are important to note is the inability of users in disconnected mode to exchange data with each other. BlueFS[31] and EnsemBlue[33], which are an extension of Coda, attempt to cover this issue, however they still have a single central server.

2.6.2 Roam

Roam[39] is an optimistic replication file system that aims to provide a high scalability model for mobile computing. Roam is an extension to the Rumor[13] file system and therefore shares many of its characteristics. On the other hand, Rumor is a descendant of Ficus[12]. Although Rumor works better than Ficus for mobile environments, it shares many of its scalability issues. Roam solves Rumor’s scalability problem using a hybrid reconciliation model and dynamic version-vector management. It allows the synchronization between any two mobile devices using a peer-to-peer model. Thus, unlike Coda[44], it does not need a central server to propagate updates between devices. This feature enhances Roam’s availability. It also uses a partial/selective replication scheme, so each client’s cache contains only a subset
Roam’s architecture combines peer-to-peer and client-server models. The result of this combination is a hybrid architecture, called Ward Model, which is responsible for achieving the Roam’s scalability requirement. Ward Model is a cluster of wards (wide area replication domains). A ward is a group of nearby replicas. Replicas are grouped into a ward taking into a count a set of parameters (geographic location, bandwidth, etc.). All replicas within a ward are peers that can directly synchronize with each other. A ward master is elected in each ward. It can be seen as a server in client-server model, but with some peculiar differences. For instance, any ward member can be elected as the ward master, since all ward members are peers. Ward masters are responsible for maintaining consistency among wards.

An adaptive ring topology is used within and between wards to perform the reconciliation process. Reconciliation is based on a pull model approach. It consists of two main processes running on every ward member: recon process and server process. When reconciliation begins, the recon process requests metadata to the server process that runs on a remote replica. Metadata is then used to check which local files require updates regarding the remote files. This check is based on comparison of dynamic version vectors assigned to each file. Finally, the recon process fetches from the server process the files that need to be updated and then merge these files into the local file system. Conflicts occur when, during this process, files are concurrently updated. In this case, if a resolver is associated with the conflicting files, the conflict can be automatically resolved. Otherwise, user is notified to manually decide which files want to preserve.

In sum, Roam is a robust and flexible replicated file system that provides high scalability and availability requirements for mobile users. The main issue of this system is the number of reconciliations required for an update to reach all replicas. Moreover, these updates are only tentative and therefore applications that require more strong consistency guarantees cannot be used.

2.6.3 Haddock-FS

Haddock-FS[5] is a peer-to-peer replicated file system that supports co-present collaborative activities in mobile ad-hoc environments. It is an update log-based system that distinguishes between tentative and stable updates. Like Bayou[9], Haddock-FS uses a primary commit scheme, in which a single replica, called primary replica, is responsible for selecting new stable updates.

Haddock-FS employs mechanisms to reduce memory and bandwidth usage of mobile devices. It applies a bounded log truncation scheme that reduces the storage overhead of the update logs stored at each device. Moreover, it uses deduplication techniques, similar to those used by LBFS[29], to minimize the size of update data transferred during the synchronization process. Haddock-FS also provides an adaptable file system consistency model for applications with differing consistency demands.

Haddock-FS consistency protocol relies on dynamic version vectors[39] to detect conflicts. During update propagation, if a causal conflict between two tentative file updates is detected, a notification is sent to the replica manager and no action is taken by default for conflict resolution. Nevertheless, the author states that an application-specific conflict procedure can be installed in order to converge both replicas to a common state. W.r.t directory conflicts most of them can be automatically resolved by Haddock-FS, since they are system defined objects whose semantic is well-known.
Haddock-FS still has some limitations. For instance, all updates diverging from the stable path that are not consensually selected to become stable are discarded without providing any back-up to the user. Also, if the primary replica fails or becomes unreachable update committing can be severely disrupted allowing a wide divergence between replicas. This last drawback is related with the primary commit scheme employed by Haddock-FS.

2.7 Data-Sharing Middleware

The main goal of a data-sharing middleware system is to provide an abstraction interface for developers and applications to share consistent data among devices. Comparing to distributed file systems, the major advantage of this type of systems is related to their operating system interoperability. In other words, data-sharing middleware systems can be used by applications to store and manage replicated data regardless of the operating system or file system deployed in the device. Some examples of these systems are described below.

2.7.1 Footloose

Footloose\cite{33} is perhaps the closest system to meet the main goal of MultiRep. It is a single-user replication system that aims to provide a user-centered data store that can share data and reconcile conflicts across multiple devices. Footloose is an optimistic system based on two fundamental ideas: *physical eventual consistency* and *selective conflict resolution*. The first idea is based on a protocol that uses physical proximity of devices to enhance consistency. The second idea allows applications distributed among some of the devices to share data, even when conflicts occur during this process. These applications can choose to resolve conflicts or leave them for others to resolve.

Footloose assumes that sporadic connections between devices form a connected graph over time. It also assumes that devices only understand and can resolve conflicts for a few data types. Thus, it classifies devices into two classes: *smart* and *dumb*. For a given application, a device can be *smart* or *dumb*, depending on whether or not it can resolve conflicts for that application.

This system consists of two main components: Footloose Store (FLS) and Footloose Protocol Daemon (FPD). FLS is responsible for storing shared data of applications and for implementing the *selective conflict resolution* model. Every application that desires to store and read a specific kind of data must make a registration with the FLS. A device that has registrations is *interested* in that type of data. Interest registrations are shared among all of the devices and used for routing decisions. A device with no interest in a particular type of data can be used, under certain conditions, to carry such data to interested devices. All applications that wish to share data among devices place *UpdateEvents* in the FLS. Each *UpdateEvent* represents a change on a specific type of data. FLS communicate with FPD to read new data from other devices. FPD is responsible for implementing *physical eventual consistency* and no lost updates using *StatusVectors*. *StatusVectors* are a variation of traditional version vectors that contains one-byte element for every device. With this mechanism, FPD can detect if two devices need to exchange information about a given *UpdateEvent*. Moreover, *StatusVectors* indicate which files are no longer needed and therefore can be collected by the garbage collector.
Although Footloose performs relatively well on some topologies (e.g. clique, star, ring) with a considerable number of devices, it still has some limitations. For instance, Footloose does not work with dynamically changing device sets, because Status Vectors are built with a fixed size. Moreover, Footloose does not provide a file system interface to the applications, so it forces them to use the FLS’s interface for storing and sharing data. Also, it maintains UpdateEvents in FLS until all interested devices have seen such updates, which could be a problem for devices with limited memory capacity. Furthermore, it has no system-wide knowledge of the locations of the most recent pieces of data, which could be very useful for users and applications.

2.7.2 Eyo

Eyo[46, 47] is a single-user metadata-everywhere storage system based on a new principle, called device transparency. Device transparency is based on the idea that user’s data should be managed transparently by the user from any of his devices. So, Eyo, that is also an optimistic replication system, aims to provide on each device a global consistent view of all user’s data objects distributed among devices. To achieve this, no manual synchronization or central server is required. Also, it does not need to store the entire data collection on each device to provide such global view. Instead, it stores on each device the metadata of all objects in the system, separating object metadata from content. This metadata contains two types of attributes: storage system attributes describing which devices hold which objects; application-specific attributes such as names and types.

The metadata-everywhere approach has several advantages. First, it gives at any device a global view of all data spread across devices. Second, it allows users or applications to identify, move, and search objects in the system from any device, even when these objects are currently stored in inaccessible devices. Finally, it enables applications to inform the user when an object, which was not yet obtained by the device, has a recent and possibly conflicting update. Metada-everywhere approach is feasible for mobile devices, because the metadata associated with each object is roughly constant regardless the object data type [47]. Also, the memory capacity of these devices has grown increasingly.

In Eyo, applications create and manipulate objects with the Eyo’s storage API. Each object has associated a unique ID and a directed acyclic graph of object versions. Edges in the version graph denote parent-child relationships between those versions. When an object is modified by an application it creates a new version specifying one or more parent versions. A conflict arises when two different versions of the same object have the same parent version. In this case, the conflict resolution is the sole responsibility of applications. Eyo’s API enables applications to specify placement rules, as in Cimbiosys[38] and Perspective[42], to indicate on which device a certain object must be stored. Status notifications also defined using storage API are used to inform applications of new updates.

Eyo reconciliation process is based on metadata and content synchronization. Metadata is synchronized continuously in the background. In this process, the new metadata and update log is exchanged between devices using an overlay network. The content is synchronized after the metadata synchronization.

Cimbiosys and Perspective are the two systems most closely related to Eyo. Nevertheless, neither attempts to provide a complete view over the data collection from disconnected devices or aid applications in reasoning about object version histories. Also, Roma[49] attempts to locate current versions of personal
files and ensure their availability across different repositories. However, it is not totally decentralized since it maintains a single central server with all metadata in the system.

Although Eyo provides a totally decentralized model based on device transparency that has several advantages, it has some limitations. For instance, in Eyo, the smallest device, in terms of storage capacity, in each group must be able to hold a copy of all metadata of that group, which may not be possible if the amount of metadata and number of devices are too large. Moreover, Eyo assumes that data objects will be larger than metadata, which may not be true for a certain data types used by applications. Like Footloose, Eyo does not provide a file system interface to applications.

2.8 File Synchronizers

A file synchronizer[4] is a user level tool responsible for reconciling replicated files and folders of the user among multiple devices. Usually, these systems do not attempt to present a “single file-system” semantics: users are aware that their files are replicated and that updates are only propagated between devices when a file synchronization process is initiated (often by the user). A file synchronizer is divided into two main components: the update detector and the reconciler. The update detector is responsible for the recognition of the updates to be exchanged between devices since the last point of synchronization. These components can be implemented by different update detection strategies as Balmubramaniam et al.[4] described (e.g. simple modtime update detector that uses the “last modified time” provided by operating systems for update detection). The reconciler is responsible for combining the updates between devices in order to maintain files consistency. When merging different file versions the reconciler faces six types of conflicts most of them defined by Unison[37]:

1. Change the contents of the same file on different replicas
2. Create the same new file with different contents on different replicas
3. Change a file inside a folder on one replica and delete that folder on another
4. Change the name of one file/folder to different names in different replicas
5. Delete a file on one replica and change its contents on another
6. Delete a file/folder on one replica and change its name on another

There are two types of file synchronizers: offline file synchronizers and online file synchronizers. The first type, unlike the second, does not need access to an Internet service for maintaining files synchronized between devices. Current existing solutions based on these two types are described below.

2.8.1 Dropbox

Dropbox[1] is an online file synchronizer that allows users to share and store files and folders with others across the Internet. It uses cloud computing, for file synchronization between multiple devices.

Dropbox creates a shared folder that is automatically uploaded to the cloud and replicated between devices with the Dropbox client installed. This client, available for different operating systems, is responsible for monitoring the shared folder. Any files dropped into this folder will be indexed, hashed and then compared to other files on the Dropbox’s servers. If the file does not exist on the servers, it is uploaded to the cloud and other devices. If the file already exist, a conflict arises and the two conflicting versions of the file are
stored at the servers. Dropbox also enables users to upload files manually through its website.

This system also can be used for collaborative working between multiple users. Thus, a user can define, through Dropbox's website, a set of permissions to allow others to access specific folders. To help users recovering deleted files or simply see the history of files modifications, a version control system is also implemented in Dropbox. The version history is paired with the use of delta encoding technology. To save bandwidth and time, Dropbox only uploads/downloads changes made to the file instead the entire file. A *binary diff* is employed between the old and the new version of the file to obtain only the pieces that must be uploaded. Dropbox also compresses these changes before transferring them to the *cloud*.

Although Dropbox works very well and is one of the most popular file synchronizers, it has some drawbacks. In Dropbox, disconnected clients cannot share updates with each other. It depends strongly on the Internet connection to share updated resources without taking the advantage of physical proximity of devices. Another drawback is related with the lower storage capacity of mobile users. Dropbox does not allow users to select the most critical files to be stored in case there is no more space left on device. Also, when conflicts occur, Dropbox does not provide any relevant information to the user about them. The user must manually resolve the conflict without having any help provided by the system. Furthermore, Dropbox can only detect conflicts 1 and 2 presented in section 2.8.

2.8.2 Microsoft’s Briefcase

Microsoft’s Briefcase[24] is an offline file synchronizer that was introduced in Windows 95. The new versions of Windows maintain this tool to let users keep their files synchronized between devices. Thus, when users want to replicate their files from a device to another, they just need to create a *briefcase* folder, then drop target files into *briefcase* and finally copy it to another device (by using a removable drive, for instance). The file synchronization process of Briefcase is manually initiated by the user. In particular, a user can choose the files or folders that wants to synchronize. The conflict resolution is also done by the user. For example, when two files are concurrently modified, Briefcase allows users to choose the version they want to maintain.

Briefcase’s file synchronization mechanism has several problems. First, it does not propagate the creation of folders until they have a file inside. Second, it does not report any information to the user about conflicts presented in section 2.8. Third, if a folder or file is renamed Briefcase is not able to detect that it is still the same folder or file. In most cases, it splits the folder or file from the original, rendering it an orphan. In most of these problems, the two file systems are not identical after synchronizing the *briefcase* folder. Smart Briefcases[21] is a solution based on the Microsoft’s Briefcase that aims to cover these limitations.

Although Briefcase is relatively good to maintain files synchronized between two or more devices, it has some drawbacks. For instance, imagine that a user copies a *briefcase* folder with the same contents from a device A to devices B and C. The devices B and C can reconcile files and folders with device A. However, they cannot propagate updates between them.

2.8.3 GoodSync

GoodSync[2] is a backup and an offline file synchronizer developed by Siber Systems that works with Windows and Mac. It provides a graphical user interface (GUI) that allows the creation of several *jobs*. A *job* consists
of several synchronized pairs of folders representing the user task behind the goal of synchronization. Several jobs can synchronize one folder to many other folders. The synchronization of a job can be scheduled or manually initiated by the user. GoodSync allows the synchronization directly between computers, removable storage devices, and Windows Mobile devices without requiring access to any Internet service. Nevertheless, it can also synchronize to cloud storage servers like online file synchronizers. Like Microsoft’s Briefcase, GoodSync can synchronize files between two disconnected devices using a removable storage device (e.g. USB disk) as an intermediary. Furthermore, it allows the synchronization between two devices over the Internet.

GoodSync does not require a file system monitoring to detect changes in files. Changes are detected by comparing current file status to the stored file status. Most conflicts presented in section 2.8 are reported by GoodSync. The conflict resolution is manually done by the user. A log history of the actions that were taken in each pair of folders are described by GoodSync’s GUI.

Although GoodSync is an excellent file synchronizer, it has some drawbacks. For instance, when a user wants to synchronize files directly between two devices, he must decide which device is the client and which is the server. Moreover, on the server side, he must configure the server’s properties and share individual folders using the sharing and security options provided by the operating system. Also, GoodSync does not provide any process for sharing files among multiple devices. If a user wants to synchronize a set of files between several devices, he must create a job for each pair of devices. Additionally, GoodSync does not offer any help for managing files across several devices, since a user does not know on which device a certain file is stored. Also, it cannot detect conflicts 4 and 6 presented in section 2.8.

2.8.4 SyncToy

SyncToy[27] is an offline file synchronizer built by Microsoft using the Microsoft Sync Framework[25]. As in GoodSync[2], SyncToy’s file synchronization process is performed by creating a folder pair that represents the two folders (“left” and “right” folders) to be compared and synchronized. These folders can be stored on a local drive, on an external device, or on a network share from another computer. A folder pair is created using the SyncToy’s graphical user interface.

SyncToy defines three modes of file synchronization: Echo, Synchronize and Contribute. Echo works like a standard one-way sync, copying all changes that have been made on the “left” folder to the “right” folder. Synchronize propagates these changes on both directions. Contribute works like Echo but it does not propagate deletes of files, only renames. During the synchronization process which is manually initiated by the user, SyncToy stores snapshots of each folder. A snapshot contains information about each file such as size, date of the last synchronization and hashes of file contents.

Although SyncToy can detect most conflicts presented in section 2.8, it has some limitations. For instance, when a conflict occurs, SyncToy does not report any information about the conflict to the user. Moreover, in case of conflict, it does not allow users to choose which version of a file they want to keep. If two files have been concurrently modified, SyncToy only offers two actions: overwrite one file by another only in one direction (e.g. from “left” to “right”); ignore modifications performed on both files. Like GoodSync, SyncToy does not provide any tool to help users synchronize and manage files across multiple devices. Also, SyncToy
cannot detect conflict 6 presented in section 2.8.

2.8.5 ActiveSync

ActiveSync\[26\] is a mobile data synchronization technology and protocol developed by Microsoft. It allows mobile devices to synchronize several types of data with a computer, using a USB\[6\], Infrared\[7\] or Bluetooth\[14\] connection. Starting with Windows Vista, ActiveSync has been replaced with the Windows Mobile Device Center\[8\]. Windows Mobile Device Center works similarly to ActiveSync but it uses the Microsoft Sync Framework which allows some new features.

In ActiveSync, data synchronization process is automatically triggered when two devices, a mobile device and a desktop PC or server, are connected together. During this phase, conflict resolution is performed automatically without user intervention. ActiveSync deals with most conflicts described in section 2.8, as well as conflicts that can occur when mobile devices do not have enough space to store data from other devices.

Although ActiveSync enables users to keep files synchronized between a desktop PC and several mobile devices, it does not allow data synchronization directly between mobile devices. Also, it does not report any conflict to the user. It only provides two options for conflict resolution: overwrite files on the handheld device; overwrite files on the desktop PC. Also, since it is a state-transfer system w.r.t. update propagation, it is not able to save network resources of portable devices during the synchronization process. For instance, when a user modifies only a few bytes of a file on a mobile device, ActiveSync sends the entire file to the desktop PC instead of sending only the changed bytes.

2.9 Summary

The previous sections described several solutions that perform data replication among multiple devices while enforcing consistency.

W.r.t. the main goals of MultiRep, we conclude that most solutions do not provide any mechanism to help users manage and resolve conflicts of files and folders stored in multiple devices. Additionally, many of these systems ensure file consistency with central servers or cloud storage services restricting communication paths between devices. Also, most current solutions do not allow users to view at any device the files and folders that are stored by other devices. Although some systems, such as Eyo, provide such information, in general they do not export a file system interface and thus cannot be used with legacy applications.

Table 1 in page 22 presents a summary of some studied systems w.r.t. optimistic properties mentioned in previous sections. Table 2 in page 23 presents a summary of the studied file synchronizers taking into account the main goals of MultiRep.

---

6http://www.usb.org/home accessed in 01/12/2010
7http://science.hq.nasa.gov/kids/imagers/emis/infrared.html accessed on 01/12/2010
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
<td>Distributed File System</td>
<td>Distributed File System</td>
<td>Distributed File System</td>
<td>Data-sharing middleware</td>
<td>Data-sharing middleware</td>
</tr>
<tr>
<td>Operations</td>
<td>State-based transfer for files, operation transfer for directories</td>
<td>State-based transfer for files, operation transfer for directories</td>
<td>State-based transfer for files, operation transfer for directories</td>
<td>N/A</td>
<td>State-based transfer for data, operation transfer for metadata</td>
</tr>
<tr>
<td>Propagation</td>
<td>Hybrid</td>
<td>Pull</td>
<td>Pull</td>
<td>Push</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Scheduling</td>
<td>Syntactic for files, semantic for directories</td>
<td>Syntactic for files, semantic for directories</td>
<td>Syntactic for files, semantic for directories</td>
<td>N/A</td>
<td>Syntactic</td>
</tr>
<tr>
<td>Versioning mechanism</td>
<td>Static version vectors</td>
<td>Dynamic version vectors</td>
<td>Dynamic version vectors (current implementation uses static version vectors)</td>
<td>Status vectors (a variation of static version vectors)</td>
<td>Version graphs for data, static version vectors for metadata</td>
</tr>
<tr>
<td>Resolving conflicts</td>
<td>Automatically for directories, manually for files</td>
<td>Automatically for directories, manually for files</td>
<td>Automatically for directories, manually for files</td>
<td>Conflict resolution is totally handled by applications</td>
<td>Conflict resolution is totally handled by applications</td>
</tr>
<tr>
<td>Allow the deployment of an application-specific conflict resolver</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>NA</td>
<td>No</td>
</tr>
<tr>
<td>Partial replication</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Yes (although metadata is fully replicated, data is partially replicated)</td>
</tr>
<tr>
<td>Arbitrary consistency</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Topology independence</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the studied distributed file systems and data-sharing middleware systems w.r.t. optimistic replication techniques. 'N/A' means that the information is not available in this case.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platforms</strong></td>
<td>Windows, Mac, Linux, Android, iPhone, iPad, BlackBerry</td>
<td>Windows</td>
<td>Windows, Mac</td>
<td>Windows</td>
<td>Windows, Windows Mobile</td>
</tr>
<tr>
<td><strong>Type of synchronizer</strong></td>
<td>Online</td>
<td>Offline</td>
<td>Offline/online</td>
<td>Offline</td>
<td>Offline</td>
</tr>
<tr>
<td><strong>Support for synchronizing handheld devices</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Backup Service (users can restore old data from a version history)</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Easy to sync data between multiple devices</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Provide an explicit conflict notification</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>In case of conflicts</strong></td>
<td>Keep the two conflicting versions of the file and do nothing</td>
<td>Allow users to choose which file version they want to keep</td>
<td>Alert users and allow users to choose which file version they want to keep</td>
<td>Allow users to choose which file version they want to keep</td>
<td>Automatically resolve conflicts - overwrite changes from the handheld device to the desktop or vice-versa (previously configured by the user)</td>
</tr>
<tr>
<td><strong>Provide relevant information regarding conflicts to help users resolve them</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Correctly handle renaming/deletion of files and folders</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Users can view and manage their entire data collection from any of their devices</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Topology independence</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Run without modifications to applications</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2: Comparison between the studied file synchronizers w.r.t. the main goals of MultiRep
3 Architecture

This section presents the architecture of the solution proposed in this dissertation. All architectural decisions taken during the design of the MultiRep system aimed to fulfill the following requirements:

- Consistency - MultiRep must ensure eventual consistency of replicated files and folders on multiple devices. It should be able to converge different replica versions;
- Handling conflicts - it must detect all conflicts presented in section 2.8 by employing a mechanism to timestamp replica versions. Also, it should be able to resolve conflicts whenever possible. If it cannot resolve a conflict automatically, MultiRep must provide all relevant information to the user for manual conflict resolution;
- Network flexibility - it must employ a file synchronization mechanism totally decentralized that allows users to synchronize data between any device;
- High availability - users should be able to change files on any device, anytime, even when these devices have no network connection;
- Partial replication - users should be able to store only a subset of data at any device;
- Efficient memory and bandwidth usage - it must employ mechanisms to reduce memory and bandwidth usage during the pairwise synchronization scheme;
- Performance - it must monitor access to files by the user and perform all metadata operations efficiently, without affecting system performance;
- Independent - the deployment of the MultiRep system should not involve any change to the OS kernel or any other application;
- Transparent - when no conflicts arises it must be transparent to the user;

The remainder of this section is organized as follows. Section 3.1 presents an overview of the MultiRep's architecture. Section 3.2 describes the module decomposition and data structures of the architecture.

3.1 System Overview

MultiRep is a peer-to-peer file sharing middleware that allows a single-user to keep data replicated and consistent among multiple devices. MultiRep is an extension to the Smart Briefcases[21] and therefore shares many of its characteristics. Despite Smart Briefcases has improved many features of Microsoft’s Briefcase[24], it still does not allow the synchronization process between any pair of devices. MultiRep aims to cover this limitation.

Using MultiRep is very simple. First of all, a user creates a special folder, called briefcase, as he would create a normal folder. Then, he copies all files to be replicated into briefcase. Finally, he copies the briefcase folder to other devices (e.g. using a SD card) in which he wants to keep files synchronized. Since the interface and usage of briefcase folders is similar to the ones offered by the current file systems, users and applications can transparently manage their data within a briefcase as in a normal folder.

MultiRep keeps track of the relationship between files and folders on multiple devices. To do this, when briefcases are copied among devices, MultiRep stores the metadata of the several synchronization pairs created during this process. So with this information, it knows which devices can synchronize a given
Therefore, when using the MultiRep system, users can synchronize their briefcases between any pair of devices.

Also, with MultiRep, users can change files and folders inside a briefcase at any disconnected device. MultiRep logs all actions performed on each device by the user, storing metadata that describes all relevant information about files and folders required for the synchronization process. Also, it employs a version timestamp mechanism based on version vectors[34] to detect conflicts and determine the set of updates to be exchanged between replica managers.

Users can also view on any device the files and folders stored in all briefcases maintained by other devices even if they are turned off or inaccessible. To achieve this, MultiRep stores in each device the metadata of all other devices that directly or indirectly participated in pairwise synchronization sessions. For instance, if a device $A$ synchronizes with device $B$ that previously synchronized with device $C$, then device $A$ must contain all metadata w.r.t. files and folders stored in devices $B$ and $C$.

When a user wants to synchronize all data modified inside a briefcase, between two connected devices, he can start the MultiRep's file synchronization process. During this process, all conflicts detected that cannot be automatically resolved by the system are reported to the user. In this case, MultiRep provides all relevant information to help users resolve conflicts. In particular, a description of the conflict is presented and some options for conflict resolution are provided. These options allow users to choose which version of the file they want to keep. Additionally, diff engine tools for plain text files are available for detecting the differences between the contents of files in conflict. MultiRep also allows users to postpone the conflict resolution.

It’s important to note that MultiRep does not need access to any central server or Internet service during the synchronization process, since all replicas are peers.

### 3.2 Layers, Modules and Data Structures

This section describes the module decomposition and structure of the MultiRep middleware. The architecture of the system, illustrated in figure 2, is divided into 4 main layers: GUI Layer, Consistency Layer, Monitorization Layer and Network Layer. Each layer is composed of several modules. The responsibility of each layer and module, along with the data structures and algorithms used, is explained below.

#### 3.2.1 Monitorization Layer

This layer is responsible for handling modifications performed by users and applications in the file system. In order to achieve this, the Monitorization Layer is divided into three main modules:

- **File System Monitor** - module responsible for monitoring all changes performed to files and folders stored inside a given briefcase. When a briefcase is created a File System Monitor is associated with that briefcase.
- **Drive Monitor** - this module is responsible for monitoring the creation of new synchronization pairs and the creation, renaming or deletion of briefcases. MultiRep associates a different Drive Monitor to each drive in the user’s device.
• Drive Detector - module responsible for detecting when a removable drive is inserted or removed on the user’s device. It is also responsible for triggering the creation and association of a Drive Monitor to watch over that drive.

All changes to the briefcases are reported by the Operating System Layer to the Drive Monitor or File System Monitor. These modules send all relevant information about these changes to the Metadata Manager.

A synchronization pair between two remote replicas is created when a given briefcase is copied from one device to another. The Drive Monitor is responsible for handling this process using the Network layer to establish the connection between two devices. Figure 3 illustrates this process.

Figure 2: MultiRep Architecture

3.2.2 Consistency Layer

This layer guarantees the eventual consistency of files and folders stored inside a briefcase. In order to achieve this, the consistency layer is mainly composed of the following modules:

Figure 3: The creation of a synchronization pair between two remote replicas. The syncPairRequest message contains all network information of replica A. The syncPairResponse message contains all metadata of the briefcase required to synchronize files and folders.
• Reconciliation Manager - module that implements the synchronization process. It detects all conflicts that occur during this process.
• Metadata Manager - module responsible for storing and managing all metadata required for the synchronization process. It also maintains the metadata of all briefcases stored in devices previously synchronized.
• Version Manager - module that implements the version vector mechanism for tracking changes to data on multiple devices.
• Conflict Resolver - module responsible for conflict resolution. It also stores all metadata associated with conflicts.
• Diff Engine Modules - this module is used to detect the differences between the contents of files in conflict. Since this module is independent from others, the MultiRep system allows developers to add new difference engines to read contents from specific types of files.

The synchronization process implemented by the Reconciliation Manager can be triggered by two layers: GUI Layer - whenever a user or application uses the MultiRep’s GUI to start data synchronization with other devices; Network Layer - whenever it receives a synchronization request from a remote replica. When the Reconciliation Manager starts the synchronization process, it uses all data structures maintained by the Metadata Manager to compare and update files/folders in the briefcases being synchronized. Different versions of files are compared using the version mechanism implemented by the Version Manager. When the Reconciliation Manager detects a conflict during this process, it invokes a specific Conflict Resolver. The Conflict Resolver can use Diff Engine Modules to display the contents of two replicas of the same file side-by-side.

Every time the Metadata Manager receives information from the Reconciliation Manager or Monitorization Layer regarding any modification performed to a given briefcase, the Metadata Manager updates its data structures using the Version Manager to timestamp the new versions created during this process.

It is important to note that the Version Manager uses the version vector approach instead of using the hash history approach for two reasons. The first reason lies in the fact that, in contrast to version vectors, hash values carry no explicit causal information. The second reason is related to the performance overhead regarding the calculation of hash values for each replica version that is created. Although version vectors have a variable size depending on the number of replicas, in MultiRep, the storage overhead w.r.t. this mechanism is not significant, since the number of devices owned by a single user is typically low.

3.2.3 Network Layer

The Network Layer allows communication between multiple instances of MultiRep running on different devices. It is responsible for opening the communication channel between replicas, as well as storing all network information about them. It’s important to note that all replicas are peers that exchange messages with each other directly. For this reason, MultiRep does not require any overlay network combining all devices for update propagation.
3.2.4 GUI Layer

The graphical user interface (GUI) of MultiRep is similar to the one used in the Smart Briefcases system. MultiRep’s GUI should provide a menu with several options to synchronize user’s briefcases, as well as a conflict resolution window with the description of the conflict and the several options for its resolution.

3.2.5 Data Structures

Most structures required for the synchronization process are stored and managed by the Metadata Manager. These structures are organized as follows:

- File and folder structures - objects that store all metadata of files and folders stored inside a briefcase. Each of these structures has an associated version vector for tracking changes to data, as well as a unique identifier for recognizing a given file or folder within a briefcase regardless of its name.
- Directory tree structure - a tree representing the hierarchical structure of files and folders inside a briefcase. Directory trees are used to compare which files and folders are new, renamed or deleted during the synchronization process. These trees are composed by file and folder structures.
- Synchronization pairs structure - structure that maintains a list of all briefcases stored inside user’s device. Also, it stores all information regarding the synchronization pairs created between local and remote briefcases.
- Briefcase structure - this structure keeps information regarding renaming or deletion of a briefcase during disconnected operation. When the system connection to the network is resumed, all modifications performed to that briefcase are propagated to other devices. This structure contains a version vector for tracking these modifications and an identifier.
- Devices structure - structure that maintains the metadata of all briefcases stored by devices that directly or indirectly participated in pairwise reconciliation sessions.

4 Methodology and Evaluation

This section describes the main measurements and methods used to evaluate the MultiRep system. The evaluation of MultiRep will be performed by three assessment processes: quantitative, qualitative and comparative. The quantitative measures that will be taken into account to evaluate the system are:

- Memory usage - the size occupied by all data structures presented in section 3.2.5. It is important to measure the relation between the increase number of files and folders stored inside a briefcase and the memory used by the application.
- Bandwidth usage - the number of exchanged messages and the amount of data propagated during the synchronization process.
- Performance - the performance of the monitoring process and synchronization process. These processes should not create a significant overhead w.r.t. the throughput of the system.
- Conflict handling - the number of conflicts detected, how many are resolved automatically by the system and how many have to be handled by the user.
• Updates required to reach eventual consistency - number of updates required to converge the several replicas in the system to a common state for different network topologies and number of devices.

The qualitative evaluation will assess the transparency of the system and the benefits of MultiRep for managing user data across multiple devices.

• Usability - The system must be evaluated by the users in terms of learnability and efficiency. MultiRep must provide all relevant information about the state of the application and the conflicts that occurred during synchronization without disturbing the user activities.

The comparative evaluation will consist in a comparison between the quantitative and qualitative results obtained from the previous evaluations and the analogous results collected from the evaluation of Smart Briefcases.

5 Conclusion

Most current solutions allow users to synchronize data among multiple devices through central servers or Internet services. However, sometimes portable devices have no network connection to access these services. Due to this fact, a system like MultiRep that take advantage of proximity of devices to share data among them, is required to let users keep exchanging information between any device anywhere anytime.

This document has provided an overview of the current state-of-the-art technologies for sharing data between multiple devices using optimistic approaches to ensure data consistency. First of all, the main properties of optimistic replication systems were described, as well as the advantages and disadvantages of several version timestamping mechanisms to detect conflicts and determine the set of updates to be exchanged between sites during reconciliation sessions. Then, relevant systems were discussed, compared and divided into three categories: Distributed File System, Data-Sharing Middleware and File Synchronizer. Finally, this document also presented an overview of the preliminary MultiRep’s architecture. MultiRep is a single-user file synchronizer based on version vectors for tracking changes to files and folders on multiple devices. It allows users to synchronize files and folders between any device reporting all relevant information about conflicts that occur during this process.

References


## Appendix: Work Scheduling

The following table presents the proposed schedule for the Dissertation Course.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Date Start</th>
<th>Date End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improve architecture design</td>
<td>13/01/2011</td>
<td>01/02/2011</td>
</tr>
<tr>
<td>Selection of tools</td>
<td>01/02/2011</td>
<td>15/02/2011</td>
</tr>
<tr>
<td>Implementation</td>
<td>15/02/2011</td>
<td>01/05/2011</td>
</tr>
<tr>
<td>Evaluate system</td>
<td>01/05/2011</td>
<td>17/05/2011</td>
</tr>
<tr>
<td>Write final report and article</td>
<td>17/05/2011</td>
<td>20/07/2011</td>
</tr>
<tr>
<td>Thesis review</td>
<td>20/07/2011</td>
<td>30/07/2011</td>
</tr>
<tr>
<td>Thesis delivery</td>
<td>30/07/2011</td>
<td></td>
</tr>
</tbody>
</table>