Log Garbage Collector-based Real Time Collaborative Editor for Mobile Devices

Moulay Driss Mechaoui[‡], Asma Cherif[*†], Abdessamad Imine[*§] and Fatima Bendella[¶]
[‡]Mostaghanem University, Algeria
Email: moulaydrissnet@yahoo.fr
[*]Inria Lorraine and Nancy 2 University, France
[†]Email: cheriasm@loria.fr
[§]Email: imine@loria.fr
[¶]USTO, University of Oran, Algeria
Email: bendella_fatima@yahoo.fr

***Abstract*—The mobile phone technologies are becoming pervasive in recent years. These items such as IPhones, IPad and Androïds are very attractive since they provide relatively good resources for a mobile device. Several works aim at integrating desktop applications in these tools to make them closer to the real computer. However, adapting desktop applications to these tools is a challenging problem as they do not have the same features. Real time collaborative editors are famous applications allowing for several users to edit the same shared document simultaneously. Such an application is more and more used not only in professional fields but also in a personal context. In this work, we extend decentralized collaborative editors to mobile devices by conceiving a successful garbage collection scheme that optimally manages mobile devices resources. We propose a novel design for distributed garbage collection that ensures a good behavior of the application through the good measurements obtained for different types of mobile phones.**

Keywords: Real-time collaboration, Distributed garbage collection, Mobile device based applications.

I. Introduction

Motivations. Mobile Devices such as PDAs and cell phones are becoming more and more pervasive. Several works try to integrate desktop applications on these devices. Among these applications, we are mostly interested on collaborative editors (*e.g.* Google Docs, Abiword) which provide computer support for modifying simultaneously shared documents, such as articles, wiki pages and programming source code, by dispersed users. For instance, researchers may use a collaborative editor to edit the same article simultaneously without the need of being members of the same organization. In this work, we investigate on Real Time Collaborative Editors (RCE) on mobile devices. It should be noted that, RCE have specific requirements that may not match with those characterizing mobile devices, namely:

- *High local responsiveness*: the system has to be as responsive as its single-user editors [1], [2];
- *High concurrency*: the users must be able to concurrently and freely modify any part of the shared document at any time [1], [2];
- *Consistency*: as the shared objects are replicated, the users must eventually be able to see a converged view of all copies [1], [2];
- *Decentralized coordination*: all concurrent updates must be synchronized in decentralized fashion in order to avoid a single point of failure;
- *Scalability*: a group must be dynamic in the sense that users may join or leave the group at any time.

These characteristics pose a significant challenge on the design of systems that support collaboration using mobile devices. We stress that even though mobile phones offer ad hoc communications and are suitable to host distributed applications, they are battery-powered so less powerful and have limited storage capacities compared to those offered by computers.

Moreover, collaborative editors are based on log usage, as it is necessary to store the track of the operations received to ensure the convergence of the shared data. The motivation for maintaining a log at each site is that a remote operation must integrate the effect of all concurrent operations to be executed on the receiver site. However, not all operations received by a site must be kept in its log. In particular, an operation can be safely deleted from a site's log if: (i) it is already received in all other sites and (ii) all operations that depend on it are received by all sites.

Deploying RCE based on logs on mobile devices can be costly. Indeed, they require lot of memory to manage the increasing log size. Consequently, a mechanism of garbage collection must be set up to allow for log reset at a given time and then to start again the collaboration with empty logs what improves the performances and response time of the collaborative application.

The motivation for garbage collection is twofold. Firstly, storage capacity is not infinite as in practice memory always has limited size. Secondly, with the continuous increase of log size, the performances of the system degrade. Hence we need to devise a garbage collection technique allowing to delete all operations already seen and executed at collaborating sites at a given time knowing that logs are not identical in different sites as we allow out-of-order execution of operations. Moreover, group size in collaboration model is dynamic (churn) which complicates the garbage task.

COLLABORATECOM 2010, October 9-12, Chicago, USA

DOI 10.4108/icst.collaboratecom.2010.28

Related work. Applying garbage collection on a distributed collaborative editor assumes that we have a global view of the distributed system state. Chandy and Lamport [10] propose an algorithm to determine global states of distributed systems by recording a logical (or causal) snapshot of the system. This algorithm is based on the hypotheses below: (i) no failures which means that all messages arrive intact, (ii) the communication channels are unidirectional and FIFO-ordered and (iii) there is a communication path between every process pair. This FIFO communication channels ensures that reception order of messages is the same as the emission order which is not the case for RCEs.

The solution proposed in [9], is a protocol that allows sites in a replicated data base system to discard old updates for maintaining mutual consistency. This protocol use [10] to determine the global state of the distributed system and timestamps to ensure a total order of update. This solution could not meet RCE requirements since it is hard to ensure a total order in decentralized fashion with dynamic groups.

One can consider the garbage collection as a consensus problem in distributed systems, in our solution the decided value of consensus is the Log. It is impossible to solve consensus problem in asynchronous systems because we cannot determine with certainty whether a process has crashed or not (it may be slow, or its messages are delayed), the FLP model [11] proposed by Fischer, Lynch and Paterson proved that even if at most one process may crash, and all links are reliable, it is impossible to achieve consensus. To solve this problem, other works propose to enrich the asynchronous system with a failure detector. For instance, [12] propose a simple $\Diamond$S-based consensus protocol, this protocol uses $\Diamond$S as failure detector [13] and requires a majority of correct process. It was shown that this protocol never block and terminates. It helps to achieve consensus on a common value which is decided by the majority of correct processes. The specification behind this protocol is simple and allow to decide on the same final decision. However, our model is interactive and more complicated since we aim to decide on a final value that is not necessarily the value proposed by any of the collaborating users but may be a new value aggregated from the proposals of other users and the system evolution.

In database area, the work of [7] is of relevance. It proposes a pruning technique allowing for the deletion of different updates that are no longer needed by the system. However, the use of timestamps make it inappropriate in a dynamic context. Other works such as [8] have been proposed to focus on garbage collection in object oriented databases but interest rather in memory management to delete unused objects and not in cleaning logs. In our knowledge, the only previous work on distributed collaborative editors that proposed a garbage collection technique to reduce log size is presented in [2]. Unfortunately, this solution is based on the use of state vectors thus only meets collaborations with a fixed group size thus being inappropriate to RCE requirements.

Contributions. In this paper, we have based on previous works on collaborative editors to extend the collaboration model to mobile devices. We design a new approach to ensure garbage collection in a decentralized fashion in order to alleviate the storage capacity needed by collaborative editors models. We also demonstrate a good behavior of our garbage collection scheme by the experimental study that we will discuss later.

Outline. This paper is organized as follows: In Section 2, we address garbage collection issues in RCE. In Section 3, we give an overview on the coordination model that we use. Section 4 presents the concept of our garbage collection. Section 5 illustrates performance study and section 6 summarizes contributions and discusses future work.

II. Coordination Model

Real-Time Collaborative Editors (RCE) allows many users (or sites) to concurrently update the shared data and next to synchronize their divergent replicas in order to obtain the same data. The updates of each site are executed on the local replica immediately without being blocked or delayed, and then are propagated to other sites to be executed again. As a long established convention in RCE [1], [2], the shared object is a finite sequence of elements from any data type. For instance, an element may be regarded as a character, a paragraph, a page, a slide, an XML node, etc. It is assumed that the shared object can only be modified by the following primitive operations: (i) $Ins(p, e)$ inserts the element e at position p; (ii) $Del(p)$ deletes the element at position p. We shall refer to the state of a document by St and to operations that alter document state by *cooperative operations.* A crucial issue when designing shared objects with a replicated architecture and arbitrary messages communication between sites is the *consistency maintenance* (or *convergence*) of all replicas. To illustrate this problem, consider the group text editor scenario shown in Figure 1. There are two users (on two sites) working on a shared document represented by a sequence of characters. Initially, both copies hold the string " *efecte*". Site 1 executes operation $o_1 = Ins(1,f)$ to insert the character *f* at position 1. Concurrently, site 2 performs $o_2 = Del(5)$ to delete the character *e* at position 5. When o_1 is received and executed on site 2, it produces the expected string "*effect*". But, when o_2 is received on site 1, it does not take into account that o_1 has been executed before it and it produces the string "*effece*". The result at site 1 is different from the result of site 2 and it apparently violates the intention of o_2 since the last character *e*, which was intended to be deleted, is still present in the final string. Consequently, we obtain a *divergence* between sites 1 and 2. It should be pointed out that even if a serialization protocol [1] was used to require that all sites execute o_1 and o_2 in the same order (*i.e.* a global order on concurrent operations) to obtain an identical result *effece*, this identical result is still inconsistent with the original intention of o_2.

To maintain consistency of the shared document, we use the Operational Transformation (OT) approach which has been proposed in [1]. In general, it consists of application-dependent transformation algorithm, called IT, such that for

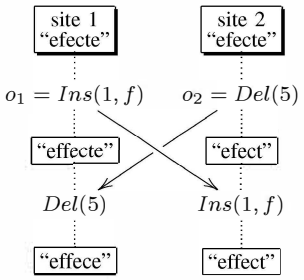


Fig. 1. Incorrect integration.

every possible pair of concurrent operations, the application programmer has to specify how to integrate these operations regardless of reception order. In Figure 2, we illustrate the effect of IT on the previous example. At site 1, o_2 needs to be transformed in order to include the effects of o_1: $o_2' = IT((Del(6, e), Ins(2, f)) = Del(7, e)$. The deletion position of o_2 is incremented because o_1 has inserted a character at position 1, which is before the character deleted by o_2.

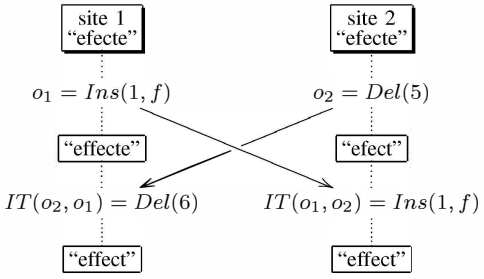


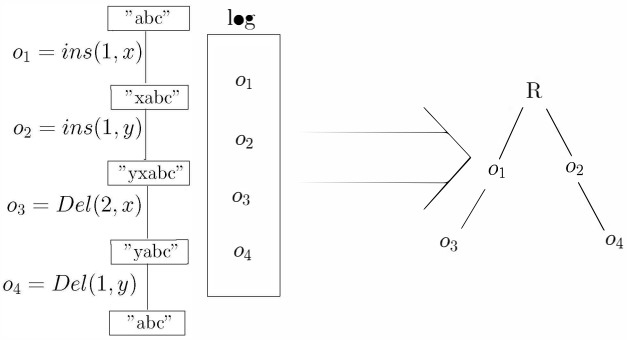
Fig. 2. Integration with transformation.

It should be noted that OT enables us to ensure the consistency for *any number* of concurrent operations which can be executed in *arbitrary order* [3], [4] (*i.e.* no global order is necessary).

For managing collaborative editing work in a decentralized and scalable fashion, we reuse an OT-based framework that owns the following features [5]: (i) It supports an unconstrained collaborative editing work (without the necessity of central coordination). Using optimistic replication scheme, it provides simultaneous access to shared documents. (ii) Instead of vector timestamps [1], it uses a simple technique to preserve causality relation based on a *dependency tree* where each operation has only to store the operation identity whose it directly depends on (see Figure 3). This tree-based causality relation is independent on the number of users and it provides high concurrency in comparison with vector timestamps. (iii) Using OT approach, reconciliation of divergent copies is done automatically in decentralized fashion. (iv) This framework can scale naturally thanks to its minimal causality dependency relation. In other words, it may be deployed easily in Peer-to-Peer (P2P) networks.

To be more general, we define the dependency relation as follows:

Definition 2.1: **(Dependency Relation)** A cooperative operation o_2 depends on another cooperative operation o_1, iff



Deleting an element depends on the operation that has inseted this element. So, o_3 depends semantically on o_1 and o_4 depends ono_2. There are other dependecy relations that we will not discuss here due to space lack. For more details the reader can refer to [5].
Suppose that o_1 and o_2 depend on a root R. Then the log is seen as the set of leaves $\{o_3, o_4\}$.

Fig. 3. Dependency tree built from log.

(i) o_2 has seen the effect of o_1 and, (ii) o_1 and o_2 alter the same element or (iii) o_1 and o_2 alter different elements but the execution of o_1 and o_2 in different orders on the same state results on two different states.

In this OT-based framework, every site generates operations sequentially and stores these operations in a data-structure called a *log*. Two important steps are performed as follows:

- Generation of local operation: When an operation o is locally generated, it is immediately executed on its generation state and next it is stored in the local log. Once executed, its dependency is computed in order to determine its direct predecessor. After the determination of the dependency, this operation is propagated to all sites in order to be executed on other copies of the shared document.
- Integration of remote operation: Each site uses a queue to store the remote operations coming from other sites. To preserve the causality dependency, a remote operation o is extracted from the queue when it is *causally-ready* ($i.e$ if its dependency has been already integrated on receiver site). Next, we compute the transformed form o' to be executed on current state using IT function. Finally o', is executed on the current state and stored in the local log.

Moreover, this framework enables the dynamic groups in the sense that users may quit or enter the groups at any time. When joining the group, a new user requests the current document state and the current log from the nearest user in order to start the collaboration with the members of this group.

A stable state in a RCE is achieved when all generated operations have been performed at all sites. So, the following criteria should be ensured [5]:

Definition 2.2: **(Consistency Criteria)** A RCE is *consistent*

iff it satisfies the following properties:

1) *Dependency preservation:* if o_1 depends on o_2 then o_1 is executed before o_2 at all sites.
2) *Convergence:* when all sites have performed the same set of operations, the copies of the shared document are identical.

At stable state, site logs are not necessarily identical because the concurrent operations may be executed in different orders. Nevertheless, these logs must be equivalent in the sense that they must lead to the same final state. For instance, in the scenario presented in Figure 2 the two logs $[Ins(1, f); Del(6)]$ and $[Del(5); Ins(1, f)]$ are not identical but lead to the same final state thus they are equivalent.

Definition 2.3: **(Equivalent Logs)** Two logs are equivalent iff they produce the same state when applied to a given state St.

Our objective here is to develop on the top of this framework a garbage collection layer for managing the memory resources allowed to the log sites while preserving the consistency criteria of RCE (see Definition 2.2). Of course, this garbage collection-based RCE will be well suited for mobile devices.

III. Garbage Collection Issues

We have already shown the importance of garbage collection in RCEs dedicated to mobile devices. Unfortunately, the garbage collection is really a hard task under the requirements discussed before as it must address the following issues:

First Issue. Since logs are equivalent and not identical from one site to another due to out of order execution of operations, the log removal operation may be executed on different contexts from one site to another. In such situation, collaborative editing after the garbage collection procedure inevitably leads to divergence cases as it is shown in the scenario of Figure 4. In this scenario, we consider two users that begin the collaboration with the same initial state "eact". Site 1 generates $o_1 = ins(1, r)$ to insert the character r at position 1 and gets the state "react". The second site removes the character "e" at position 1 with the operation $o_2 = del(1)$ and then gets the state "act". Concurrently, the site 1 initiates garbage collection, cleans his log and propagates the removal order to the site 2. The latter receives the garbage collection order before the operation o_1. Hence, when he receives $o_1 = ins(1, r)$, he executes it in its received form since IT function has no impact on o_1 (the log is empty). Consequently, the final state at site 2 is "ract". Now, site 1 receives $o_2 = del(1)$ and executes it also in its reception form as his log is empty and then derives the state "eact" which is different from the state of site 2.

It should be pointed out that an operation may also loose its dependency and remains eternally unready if logs are removed in arbitrary fashion.

Accordingly, cleaning process requires a global view of the collaboration state in order to ensure that log removal will be executed at the same context in all collaborating sites. In other words, we must be able to deduce whether an operation was received by all users or not, and whether it is needed

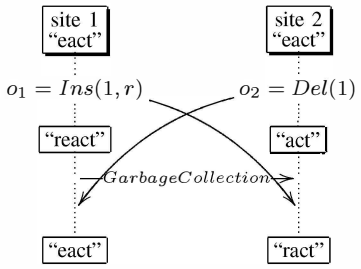


Fig. 4. Divergence caused by garbage collection applied on different contexts

by an other operation for an integration process. If it is the case, then the operation could be safely removed from the log. Otherwise, replicas will diverge.

To overcome this problem, it is necessary to draw a global view on the state of each user in order to decide about the portion of log that could be deleted without leading to divergence. This is possible to achieve through request-response messages before log removal. The question that arises here: is global view possible in a dynamic and distributed collaboration?

Second Issue. Suppose that the garbage collection only begins when all users are ready to clean their logs. To know if a user is able to remove his log or not, we must wait for his answer. However in a peer-to-peer context, a user may leave the group at any time and thus never responds to the garbage request. A user can also crash down and be prevented to respond. In both cases, it is difficult to take the correct garbage decision while preserving consistency criteria (see Definition 2.2). Because of message asynchrony, a user has no safe means to know whether another user has or has not crash. Even the timeout approach has many inconveniences. For instance, a user may respond just after the timeout expiration. Moreover, even the timeout itself is difficult to define.

The solution proposed in [2] relies on state vectors to draw a state view for each user. Every silent user has to send periodically the value of its state vector to other users. State vectors of active sites are deduced from the operations they perform. These vectors are used to calculate a minimal state vector allowing for garbage decisions. However, this approach is limited to static groups and could not be used in the case of dynamic groups because of failure impact on the garbage procedure $i.e.$ when a site is absent or silent, its state vector remains unchanged, then other users could no more delete operations from their logs.

Third Issue. Another issue that could be faced when trying to garbage logs in RCE is when the garbage process is disturbed by the join of a new user. It is known that in peer-to-peer context, a new user can join the group at any time. When a new peer tries to join the group while its members are processing a garbage collection procedure, we may diverge if that user receives the current log before its deletion by all the members of the group. Hence, the new user is able to generate new operations based on a context completely different from other

user contexts. Consequently, we inevitably diverge according to the scenario presented in Figure 4.

To illustrate this, let us consider the scenario in Figure 5 in which we have 3 sites collaborating together in order to edit the same document. The initial group is only composed of users s_1 and s_2. Then site s_3 decides to join the collaboration. Site s_3 requests the log and the state from site s_2. Concurrently, the site s_1 initiates a garbage collection and propagates the request to site s_2. The latter sends the log and the state to s_3 before receiving the request of the former. Then he decides to empty his log as well as site s_1 does. When s_3 is ready to join the group, he has a non empty log while others have empty ones. It is obvious that the context of s_3 is different from that of s_1 and s_2 which leads to divergence.

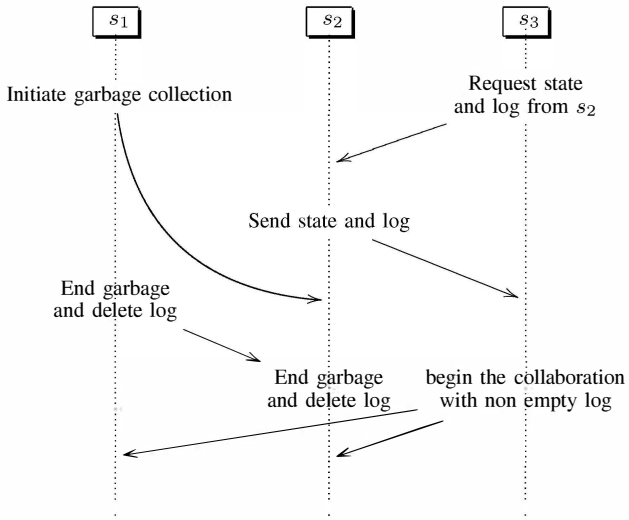


Fig. 5. Divergence caused by a new user joining the group.

IV. Our Garbage Collection Scheme

A. Garbage Collection Algorithm

In this section, we will discuss the concept of garbage and devise solutions for garbage collection. Each site maintains its own log which can be seen as a dependency tree using the semantical dependency relations (see Figure 3). The leaves of this tree represent a summary about what a site has received since an operation is causally ready only when its dependency is already executed. Let $\mathcal{L}$ be the set of leaves maintained by each site s. For instance, in the example of Figure 3, $\mathcal{L} = \{o_3, o_4\}$. Consequently, two sites having the same set of leaves means that they have executed the same set of operations and hence they have equivalent logs. This set of leaves is updated each time a local operation is generated or a distant one is received by replacing old leaves by new ones. The root of the dependency tree noted R represents the identity of the site that performed the last the garbage collection procedure. The root has no effect on the state but rather serves as a break point indicating a garbage collection initiation. We assume that initially every site begins with an empty root.

It is obvious that the tree structure helps to draw the set of leaves and thus facilitates the comparison between different user logs. Consequently we reduce the size of parsed operations to decide whether the garbage procedure is allowed or not since we only compare the set of leaves rather than the entire logs. The tree structure allowing for leaves comparison is used in order to check whether all users have the same context or not thus we overcome the **First Issue** presented in Section III.

To proceed garbage collection, the collaborating sites exchange some garbage messages to decide whether it is possible to clean logs or not. In fact, it is not always possible to delete operations from the log. For instance, two sites having different sets of leaves are unable to clean their logs since we can deduce that there are operations in network not yet received by all sites. To do so, collaborating sites have to exchange the following messages in order to decide about the garbage:

- Garbage collection initiation (GCI): is a message sent by the site initiating the garbage collection procedure; the GCI contains the site identity as well as the set of leaves $\mathcal{L}$.
- Acquirement (ACK): when a site receives the GCI message, he computes the difference between his set of leaves $\mathcal{L}$ and that received in the GCI, then sends the result to the initiator.
- Garbage collection order (GCO): this message contains the root of the new empty tree that will replace the old one. Thus, when it is received from the initiator, it leads to the removal of the log.

A user can have different three states according to the collaboration state: *blocked*, *active* and *passive*. He is blocked when a garbage collection procedure is processing and as soon as the garbage ends he turns to the active state. Otherwise, the user is passive (this is the state when a user joins the group).

In Algorithm 1, we give all the steps of the control concurrency algorithm taking into account the garbage collection messages.

When a user generates a local operation, he invokes the function that will integrate remote operations (INTEGRATE_LOCAL_OPERATION) discussed in section II. When a user decides to initiate a garbage collection, he processes the LUNCH_GC procedure. Now, when he receives a garbage collection message, he calls the RECEIVE_GCI_MESSAGE (m) procedure detailed in Algorithm 2 where we can see that according to the type of the received message (GCI, ACK or GCO) the user will apply the corresponding processing. If the message received is a GCI, the user invokes the RECEIVE_GCI_MESSAGE (m) procedure. To summarize, the garbage collection scheme proceeds in five steps:

1) The garbage collection initiating site stops the local generation of the operations to turn into blocked state, and sends the GCI message to the rest of the group. At the mean time, the initiator continues the integration of remote operations.
2) When receiving a GCI message, each site stops the local generation of operations, and checks if the operations

```
1: Main:
2: JOIN
3: INITIALIZATION
4: while not aborted do
5:    if there is an input message m then
6:       GENERATE_MESSAGE(m)
7:    else
8:       RECEIVE_MESSAGE
9:    end if
10: end while

11: INITIALIZATION:
12: state ← active
13: R ← " "
14: W ← ∅
15: L ← ∅
16: s ← Identification of local user
17: initiator ← false

18: GENERATE_MESSAGE(m):
19: if m is an operation then
20:    INTEGRATE_LOCAL_OPERATION
21: else
22:    if m is a GCI then
23:       LUNCH_GC()
24:    end if
25: end if

26: RECEIVE_MESSAGE:
27: if m is an operation then
28:    if m ∈ W then
29:       W ← W − m
30:    end if
31:    INTEGRATE_REMOTE_OPERATION
32: else
33:    if m is a log request then
34:       if state=active then
35:          SEND_LOG
36:       end if
37:    end if
38: else
39:    RECEIVE_GCI_MESSAGE (m)
40: end if

41: JOIN:
42: state ← passive
43: wait θ
44: send a request log message
45: wait until receiving log
46: for all operation in the log do
47:    INTEGRATE_REMOTE_OPERATION
48: end for
```

Algorithm 1: Control Concurrency Algorithm with Garbage Collection scheme

contained in the GCI leaves have already been executed or not. To check it, we simply compute the difference between the receiver set of leaves and the GCI one. The resulting set is returned to the initiator in an ACK message. The difference represents the leaves executed by the site and not yet seen by the initiator site.

3) Each time the initiator receives an ACK message, he stores the difference locally in his own list of waited operations $\mathcal{W}$. This list is updated every time he receives one of the waited operations by extracting this operation from the set. In other words, an ACK message is causally ready when all leaves it contains are locally executed.
4) The initiator remains blocked until all ACK are received and all waited operations are executed locally. Note that waited ACK concerns only connected peers that were discovered initially by the initiator and are continuing the collaboration. New peers or disconnected peers are ignored. Thus we ensure that the initiator will not wait indefinitely for ACKs and overcome the **Second Issue**. When all waited ACK are received and $\mathcal{W} = \emptyset$, the initiator destroys his log and sends the garbage order GCO to all the group (see Algorithm 3).
5) When receiving the GCO message, a site executes il when it is causally ready. A GCO message is causally ready when all operations in its leaves set are already executed at the receiver site. Otherwise, the GCO is not ready and thus the log removal could not be executed. When the GCO message is causally ready, we verify whether the set of leaves that it contains is equal to the local set of leaves. If not, the user is ignored and considered as a new user in the group (he must request the log from other users). Otherwise, the site deletes its local log and start again the local generation with an empty log containing the same root received in the GCO.

```
1: RECEIVE_GC_MESSAGE (m)
2: if m = GCI(s', L'_s) then
3:    if s < s' and initiator = true then
4:       initiator ← false {Abort garbage collection initiation}
5:    end if
6:    state ← blocked
7:    L_r ← L \ L_s
8:    send ACK(L_r)
9: else
10:   if m = ACK(L', s') and initiator = true then
11:      W ← W ∪ L'
12:      D ← D \ {s'}
13:   end if
14: else
15:   if m = GCO(s', L') then
16:      if GCO is causally ready and L' = L
17:      clean log
18:      R ← s'
19:      state ← active
20:   end if
21: end if
```

Algorithm 2: Receive garbage message procedure

Note that each collaborating site can generate a garbage collection at any time. Hence, it is possible, that two or more users initiate two garbage procedures concurrently. To address this case, we associate a unique identifier that is randomly generated in order to ensure fairness and offer to all users the same opportunities to initiate a garbage collection procedure. We assume that these identifiers are totally ordered according to their priorities. The initiator identifier is sent in the GCI. If an initiator receives a remote GCI, he just compares his identifier to that received in the GCI message (see Algorithm 2). If he has the high priority, he continues his

garbage otherwise, he stops the local garbage procedure and respond to the initiator by an ACK messages as normal users (see Algorithm 3).

```
1: LUNCH_GC()
2: initiator ← true
3: state ← blocked
4: 𝒟 ← OnlinePeers()
5: send GCI(s,ℒ)
6: wait until 𝒟 ∩ OnlinePeers() = ∅ and 𝒲 = ∅
7: send GCO(s,ℒ)
8: initiator ← false
9: clean log
10: state ← active
```

Algorithm 3: Lunch garbage collection procedure

As our algorithm meets peer-to peer networks, users can join the group at any time. The question that arises here is: what to do when a new user joins the group during a garbage collection procedure? In fact, as discussed in Section III, this situation can lead to replicas divergence if the user that receives the new one request for log and state has not yet received the GCI message. To avoid the **Third Issue** mentioned in section III, we simply force the new peer to wait θ time before requesting the log from the nearest peer. The time θ corresponds to the maximal bound needed by a message to traverse the network from any sender to its receiver. In our model, we consider that θ is a parameter depending on network configurations. Consequently, the new users should follow the following steps. First, he sets his state to passive and waits θ time then sends a request to the nearest peer in order to get the log and the state. If the user who receives this request is in a blocked state, then the requestor must wait until the requested user is unblocked. If it is not the case ($i.e$ the requested peer is in active state) he sends his log and state to the requestor. Otherwise, he waits the reception of the GCO message and then responds (see Algorithm 1).

B. Illustrating Examples

Example 1. To illustrate the garbage collection scheme, consider the scenario in Figure 6. In this Figure, we consider three collaborating sites s_1, s_2 and s_3 where s_1 decides to initiate a garbage collection. The set of leaves of each site are referred to as $\mathcal{L}_{s_1}$, $\mathcal{L}_{s_2}$ and $\mathcal{L}_{s_3}$ respectively. To initiate the garbage collection, site s_1 stops the local generation of operations, and sends to other sites a garbage collection message $GCI(s_1, \mathcal{L}_{s_1})$ containing his set of leaves $\mathcal{L}_{s_1}$. When the GCI message arrives at sites s_2 and s_3, each site computes the difference between the received set of leaves $\mathcal{L}_{s_1}$ and the local one ($\mathcal{L}_{s_2}$ for s_2 and $\mathcal{L}_{s_3}$ for s_3). Then the resulting set is sent to the initiator of GCI (s_1) through the acquirement message ACK. The set of leaves sent in ACK message by s_2 and s_3 are added to the set of waited operations ($\mathcal{W}$) by site s_1. This list is updated every time a remote operation is received (by removing the received operation from $\mathcal{W}$). The group still blocked until all ACK messages are received by s_1 and his $\mathcal{W}$ is empty ($i.e.$ all waited operations are locally executed). Then s_1 cleans the log, adds his identity as the root the new root of the dependency tree, and sends the GCO containing the cleaning order to other sites of the group (s_1 and s_2). After the reception of the GCO message, the three sites starts again the local generation with empty logs.

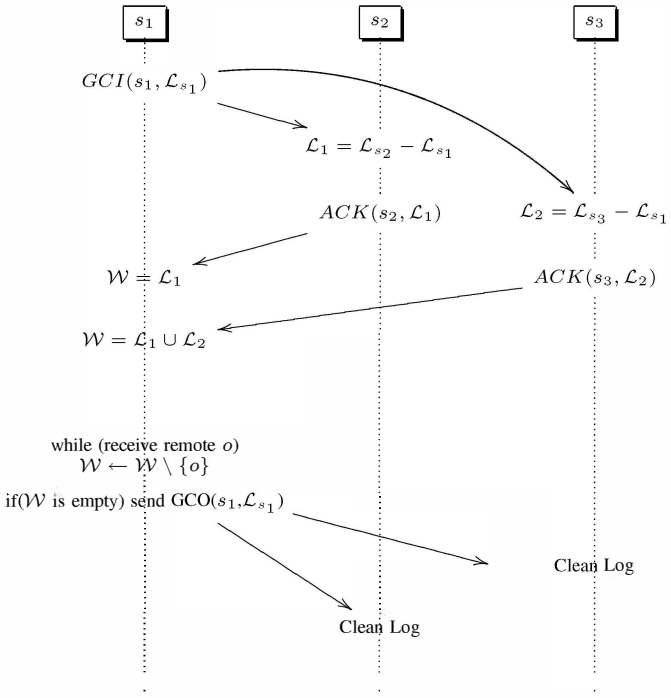


Fig. 6. Garbage collection scenario 1.

Example 2. To illustrate how we proceed garbage collection in the case of slow or non responding peers, let us consider the scenario illustrated in Figure 7 where we have three collaborating sites s_1, s_2 and s_3. Let $\mathcal{L}_{s_1}$, $\mathcal{L}_{s_2}$ and $\mathcal{L}_{s_3}$ be the set of leaves of s_1, s_2 and s_3 respectively. Suppose that site s_1 initiates the garbage collection. So, he stops the local generation of operations, and sends to other sites a garbage collection message $GCI(s_1, \mathcal{L}_{s_1})$ containing his set of leaves $\mathcal{L}_{s_1}$. When the GCI message arrives at sites s_2 and s_3, each site computes the difference between the received set of leaves $\mathcal{L}_{s_1}$ and the local one ($\mathcal{L}_{s_2}$ for s_2 and $\mathcal{L}_{s_3}$ for s_3). Suppose that $\mathcal{L}_{s_1} = \emptyset$ which means that s_2 has the same set of leaves as s_1 and that $\mathcal{L}_{s_3} \neq \emptyset$. Moreover, s_3 is a slow peer, thus his ACK is not received by s_1. When s_1 receives s_2's ACK, he rediscovers connected peers and finds that s_3 does not respond. Consequently, he only consider the ACK of s_2. Since $\mathcal{W} = \emptyset$, s_1 sends the GCO message. When received by s_2, he deletes his log since $\mathcal{L}_1 = \mathcal{L}_2$. However, at site s_3, the log removal is not executed until all s_1 leaves are executed at site s_3 and $\mathcal{L}_{s_3} = \mathcal{L}_{s_1}$ to be sure that all operations seen at site s_1 when initiating the GC are received by s_3. It should be noted that if site s_3 has generated an operation concurrently to the garbage collection procedure ($\mathcal{L}_{s_3} \neq \mathcal{L}_{s_1}$), this operation will be lost since the set of leaves is different from that of the initiator. Consequently, the user will be rejected from the garbage collection procedure and considered as a new user

who joins the group thus needing to request the log and the state after the garbage ends. This issue does not concern local networks since the loss of messages is not frequent but may be faced in large networks such as Internet.

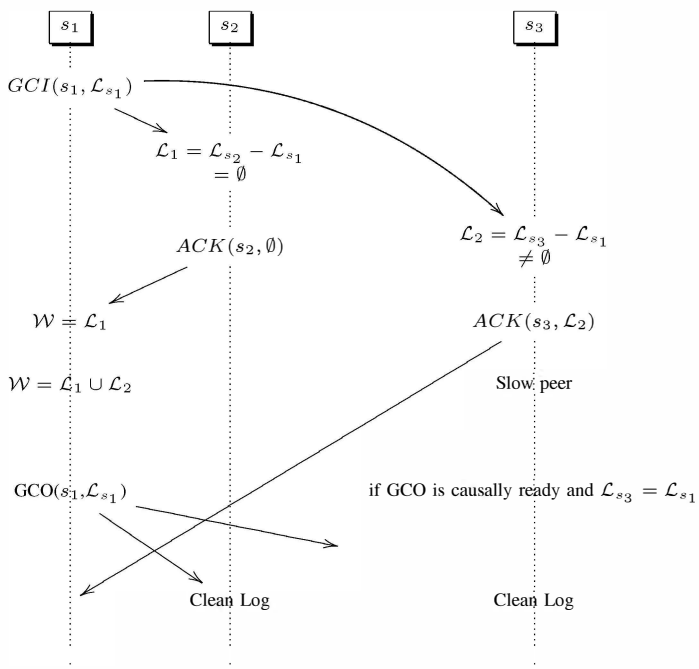


Fig. 7. Garbage collection scenario 2 (case of a slow peer).

V. Implementation and Performance Measurement

To validate our model, we have implemented a prototype based on the model discussed before using Java Platform, Micro Edition (Java ME) [14] which provides a robust and flexible environment for the embedded applications, and proposes two configurations to simulate mobile environments: CDC (Connected Device Configuration) [15] specifies an environment for terminals connected where memory is usually greater than 512 Kb such as tablets screen phones, digital television and cell phones as Nokia 9500, Sony Ericsson P990i and Samsung C6620. CLDC (Connected Limited Device Configuration) [16] target devices with limited or low resources such as mobile phones, PDAs, or light wireless peripheral. For example, BlackBerry, Nokia (6600,E73,N93,...), Motorola (i560,i730,...) and Sumsung (A737,D500,...) offer CLDC environment. The realized prototype was developed with netbeans 6.8 under Windows operating system. In Figure 8, we see different screen shots of the main windows on the CLDC environment while in Figure 9), we illustrate a screen shot of our prototype implemented for CDC environment.

A. Response Time

For our evaluation performance, we consider the following times (see Figure 10) used to calculate the response time of our model:

- t_g is the time required to generate a local operation;
- t_i is the time to integrate a remote operation;

Fig. 8. Screen shots of the CLDC prototype.

Fig. 9. Screen shot of the CDC prototype.

- t_c is the time required to communicate an operation to a peer through network;
- t_r is the response time, we can obviously see that $t_r = t_g + t_i + t_c$.

In general, it is established that the OT-based collaborative editors must provide $t_r < 100ms$ [6]. The lower the response time is the better the collaboration is. As a matter of fact, the user is able to see different updates made on the shared documents instantly.

To investigate the performance of our prototype realized for mobile phones, we did experimental tests for the behavior of the two developed models (CDC and CLDC). The first experiment consists of calculating the response time in the worst case. Note that the worst case occurs when the log contains 100% delete operations(see [5]). Then we measured the time required to generate an insertion and integrate it at a remote site.

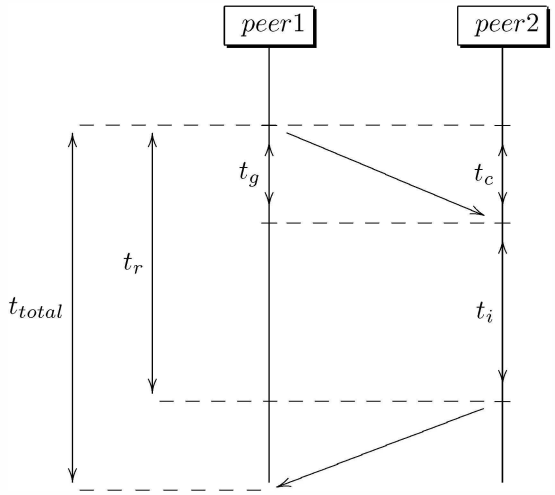


Fig. 10. Response time.

Figure 11 shows the response time for different log size values for the CDC environment. These measurements reflect the times t_g, t_i and their sum t_r. The execution time falls within $100ms$ for all $|H| \leq 27500$. Figure 12 shows the response time for different log size values for the CLDC environment. In this case, the execution time falls within $100ms$ for all $|H| \leq 14000$.

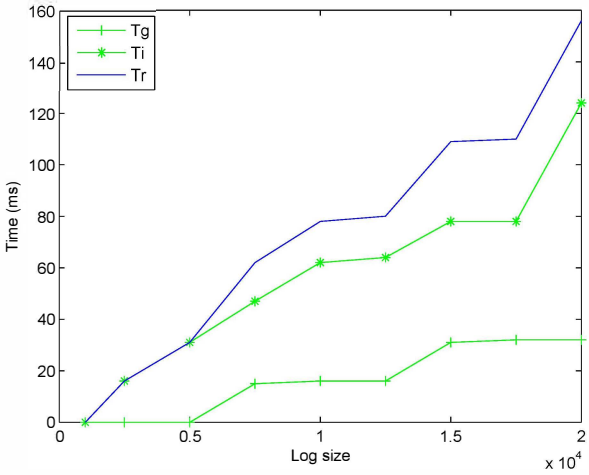


Fig. 11. Response Time for CDC mobile phone.

We conclude that beyond 27500 (respc. 14000) for the CDC (respc. CLDC) environment, logs should be cleaned through garbage collection mechanism. This result is encouraging since it allows for a large number of operations that users can exchange before reaching the maximal born defined for response time (and thus a lower quality for collaboration). When reaching given sizes, all users will start again the collaboration with empty logs which makes the collaboration more and more efficient.

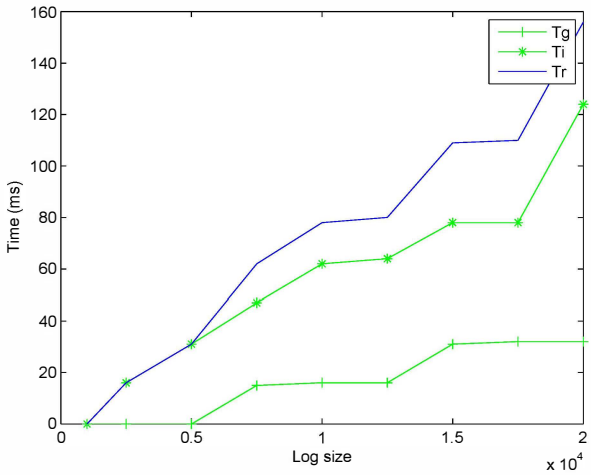


Fig. 12. Response Time for CLDC mobile phone.

B. Garbage Collection Time

The following experiment is realized to measure the blockage time needed by the group in order to perform garbage collection and collaborate again. The experiment measures this time for different values of the leaves set owned by the initiator. According to the results shown in Figure 13, we deduce that the blockage time is acceptable till the size 10000, where users have to wait only 5 seconds to start again their collaboration. It should be noted that 10000 does not represent the log size but rather the set of leaves size which means that the log could contains more than this number of operations.

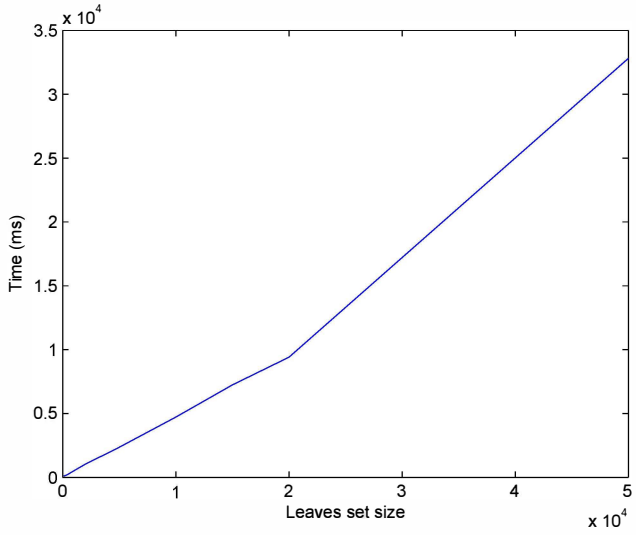


Fig. 13. Garbage collection time variation with leaves set size.

VI. Conclusion

In this work we proposed a garbage collection scheme for real time collaborative editor based on OT approach and

built upon peer-to-peer networks. The main objective of this scheme is to extend decentralized collaborative editors to mobile devices as mobile phones, PDA, etc. Compared to other OT-based collaborative editors, our editor proposes a novel design for distributed garbage collection.

Our garbage collection scheme allows the integration of collaborative editors in mobile devices, it preserves the data convergence, and optimizes the editor log size, and hence offers better performances. The most important income is that our solution is well suited to dynamic groups. However, it is limited by blocking the collaboration during garbage collection process since all users who receive GCI message block the local generation of operations until the end of the garbage collection process. Moreover, slow sites may lose their operations when receiving the garbage order. In such situation, our garbage collection scheme enforces these slow peers to be considered as new peers joining the group in order to recover correct log and shared state from peers who have successfully completed the garbage collection procedure. Consequently, despite of the loss of operations for slow peers, the data convergence is still ensured. Note that the loss of operations does not occur in small networks, but only for wide connections ($e.g$ Internet).

In future work, we plan to improve our garbage collection scheme by avoiding blocking garbage collection scheme and integrating off line work. We will also investigate in developing a security design that meets collaborative work requirements for mobile devices.

REFERENCES

[1] Clarence A. Ellis and Simon J. Gibbs (1989): *Concurrency Control in Groupware Systems*. *SIGMOD Conference* 18, pp. 399–407.

[2] C. Sun (1998): *Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems*. *ACM Transactions on Computer-Human Interaction* 5, pp. 63–108.

[3] Matthias Ressel and Doris Nitsche-Ruhland and Rul Gunzenhauser (November 1996): *An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors*. *ACM CSCW'96"*, Boston, USA. pp. 288-297.

[4] Brad Lushman and Gordon V. Cormack (2003): *Proof of correctness of Ressel's adOPTed algorithm*, *Information Processing Letters* 86, pp. 303–310, Elseiver B.V.

[5] A. Imine (2009) *Coordination Model for Real-Time Collaborative Editors*. *COORDINATION*, pp. 225-246.

[6] Du Li and Rui Li (2008), *An Operational Transformation Algorithm and Performance Evaluation* 17, *Computer Supported Cooperative Work*, pp. 469-508.

[7] P. Samarati and P. Ammann and S. Jajodia (1996): *Maintaining replicated authorizations in distributed database systems*. *Data & Knowledge Engineering journal* 18, pp. 55–84.

[8] P. Roy and S. Seshadri and A. Silberschatz and S. Sudarshan and S. Ashwin (1997). *Garbage collection in object oriented databases using transactional cyclic reference counting*, *In VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pp. 366–375.

[9] Sunil Sarin and Nancy Lynch and A. Lynch (1987), *Discard Obsolete Information In A Replicated Database System*. *IEEE Transaction on Software Engineering*, Vol. SE-13, No. 1, pp. 39-47.

[10] K. M, Chandy and L. Lamport (1985). *Distributed snapshots: Determining Global States of Distributed System*,vol 3, No.1. *ACM Transactions on Computer Systems*, pp. 63-75.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson (1985): *Impossibility of distributed consensus with one faulty process*. *Journal of the ACM*, 32(2), pp.374-382.

[12] A. Mostfaoui and M. Raynal (1999): *Solving Consensus Using Chandra-Touegs Unreliable Failure Detectors: A General Quorum-Based Approach*, *Springer Velag LNCS* 1693, pp. 49-63.

[13] M. Raynal (2005): *Short Introduction to Failure Detectors for Asynchronous* , *ACM SIGACT News, Distr. Computing Column*, 36(1), pp. 53-70.

[14] http://java.sun.com/javame/index.jsp.

[15] http://java.sun.com/javame/technology/cdc/.

[16] http://java.sun.com/products/cldc/.