

Coherency Management in Ad-Hoc Group Communication

Malika Boulkenafed, Valérie Issarny
INRIA-Rocquencourt
{Malika.Boulkenafed,Valerie.Issarny}@inria.fr

Abstract: Among other features, pervasive computing aims at offering access to users' data, anytime, anywhere from whatever available terminal that is the most convenient in a given situation. However, realizing such a vision necessitates several improvements in the way servers and users terminals interact. In particular, users terminals should not tightly rely on an information server, which can be temporarily unavailable in a mobile situation. They should rather exploit all the information servers available in a given context through loose coupling with both fixed servers and peer mobile terminals. This paper concentrates most specifically on enabling information sharing among trusted mobile terminals using ad-hoc networking. It introduces a coherency management protocol for mobile ad-hoc collaborative groups. The protocol lies in the integration of both conservative and optimistic coherency management in a way that minimizes communication and hence, energy consumption while not restricting the nodes' autonomy.

Keywords: Distributed systems, coherency management, ad-hoc networks, mobile computing, wireless networks.

1 Introduction

It is now commonplace for a person to use a number of personal computing devices (e.g., laptop, hand-held, mobile phone, digital camera), and to make his/her data available to other people's devices. It is further becoming common for a user to find mobile computing devices equipped with low-power, short-range wireless connectivity such as Bluetooth¹, IEEE 802.11, IEEE 802.11a, HiperLAN, OWS (Optical Wireless Solution) and HomeRF in the context of PAN (Personal Area Network). This connectivity among devices enables users to work in a collaborative manner through different networks. Among them, ad-hoc networks are the most flexible because they enable users to cooperatively form a dynamic and temporary network without any pre-existing infrastructure. Therefore, what needs to be achieved is to enable seamless access to, possibly shared, *coherent* data, anytime, anywhere from whatever available terminal that is the most convenient in a given situation. Once a group (e.g., work meeting group) is formed, its members can share and manipulate common data. Three important features must then be offered to the group's members: (i) a secure way of sharing data; (ii) a coherency protocol to deal with concurrent updates; (iii) energy saving for mobile devices.

Security in ad-hoc network may be achieved using either symmetric or asymmetric cryptography [18]. The coherency management protocol must ensure that users access fresh data,

¹Bluetooth. <http://www.bluetooth.com>

and it should deal with conflicting operations, which may occur quite frequently in an intermittently connected environment, by detecting and resolving them. The protocol must further account for energy saving aspects. Indeed, mobile terminals that will soon be available will embed powerful hardware (e.g., LCD screens, accelerated 3D graphics, high performance processor), and an increasing number of devices (e.g., DVD, CD). However, the capacity of batteries goes up slowly and all these powerful components reduce battery life. Thus, it is mandatory to devise adequate solutions to energy saving on the mobile terminals. In particular, communication is one of the major sources of energy consumption [11].

Our coherency management protocol is integrated within a distributed system, called AdHocFS [19], that provide the following features. AdHocFS runs on both mobile and stationary heterogeneous machines including wireless handheld devices such as PDAs. It offers peer-to-peer replication. However, each data should have a copy on a trusted reliable stationary machine, which is considered as the home server for the given data. The home server identifies a *Security Domain* through its address. An example of security domain is the Web server (identified by its URL) hosting the information relating to a given project, which is used by the project's members to share data. Thus, peers belonging to the same security domain can trust each other and can build secure² ad-hoc groups in order to collaborate and share data. Groups are restricted to one hop ad-hoc networks, because we assume that collaborating peers are located in the local communication range of each other. Nevertheless, AdHocFS can be extended to multi-hop ad-hoc networks without modifying our coherency protocol, and this is a part of our future work. To a given data, within a given ad-hoc group, we associate at least the address of its home server and may further extend to both a local address should the data be cached on the terminal, and to terminals in the local communication range that cache the data, as identified using the underlying discovery service. The discovery service serves identifying *peer mobile terminals* that are accessible by the given mobile terminal and with which data may safely be shared. Such an identification relies on a service location protocol, i.e., SLP [2]. Seamless access is then realized by making sure that each peer within the ad-hoc group has a complete knowledge of all the locations of the data objects (we use the term *object* as an abstraction for different data types) cached/stored within the group. Upon discovery of peer terminals, naming structures (maintained by the naming service) but the objects of the terminals' local storage are merged so that any object accessible in the local communication range gets identified. Access to such an object from any of the mobile terminals then leads to copy the object locally if not already cached.

The coherency management protocol presented in this paper is a log-based protocol for mobile (one hop) ad-hoc environments. The log is presented within a data structure, named Coherency Control List (CCL), which provides the coherency manager with the history of group updates. It is managed in a way that minimizes message exchange and thus energy consumption. Our coherency protocol deals with coherency management within an ad-hoc group; data are synchronized between peers belonging to the same group. However, each data should have a copy on a stationary trusted machine. Synchronizing data with their reference copy, saved on the stationary machine, may be done either through an existing infrastructure like base station, or when any peer storing the data and belonging to the ad-hoc group later meets the given machine.

This paper is organized as follows: Section 2 provides a survey of existing systems that address replication and coherency management within a mobile environment. Sections 3 and 4 are the core of this paper, they present our contribution concerning coherency management within ad-hoc communication groups. Our contribution lies in the integration of both

²The data that are exchanged between mobiles are encrypted using symmetric encryption.

conservative and optimistic coherency management in a way that minimizes communication while not restricting the nodes' autonomy. Section 5 assesses the proposed coherency management protocol in terms of properties, complexity and energy consumption. It is shown that following our coherency protocol, ad-hoc configuration is less energy demanding than a client/server infrastructure based configuration. Finally, Section 6 concludes.

2 Related Work

Advances in wireless networking have enhanced classical fixed distributed systems to mobile wireless computing. Such an environment introduces specific constraints. Mobility requires replicating critical data on mobile devices, since disconnected machines must rely on their local resources. One fundamental problem in distributed systems is maintaining consistency. This problem is addressed either *conservatively* or *optimistically*. Conservative replication requires consistency to be maintained whenever data are updated. Various methods are used to achieve this, including primary copies, voting, weighted voting, locking, and tokens [24]. The conservative approach relies on constant connectivity, which is not always available in a mobile environment. Optimistic replication assumes that concurrent updates or conflicts are rare. It allows updates to be performed independently at any replica. Instead of preventing conflicts from occurring, mechanisms for detecting and resolving inconsistency are used upon update propagation. A number of distributed systems deals with mobility.

CODA [15, 25] is a replicated file system that supports mobile clients. Hence, it addresses replication and coherency concerns. Like AFS [20], CODA makes a distinction between file servers, which are physically secure, run trusted and are monitored by operational staff, and clients, which are physically dispersed, and may be mobile. Clients and servers communicate using a remote procedure call mechanism. The CODA replication model makes distinction between two types of replicas : server's replicas, which are primary replicas, and client's, which are secondary replicas. The CODA synchronization protocol does not support synchronization between clients. A client must synchronize with a group of available replicated servers (AVSG³). When a file is closed after modification it is transferred to all the members of the AVSG.

Ficus [13] is another replicated file system for intermittently connected machines. It differs from CODA in that it does not make any distinction between replicas and it enables synchronization between any two replicas. Ficus bases its coherency management protocol on the rarity of concurrent updates. It adopts *One Copy Availability* which permits read and write access when any data replica is accessible. The data update is applied immediately to the given replica, then the other replicas are eventually notified of the updates. Inaccessible replicas are not guaranteed to receive an update notification, and will eventually learn of the updates via later reconciliation. Ficus does not log updates. Periodically, for each replica housed by a node, the reconciliation protocol checks every other replicas to see if a new (possibly conflicting) version of the file exists. Hence, it may generate an important communication rate. CODA and Ficus are file systems, their solutions concerning replication and coherency management are strictly dedicated to data tree topology, and are not suitable for other data topologies.

Bayou [23, 7] is an optimistically replicated client-server system. It has been designed to support collaborative database applications in a mobile computing environment. A server is any machine that holds a complete copy of one or more databases. Clients are able to access data residing on any server with which they can communicate, and conversely, any machine holding a copy of a database should be willing to service read and write requests from other

³Available Volume Storage Group.

nearby machines. Bayou adopts a *read-any/write-any* replication scheme, as was first used in Grapevine [4]. That is, a user is able to read from and write to any copy of the database. Bayou cannot guarantee the timeliness with which writes will propagate to all other replicas since communication with many of these replicas may be temporarily infeasible. Thus, the replicated databases are only weakly consistent. Servers propagate writes among copies of the database using an anti-entropy protocol [6]. Anti-entropy ensures that all copies of a database are converging towards the same state and will eventually converge to identical states if there are no new updates. To achieve this, servers must not only receive all writes but must also order them consistently. Peer-to-peer anti-entropy is adopted to ensure that any two machines that are able to communicate will be able to propagate updates among themselves. OceanStore [3] applies Bayou’s conflict resolution mechanisms to a file system and extends it to work with untrusted servers that only ever see data in encrypted format. TACT [27] explores the spectrum between absolute consistency and Bayou’s weaker model.

Bengal [9] is a distributed replicated database system that supports mobile users. Bengal uses peer-to-peer optimistic-replication. Updates are reconciled between replicas when connectivity is available. Dynamic version vectors [5] are used to detect conflicts and collisions between updates made on partitioned system. When the comparison of two version vectors indicates a conflict, Bengal calls a series of configured conflict resolvers to attempt automatic conflict resolution. A conflict resolver is a software component that contains application-specific rules for determining which of the conflicting versions should be accepted or how a new version that merges the two should be created. Bengal users must design reconciliation topologies (e.g., star, ring, tree). The reconciliation topology does not constrain which replicas can reconcile with each other. Performing the reconciliations suggested by the topology merely guarantees that updates reach all replicas. The reconciliation mechanism is clearly the performance bottleneck of Bengal. It is essentially due to the cascading conflict resolvers calls that it generates.

Other systems have properties that help them tolerate high network latency, which may be very interesting within a mobile environment. The NFS4 protocol [26] reduces network round trips by batching file system operations. Echo [17] performs write-behind of metadata operations, allowing immediate completion of operations that traditionally require a network round trip. In JetFile [12], the last machine to write a file becomes the file’s server, and can transmit its contents directly to the next reader. LBFS [21] is a file system designed for low-bandwidth networks. It assumes that clients have enough cache to contain a user’s entire working set of files. This is not a reasonable assumption when the system runs on an environment containing hand held devices with reduced storage capacities.

As stated before, the first difficulty for maintaining coherency and enabling disconnected updates is data synchronization and conflict resolution. In general, conflict detection and resolution can not be achieved efficiently without an update log. Many existing systems restrict conflict resolution to a single data topology [1, 16]. The CODA system allows detecting four types of conflicts occurring in a data tree topology : (i) naming conflicts, (ii) deletion / modification conflicts, (iii) modifications conflicts, and (iv) file move conflict. However, symmetric directories move is allowed and may introduce cycles in the data tree. CODA assumes that such conflicts are rare and treats them like errors. Conflict resolution is done manually by the user. Ficus detects the first three conflicts enumerated above. It does not log updates but it uses version vectors as developed for Locus [22] to detect update conflicts. The Bayou system also detects update conflicts in an application-specific manner. Bayou write operation consists of a proposed update, a dependency set⁴, and

⁴The dependency set is a collection of application-supplied queries and their expected results. A conflict is detected if the queries, when run at a server against its current copy of a database, do not return the

a mergeproc, that is invoked when a write conflict is detected. The dependency set and mergeproc are both dictated by an application's semantics and may vary for each write operation issued by the application. The verification of the dependency check, the execution of the mergeproc, and the application of the update set is done atomically with respect to other database accesses on the server [7]. In log-based solutions, conflict resolution essentially relies on logs merging according to some constraints, such as temporal order [8, 23]. But these solutions cannot exploit the case where a reordering of operations would avoid a conflict. IceCube [14] attempts to find an ordering of operations contained in the merged log, that minimizes conflicts. However, exploring all possible ordering leads to a combinatorial explosion. Therefore, the developers should influence conflict resolution by providing pre- and post- conditions, which are exploited in order to reduce search space.

Our solution to coherency management in ad-hoc groups is different from previous work; it combines some of their advantages and introduces some new concepts. Within a security domain, We adopt optimistic coherency. Indeed, data can be updated concurrently within disjoint ad-hoc groups or isolated peers belonging to the security domain. Conflicts detection and resolution mechanisms are activated when any peer reaches its security domain home server in its communication range, or when it tries to access a given data within a group. Conservative coherency policy is enforced within an ad-hoc group, because, in our context we consider that ad-hoc group's peers are sharing the same interest and would like to work together in collaborative way. Therefore, they should have the same vision of shared information. Hence, concurrent updates of various copies of the same data within a group is not adequate in this case. However, update propagation is done only when any member of the group tries to access the given data. This enables reducing exchange of messages within the group. Indeed, if a group member updates its local copy of a data, propagation of this update is not necessary if none of the other group members wants to read or update the same data (even cached locally or not). We use a log, named Coherency Control List, for coherency management. It serves detecting and resolving conflicts. In addition to the updates history, the CCL includes the identity of peers that participate in the updates. While most mobile environment relies only on optimistic coherency management, coupling both optimistic and conservative techniques allows us to account for the various connectivity as enabled by today's wireless new networking capabilities. The overall solution is described in the next sections.

3 Ad-hoc Group Coherency Management

Our approach to coherency management within an ad-hoc group is built as follows: The reference copy for an object is the one stored on its home server, and hence the version of any object is identified with respect to a time-stamp that is incremented each time the reference copy is updated. Update of the reference copy occurs when a mobile terminal synchronizes with the server, which is further detailed in Section 4. Coherency within any ad hoc group is managed according to the exclusive writer protocol (see §3.1). Successive write operations, within an ad-hoc group, lead to propagate updates only from writer to writer in order to decrease communication (see §3.2). Diverging copies due to non-synchronized concurrent updates (i.e., within distinct groups⁵) are detected and reconciliated, within a given group, using a conflict detection protocol, and involving only peers that locally cache the corresponding data (see §3.2).

expected results.

⁵An isolated node is considered as a singleton group.

3.1 Exclusive Writer Protocol

Our coherency management protocol is based on an *exclusive writer* protocol within an ad-hoc group. Using read/write locking within a group, all write operations are exclusive within the group while read operations are shared. However, local data can be manipulated (read/write) independently within disjoint groups, provided that data are synchronized when integrating a group. Conservative coherency within a group is necessary because we consider that peers belonging to the same group are working in collaborative manner, and thus they must access the same version of the shared data. Nevertheless, every peer belonging to the security domain can constitute a singleton group and work independently from others. Thus, it can profit from optimistic coherency.

A peer can be in three modes regarding a given data object: **Read**, the peer can read the data; **ReadWrite**, the peer can read and write the data; and **Invalid**, the peer has no access to the data. With this protocol, either a peer modifies data in the **ReadWrite** state and all the other peers are in the **Invalid** state; or all the peers are in the **Read** state. This protocol ensures that all reads access the most recent data written within the group. This guarantees data coherency within a group, given that data are reconciliated upon their first access.

3.2 Conflict Detection Protocol

Concurrent and divergent updates are detected using our conflict detection protocol that uses the *CCL* (Coherency Control List), which is attached to each object. The CCL logs *all* the updates *since* the object was copied/synchronized from/with its reference copy, by storing in a data structure how and when the copy was modified since it was copied from the home server. If an object is created locally on the peer, the corresponding timestamp has a special value (*NIL*) to indicate that this object has not a reference copy yet.

An example of CCL is given in Figure 1. The CCL comprises the timestamp value of the reference copy at the time it was copied/synchronized with, and the log of subsequent updates on the local copy (Fig. 1-[a]). The log is the list of all successive data's members, as seen by this specific object copy (Fig. 1-[b]). Data's members give the set of peers belonging to the same ad-hoc group and locally caching the data. Every time the members composition is modified, through addition or deletion of a peer, and upon update, a new item is added to the list, which contains the new set of group members storing the data (Fig. 1-[c]). If a disconnected peer updates its local copy of the data, a new item is added to its local list, which contains only its ID. For every group composition, the list of updates is maintained (Fig. 1-[d]). The content of this list depends on the specific data objects. For instance, in a file system, the list of modified blocks.

Reconciliation of object copies on mobile nodes is achieved by comparing the CCLs associated with the two objects. CCL ccl_1 is a prefix of CCL ccl_2 if and only if: (i) their time-stamp value of the reference copy are equal, (ii) the list of groups in ccl_1 is a prefix of the list of groups in ccl_2 , and (iii) for each group composition given by the list of ccl_1 , the list of updates is the same as the one associated with the corresponding item of ccl_2 . In addition, for the case where ccl_1 and ccl_2 have the same list of group compositions, the above condition (iii) should hold for all the group compositions but the last one, while the list of modified blocks associated with the last group composition in ccl_1 should be a prefix of the one of ccl_2 . Basically, when ccl_1 is a prefix of ccl_2 , this means that the object copy associated with ccl_1 is an earlier version than the copy associated with ccl_2 . The older copy can then be updated, as identified using ccl_2 , leading to request for the corresponding updates to the owner of the latest object version. If two CCLs are not prefix of each other, then the corresponding object copies have diverged, as further addressed in Section 4.

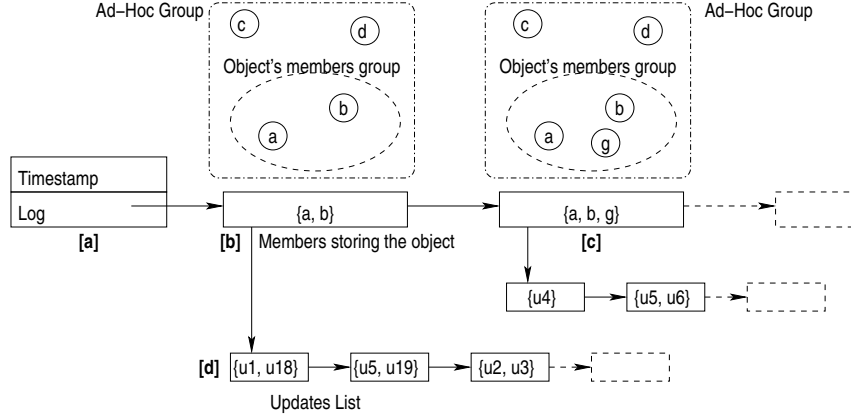


Figure 1: Coherency Control List attached to each object

3.3 Coherency Protocol

Given local CCLs, a mobile node is able to determine whether its local object copies are coherent with the objects copies stored on peer nodes of the embedding ad-hoc group. When a node n joins the ad-hoc group, its naming structure and the group's one are merged. Thus, the group's naming structure is updated by adding the names of the objects, brought by the newcomer. This enables making new objects accessible to all trusted nodes within the group. Therefore, all peers have the same naming structure. When the node n accesses any object for the first time, a coherency check is made. This enables detecting diverging copies as discussed in the previous subsection.

We use the protocol given in Figure 2 to maintain coherency within a group once its members are reconciliated. When reading an object o , a group read lock⁶ is taken (Fig. 2-[a]) and the identity of the latest writer in the group is stored. A copy of the object is first requested if the object is not stored locally (Fig. 2-[b]). If the local object was previously in either of the ReadWrite or Read mode, or has just been copied, then the local copy is up-to-date and the read access may proceed (Fig. 2-[c]). On the other hand, if the local object was previously in the Invalid mode, then the local object copy is synchronized with the one of the writer according to CCL-based reconciliation (Fig. 2-[d]). Thus, when the latest writer receives the synchronize message, it compares its object copy with the one of i using their respective CCLs (i 's CCL is sent within the synchronize message)(Fig. 2-[i]). It may then reply by: (i) *ok* if the copies are identical, (ii) *diverge* if the copies diverge, or (iii) the list of updates missing on i . Finally, the new state is stored and the local read access is done. The protocol for handling write access is quite similar. First, a group write lock is taken to ensure an exclusive write access in the current group (Fig. 2-[e]), and a copy of the object is requested to the latest writer if not cached locally (Fig. 2-[f]). Then, if the object is in the Invalid state, it is possibly outdated and it is checked for coherency and possibly updated, as done in the case of read access (Fig. 2-[g]). Once the object copy is freshed, the new state is stored (i.e. ReadWrite), the object is accessed and the set of updates is added into the object's CCL (Fig. 2-[h]). The group write lock is finally released to allow subsequent write accesses. Updates will further be propagated to the nodes storing an object copy, as read access to the objects is requested. Lazy update propagation follows from our concern

⁶Locking management is realized so that a lock held by a terminal that left the group will eventually be acquired.

```

let ObjectRead(o) =
[a] let latest_writer = GroupReadLock(o)
[b] if not cached(o) then
    i → latest_writer: GetCopy(o.name)
    o.state ← Fresh endif
match o.state with
[c] Read ∨ ReadWrite ∨ Fresh ⇒ /**/
[d] Invalid ⇒
    i → latest_writer:
    Synchronize(i, o.name, o.ccl)
match WaitAnswer with
    ok[] ⇒ /**/
    diverge[] ⇒ raise Diverge
    update[] ⇒ update(o)
done
done
o.state ← Read
let data = LocalRead(o)
GroupReadUnlock(o)
return data

let ObjectWrite(o, data) =
[e] let latest_writer = GroupWriteLock(o)
[f] if not cached(o) then
    i → latest_writer: GetCopy(o.name)
    o.state ← Fresh endif
match o.state with
    Read ∨ ReadWrite ∨ Fresh ⇒ /**/
[g] Invalid ⇒
    i → latest_writer:
    Synchronize(i, o.name, o.ccl)
match WaitAnswer with
    ok[] ⇒ /**/
    diverge[] ⇒ raise Diverge
    update[] ⇒ update(o)
done
o.state ← ReadWrite
let updates = LocalWrite(o, data)
[h] o.ccl ← UpdateCcl(o.ccl, updates)
GroupWriteUnlock(o)

let Synchronize(n, name, ccl) =
let o = Seek(name)
let status = compare(o.ccl, ccl)
[i] match status with
    OK ⇒ i → n: ok[]
    Diverge ⇒ i → n: diverge[]
    Prefix ⇒
        i → n: update[missed(o.ccl, ccl)]
done

```

Figure 2: Protocol for coherent object access

of minimizing energy consumption and hence computation and communication. However, users may will to have fresh copies upon update so as to be able to later work on the latest known version of the object. This may be easily introduced by associating priority-like attributes with objects.

4 Synchronizing Diverging Copies

Data within an ad-hoc group are updated independently from the reference copies stored on the home server as well as from other copies within distinct ad-hoc groups. The update of the reference copy occurs each time a mobile terminal synchronizes its local copy with the server. The frequency of this synchronization is determined by the user. It can be done either randomly when any mobile terminal, belonging to the group and storing the data, reaches the server, or following any fixed period. In the later case, synchronization should be done through neighbor base stations if the group location is geographically far from the home server. In this case, the local infrastructure should support and enable deployment of

the network protocol that enables such access⁷. We can imagine that in few years it will become common place to find such facilities in any building or in public transport (planes, trains, ...).

4.1 Ad-hoc Group Versus Base Station Infrastructure

A question that can be asked is why still using ad-hoc group communication and group data synchronization, when it is possible to synchronize data with the server, through base station if necessary, after each modification. Our group management protocol (detailed in [19]) allows many groups to manipulate the same data in an independent manner. Synchronizing data with its reference copy on the server upon each modification will generate an important number of divergent copies that need to be reconciliated. It is obvious that this significant number of reconciliation will represent a loss in time and energy for all peers involved in this operation, while, it is not necessary to reconcile all versions of the data. In addition, it is well known that base stations constitute bottlenecks. Further, maintaining data coherency within an ad-hoc group makes this one independent of the infrastructure presence, failure or latency.

However, ad-hoc networking is more expensive than infrastructure-based one. Indeed, when using, e.g., a 2 Mbps wireless interface being idle in ad-hoc configuration costs 843 mW, and only 66 mW in base station configuration. Therefore, being idle in ad-hoc configuration is 13 times more energy consuming than in base station configuration. But, sending or receiving a message have the same cost in term of energy consumption for a mobile device in both cases. In general, ad-hoc and infrastructure-based networking should be seen as complementary. Ad-hoc networking is more convenient for collaborative work, while the infrastructure-based one is less energy consuming when idle. We are working on making it possible to switch from one network to another in a transparent manner, and hence adapt the networking to the group's specific situation (i.e., available energy, network connectivity, surrounding environment).

4.2 Synchronizing with the Reference Copy

Synchronization with the server is managed as follows: a list of CCLs is attached to the reference copy on the server. Each element of this list contains a time stamp and the corresponding CCL. A new element is added to this list each time the time-stamp is incremented. The time-stamp is incremented each time the reference copy is updated, which occurs when a peer storing a newer version of the data synchronizes with the server. Synchronizing peer's copy with the reference one may generate some conflicts due to diverging updates.

We use the protocol given in Figure 3 to ensure synchronization between data stored on a peer and their reference copy on the home server. When reaching the home server, a peer i sends to it the CCL of the modified data and its name (Fig. 3-[a]). Then the server, first, searches for the data named $name_i$. If this data does not exist on the server, two cases are possible. The data was erased from the server, but the peer still stores it, and it is up to the user to decide either to save this data on the server or delete it from the peer (Fig. 3-[g]). The second case is when the data was locally created on the peer, and it first synchronizes it with the server (Fig. 3-[f]). In this case, the server creates a new CCL with data name equal to $name_i$, timestamp equal to 1 and log equal to log_i . Then, the server saves a copy of this new data and sends to $peer_i$ the current time-stamp within a Synchronize message. If the data $name_i$ was already stored on the server (Fig. 3-[h]), then,

⁷IEEE 802.11 in our prototype.

```

let PeerSideSynchronization(o) =
[a]  $i \rightarrow Server:(o.ccl, o.name)$ 
[b] match WaitAnswer with
    Synchronize[  $time\_stamp_s$  ]  $\Rightarrow$ 
        erase(o.log);
         $o.time\_stamp \leftarrow time\_stamp_s$ ;
[c] Update[  $time\_stamp_s, data$  ]  $\Rightarrow$ 
        erase(o.log);
         $o.time\_stamp \leftarrow time\_stamp_s$ ;
        update(o.name, data)
done

let ServerSideSynchronization=
[d] match searchCCL( $name_i$ ) with
[e] NotFound  $\Rightarrow$ 
[f] if  $time\_stamp_i = NIL$  then
    creatCCL( $name_i$ );
     $time\_stamp_s \leftarrow 1$ ;
     $log_s \leftarrow log_i$ ;
    update( $ccl_i$ );
     $server \rightarrow peer_i$ : Synchronize[  $time\_stamp_s$  ]
[g] else “The data  $name_i$  was erased on the server”
endif
[h] Ok[ $ccl_s$ ]  $\Rightarrow$ 
[i] match compare( $ccl_i, ccl_s$ ) with
[j] Equal  $\Rightarrow server \rightarrow peer_i$ : Synchronize[  $time\_stamp_s$  ]
[k] Prefix  $\Rightarrow server \rightarrow peer_i$ : Update[ $time\_stamp_s, missed(ccl_i, ccl_s)$ ]
[l] Suffix  $\Rightarrow$ 
    increment( $time\_stamp_s$ );
     $log_s \leftarrow log_i$ ;
    update( $missed(ccl_i, ccl_s)$ );
     $server \rightarrow peer_i$ : Synchronize[  $time\_stamp_s$  ]
[m] Otherwise  $\Rightarrow$ 
[n] let  $nearest\_ccl_s = Search\_Nearest\_CCL(ccl_s, ccl_i)$ 
[o] match compare( $ccl_i, nearest\_ccl_s$ ) with
[p] Equal  $\vee$  Prefix  $\Rightarrow$  Sends to the peer  $i$  all the missing updates until the last  $CCL_s$ 
[q] Suffix  $\vee$  Otherwise  $\Rightarrow$  ConflictResolution
done
done
done

```

Figure 3: Protocol for peer synchronization with its home server

the server’s last⁸ CCL is compared to the peer’s (Fig. 3-[j]). We use the notions of prefix and suffix between CCLs. The prefix is defined as in Section 3, but the equality between the timestamps is not considered to determine whether a CCL is a prefix of another one or not. The suffix is defined as follows : c_1 is suffix of c_2 if and only if c_2 is prefix of c_1 .

If the two CCLs are equal, then the server answers by sending its time-stamps within a Synchronize message (Fig. 3-[k]). In the case where the peer’s CCL ccl_i is a prefix of the server’s CCL ccl_s , the server sends to peer i the newer version of the data and its timestamp within an Update message (Fig. 3-[l]). And if ccl_i is a suffix of server’s CCL ccl_s (or ccl_s is a prefix of ccl_i), then, the server updates its copy using updates that it misses and that are in ccl_i , increments its time-stamp, and sends it to peer i within a Synchronize message (Fig. 3-[l]). On the other hand, if the two CCLs are completely different, then, we seek an older⁹ CCL that is still being stored on the server, and which is near to ccl_i (Fig. 3-[n]). A ccl_s is considered near to the ccl_i if they are equal or the ccl_s is the oldest¹⁰ suffix of ccl_i . If such CCL exists, the sever sends to the peer i all the updates the peer misses until the last CCL, which corresponds to the current server’s time-stamp, within an update message (Fig. 3-[p]). If the CCLs still being stored on the server does not match our requirements then conflict resolution is required (Fig. 3-[q]). Notice that enhanced automation of conflict resolution could be achieved using constraints for ordering CCLs, based on the proposal of [14], which is part of our futur work.

On the peer’s side, when receiving a Synchronize message, the peer i erases its log and updates its time-stamp (Fig. 3-[b]). If it receives an Update message, then, it erases its log, updates its time-stamp, and store the new version of the data (Fig. 3-[c]).

5 Assessment

The protocol that we have presented, ensures that each peer within an ad-hoc group will access the latest group’s version of the data. It enables manipulating distinct copies of the same data within different groups, which implies reconciliation when synchronizing with the server, or when meeting other peers in a group provided that data is accessed within this group. This coherency protocol aims at reducing coherency check, and hence communication and computation. We consider a group of m peers ($0 < m \leq 10$), where n peers are storing the data object

5.1 Theoretical Complexity

The number of messages that are exchanged in our coherency protocol, within an ad-hoc group to obtain data accesses, depends on both the previous state of the object’s lock and the requested operation.

If the object was write locked by peer b and a write operation is requested by peer a , four messages are necessary to validate this operation: the new writer a asks b for the write lock. Then, b sends to a the write lock and its CCL, a checks coherency, and sends back to b the list of missing updates. Finally, b sends the updates. If a read operation is requested by a , then the number of messages reach $(2 * n - 1)$ if broadcast is used, and $(3 * n - 2)$ messages otherwise: the new reader a asks the writer b for the read lock. Then, b sends to a the read lock, and its CCL to all the peers storing the data ($(n - 1)$ messages without broadcast, and one message otherwise), the $(n - 1)$ peers send back to b the list of missing updates.

⁸The last CCL corresponds to the largest time-stamp value.

⁹Whose time-stamp is inferior to the current one.

¹⁰Meaning that its time-stamp is the smallest regarding to other CCL for the same data.

Previous Lock	Requested Operation	Ad hoc		Infrastructure	
		Msgs	mW.sec	Msgs	mW.sec
Write	Write	4	13.8	8	15.21
	Read	3*N-2	96.6	5*N-2	97.05
	Read + Broadcast	2*N-1	75.17	4*N	97.05
Read	Write	N+1	37.95	N+3	29.83
	Write + Broadcast	3	13.07	5	29.83
	Read	0	0	0	0

Table 1: Cost of coherency management

Finally, b sends the updates to the $(n - 1)$ concerned peers. If the object was read locked by peer b , and a read operation is requested then no message is exchanged. However, if a write operation is requested, then, $(n + 1)$ messages are necessary to validate this operation without using a broadcast, and 3 messages otherwise: the new writer a asks peer b for write lock. Then, b sends to a the write lock. Finally, a informs all concerned peers about the write operation.

The number of messages that are exchanged within the group will increase quite significantly if the mobiles rely on base station (client/server) for their communication. The Table 1 shows the advantage of conservative coherency within ad-hoc groups versus optimistic recovery, using infrastructure-based communication, as dealt by existing mobile systems.

5.2 Energy Consumption

We use the 2.4Ghz DSSS lucent IEEE 802.11 WaveLan PC "Bronze" 2Mbps wireless interface for our evaluation. Focusing on the energy cost associated with communication in ad-hoc networking, the cost associated with the emission of one message is the sum of: the cost of emission for the sender node, the cost of reception for the destination node, and the cost of reception for non-destination nodes (i.e., terminals that receive control messages due to their location with respect to the aforementioned nodes).

The energy consumed by any mobile terminal for sending, receiving or discarding a message is given by the linear equation : $\epsilon = m * size + p$; where $size$ is the message size, and m (resp. p) denotes the incremental (resp. fixed) energy cost associated with the message [10]. The value of ϵ in ($\mu W.sec$) is detailed hereafter, for the various mobile terminals impacted by the message transmission. In point to point transmission $\epsilon_{SENDER} = 1.9 * size + 454$, $\epsilon_{DESTINATION} = 0.5 * size + 356$, and the value of ϵ for non-destination nodes is given by $\epsilon_{NonDest} = (m - 2)[-0.22 * size + 210]$. Then, the energy consumption for a point to point message is given by the sum of previous equations: $\epsilon_{msg} = (2.84 - 0.22 * m) * size + 390 + 210 * m$. In the case of broadcast, $\epsilon_{SENDER} = 1.9 * size + 266$, and for the $(m - 1)$ destination peers, $\epsilon_{DESTINATION} = (m - 1)[0.5 * size + 56]$. Then, one broadcasted message costs $\epsilon_{msg} = (1.4 - 0.5 * m) * size + 210 + 56 * m$. As it was mentioned in Subsection 4.1, being idle in ad-hoc configuration costs 843 mW, and only 66 mW in base station configuration. However, sending or receiving a message has the same cost for a mobile device in both cases. We further ignore the energy consumed by the base station and focus on the energy spent by our group of mobiles to communicate with each other via the base station. Thus, a peer a spends ($\epsilon_{SENDER} = 1.9 * size + 454$) $\mu W.sec$ to send a message to b via the base station,

on the other side, the peer b spends ($\varepsilon_{DESTINATION} = 0.5 * size + 356$) $\mu W.sec$ to receive this message from the base station.

Using IEEE 802.11, the maximum size of a message is 1500 byte. The energy cost of our protocol for a group of 10 peers¹¹, all of them storing the data ($m = n$) is depicted in Table 1 for both peer-to-peer ad-hoc communication and base station communication.

Relying on a base station is more energy consuming to obtain a write or a read access to a given data, except when the data object was read locked and a write operation is requested. Using broadcast in base station configuration, following our coherency protocol, does not make any difference regarding energy consumption, because the broadcast is performed by the base station and we do not take into account the energy consumption of the base station. However, using broadcast in peer-to-peer ad-hoc configuration, following our coherency protocol, reduces by about 22.18% when the data was write locked and a read is requested, and about 65.56% when the data was read locked and a write operation is requested.

6 Conclusions

In this paper, we have presented and evaluated a coherency management protocol in ad-hoc group communication. Our aims were to enable: (i) seamlessly access data anywhere, anytime, in a trusted and cheap way, (ii) share information with trusted computing resources that are in the terminal's communication range, and (iii) have a coherent view of data, independent of the terminal that is being used.

The goals above abstract two ideas, where coherency management is of the same importance: a collaborative group, where users share their data among their respective devices, and, a person using a number of computing devices.

Our mobile-aware coherency management protocol allows peers caching a data copy to reconcile either with their home server or with other peers when joining any group, provided that this data is cached within the group. It provides, also, a conflict detection based on the Coherency Control List attached to each data copy. Among enhancements of our system that we are further improving support for automatic conflict resolution.

References

- [1] S. Balasubramaniam and B.C. Pierce. What is a file synchronizer ? *Int. Conf. on Mobile Comp. and Netw. (MobiCom'98)*. ACM/IEEE, 1998.
- [2] C. Bettstetter and C. Renner. A comparison of service discovery protocols and implementation of the service location protocol. In *Proceedings 6th EUNICE Open European Summer School: Innovative Internet Applications*, 2000.
- [3] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. Oceanstore: An extremely wide-area storage system, November 2000.
- [4] A. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. In *Communication of the ACM 25(4)*, pages 260–274, April 1982.
- [5] Ratner David, Reiher Peter, and Popek Gerald J. Dynamic version vector maintenance. Technical Report 970022, 30, 1997.
- [6] A. J. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated data base maintenance. In *Proceedings of the*

¹¹It is the maximum size supported by the actual technology.

- 6th Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, B.C., Canada, August 1987.
- [7] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.
 - [8] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Designing and implementing asynchronous applications with bayou. In *Proc. of the Symp. on User Int. Software and Tech.*, Banff Alberta (Canada), Oct. 1997.
 - [9] Todd Ekenstam, Charles Matheny, Peter L. Reiher, and Gerald J. Popek. The bengal database replication system. *Distributed and Parallel Databases*, 9(3):187–210, 2001.
 - [10] L. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *proc. of the IEEE Infocom*, 5(8), 2001.
 - [11] P. Gauthier, D. Harada, and M. Stemm. Reducing Power Consumption for the Next Generation of PDAs: It's in the Network Interface ! January 1996.
 - [12] Bjorn Gronvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Proc. of Operating Systems Design and Implementation*, pages 251–264, 1999.
 - [13] Richard G. Guy. Ficus: A very large scale reliable distributed file system. Technical Report CSD-910018, Los Angeles, CA (USA), 1991.
 - [14] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. *Proc. of the 20th ACM Symposium on Principles of Distributed Computing (PODC 2001)*, 2001.
 - [15] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the coda file system. In *Proc. of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 202–213, 1993.
 - [16] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. of the Usenix Winter Conference*, January 1995.
 - [17] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, 1994.
 - [18] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 4th edition edition, 1996. ISBN: 0-8493-8523-7.
 - [19] Davide Mentre, Malika Boulkenafed, and Valérie Issarny. ADHOCFS: A serverless file system for mobile users. Technical report, INRIA-Rocquencourt, November 2001.
 - [20] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew : a distributed personal computing environment. *Communication of the ACM*, 29(3):184–2001, march 1986.
 - [21] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.
 - [22] D.S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transaction on Software Engineering*, pages 240–246, May 1983.
 - [23] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. Symp. on Operating Systems Principles (SOSP-16)*, pages 288–301, Saint Malo, Oct. 1997.
 - [24] David Howard Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, University of California, UC Los Angeles, January 1998.

- [25] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
- [26] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Nfs version 4 protocol. RFC 3010, Network Working Group, December 2000.
- [27] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. of the 4rd Symposium on Operating Systems Design and Implementation*, 2000.