Collaborative Editing Application in Mobile Ad-hoc Networks

By Le, Nam Nguyen Hoai

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Xavier Défago

September, 2012

Collaborative Editing Application in Mobile Ad-hoc Networks

By Le, Nam Nguyen Hoai (1010227)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Xavier Défago

and approved by
Associate Professor Xavier Défago
Professor Mizuhito Ogawa
Professor Mikifumi Shikida

August, 2012 (Submitted)

Acknowledgements

I would like to thank Associate Professor Xavier Defago, my supervisor. He introduced me into the world of distributed system and provided great advice during my research. Besides the major knowledge, I have learned many skills from his attitude in the research and teaching.

I was very pleasant and honored to have Dr. Francois Bonnet, Mr. Daiki Higashihara, Mr. Hiroyuki Hiranuma, Mr. Nguyen Dang Thanh, Mr. Nguyen Xuan Huy as my colleagues. They supported me a lot not only in my study but also in my life in JAIST.

I would like to thank my professors and friends in FIVE-JAIST Program.

Last but not least, I would like to say many thanks to my family. They have been always beside me and encouraged me.

Contents

List of Figures

Chapter 1

Introduction

1.1 Collaborative Editing Application

Collaborative editing application is a major part of Computer Supported Collaborative Work (CSCW). It enables a group of people to work together on a shared document at the same time by using their devices. Concretely, users can edit a document concurrently and watch changes from each other. Using a collaborative editing application, a group of people can create a final document reflecting group contribution.

Collaborative editing application is best known as a tool for developing software, allowing programmers to write code together. It is also used for members in a meeting to take notes on topics, create and revise a document. In education, collaborative editing application is a demonstration tool.

With the advance of mobile technology, people can use mobile devices as tools for collaborating and working anytime, anywhere. Mobile device gradually becomes more powerful and connected over wireless network. That allows developing applications that enable users, using their own mobile devices, to collaboratively work on the same document without the presence of a fixed network infrastructure. Therefore, collaborative editing application in mobile ad-hoc networks (MANETs) attracts a lot of interest.

1.2 Centralized vs. Decentralized Architecture

In a centralized architecture, collaborative editing application has to maintain a central server. All updates of users are sent to the server and, after processing, the server sends the updated document to all clients. The central server keeps the shared document and manages all aspects of the collaboration. Using a centralized architecture, collaborative editing application is easier to handle concurrent updates. Some collaborative editing applications use this architecture such as Rendezvous [1], Dolphin [2], Jupiter [3]. However, it has a number of drawbacks in the context of MANETs. Firstly, the collaboration completely depends on a central server; if the server fails, the collaboration stops immediately. In MANETs, the presence of the server is not always guaranteed. Secondly, in MANETs, user is frequently disconnected from the server because of the mobility, so the collaboration is less responsive. Consequently, a centralized architecture is less suitable for collaborative editing application in MANETs.

Conversely, in a decentralized architecture, every user carries a copy of the shared document. Each user makes any changes on local document, and after that, sends directly those changes to all other users. Using this architecture, every user is responsible for managing the collaboration in the absence of a central server. Compared to centralized architectures, decentralized architectures have some interesting advantages. Firstly, a central server is not required in a decentralized architecture, so it avoids single point of failure. Secondly, when a user is disconnected from other users, it is still able to work on its local document. When reconnecting, disconnected user can synchronize with other users to update its local document. Thirdly, in a decentralized architecture, all users can concurrently make changes on their local documents anytime regardless of the status of the server and the communication between them and the server. Clearly, that makes decentralized architecture more attentive for collaborative editing application in MANETs. Some examples of group editors using this architecture are Colab [4], GROVE (Group Outline Viewing Editor) [5], REDUCE (Real-time Distributed Unconstrained Cooperating Editing) [6, 7], GroupDesign [8], GRACE (Graphics Collaborative Editing) [9], and Draw together [10].

1.2.1 Decentralized Collaborative Editing

Besides greater flexibility and potential, a collaborative editing application in a decentralized architecture has the following major challenges:

- Document consistency: Every user in the collaboration session can update its copy whenever it wants, which promotes concurrency in the collaboration. However, in that context, it is difficult to keep all copies of the shared document to be similar and correctly reflect the intention of every user. Currently, most of collaborative editing applications are based on Operational Transformation approach to solve this problem. However, in MANETs, high latency causes a large number of concurrent operations while according to this approach, every concurrent operation needs processing to be correctly executed, which affects application performance, especially since the processing power of mobile device is still lower than current PC.

- Fault tolerance: In MANETs, mobile device frequently gets disconnected from other devices because of device mobility, fluctuating bandwidth or device failure. Consequently, in the collaboration session, when a user is disconnected, it cannot broadcast and receive messages consisting of joining, leaving, document operation to/from users that it is disconnected from. That affects the process of the joining, leaving, editing in the collaboration session. In the context of MANETs, every user is responsible for tolerating disconnection event without the presence of a central server.

1.2.2 Commutative Replicated Data Type and TreeDoc

Commutative replicated data type (CRDT) [26] is a new approach that supports eventually consistent information. This approach focuses on designing data structures such that operations can commute with each other. Therefore, consistency is archived without concurrent control.

TreeDoc [19] is a CRDT in which information is structured by a binary tree. With TreeDoc, the identifier of each node is its path on the tree. Furthermore, during the process of updating the tree, TreeDoc ensures that the path of each node is unique and not changed, so the different orders of the execution of concurrent operations lead the same state.

1.3 Contributions

The main contribution of our work is the development of an Android-based application for decentralized collaborative editing based on the concept of TreeDoc. We have adapted TreeDoc to our need in document consistency management. Our application minimizes overhead of TreeDoc by developing rebalancing. In the evaluation, we carry out the the experiment on real environment with Android mobile devices to compare the performance of TreeDoc with Operational Transformation, an approach used by most of collaborative editing applications, and consider rebalancing with application performance in order to determine the most suitable value of rebalancing period.

We provide a solution for fault tolerance in our application in order for a user who went too far outside of the transmission range of other devices can get missing messages when it has reconnected. By experiment, we investigate the effect of message loss probability to application performance.

1.4 Thesis Organization

This thesis is organized in the chapters as follow:

- Chapter 2 discusses the properties which the system must satisfy to ensure document consistency and Operation Transformation approach.
- In chapter 3, we present document consistency management in our application with the concept of TreeDoc.
- Chapter 4 is the implementation of our application.
- The performance of our application is analyzed in Chapter 5.
- Chapter 7 demonstrates the functions of our application.
- Chapter 8 provides the conclusion of this thesis and the future work.

Chapter 2

Background

2.1 System Model

The system consists of a collection of users, each of which maintains a copy of the shared document containing a set of characters. User can make any operations on its local document whenever it wants. In order to enable all users to know the last state of the shared document, any operations generated by any users have to be broadcast to all other users. After receiving the remote operation, each user replays the remote operation on its own copy such that all copies of the shared document are consistent.

Operations on the shared document are of two types:

- Insert(*posID*,*newchar*) inserts a character *newchar* at position *posID* into the document
- Delete(*posID*) deletes the character located by *posID* within the document

2.1.1 Causality

For any pair of operations, op_i and op_j generated by user S_i and S_j respectively, op_i and op_j can be either causally related or concurrent in the sense of the Lamport's happened-before relation [11].

Definition 2.1. Causal precedence ($\rightarrow$)
op_i causally precedes op_j ($op_i \rightarrow op_j$) if:

- op_i and op_j are generated by the same user ($S_i = S_j$) and op_j is generated after op_i.
- op_i and op_j are generated by two different users ($S_i \neq S_j$) and S_j executed op_i before generating op_j.
- There exists an operation op_k, such that $op_i \rightarrow op_k$ and $op_k \rightarrow op_j$.

Definition 2.2. Concurrence ($||$)
op_i and op_j are said to be concurrent ($op_i \mid\mid op_j$) if $op_i \nrightarrow op_j$ and $op_j \nrightarrow op_i$

2.1.2 State Vector

State vector [13] is a technique for capturing causal precedence among all operations in a system. It is a modification to the clock vector introduced by Mattern [18].

Definition 2.3. State vector

Let us define the set of cooperative users in a system S_1, S_2,..., S_N. Each user S_i in the system maintains a state vector which is a vector with N components $V_i = (V_i[1], V_i[2],..., V_i[N])$ in which $V_i[j]$ holds the number of operations generated by user S_j that has been executed by user S_i. In the beginning, $V_i[j] = 0$ with j=$(1, ..., N)$. After S_i executed an operation generated by user S_j, $V_i[j] = V_i[j] + 1$.

Definition 2.4. Causal precedence with state vector

op_i generated by S_i causally precedes op_j generated by S_j ($op_i \rightarrow op_j$) if $V_{op_i}[i] < V_{op_j}[i]$ where V_{op_i} is the state vector of S_i when it generates op_i, V_{op_j} is the state vector of S_j when it generates op_j

2.2 Document Consistency Properties

According to [7], a collaborative editing system is said to be consistent if it respects the following properties:

Causality preservation property

If op_i causally precedes op_j ($op_i \rightarrow op_j$), op_i is executed before op_j on all copies of the shared document. In other words, when a remote operation arrives at a user, it is executed only when all operations causally preceding it have been executed at that user. Figure 2.1 shows that when operations are executed in the same causal order, all copies are consistent, while Figure 2.2 shows otherwise.

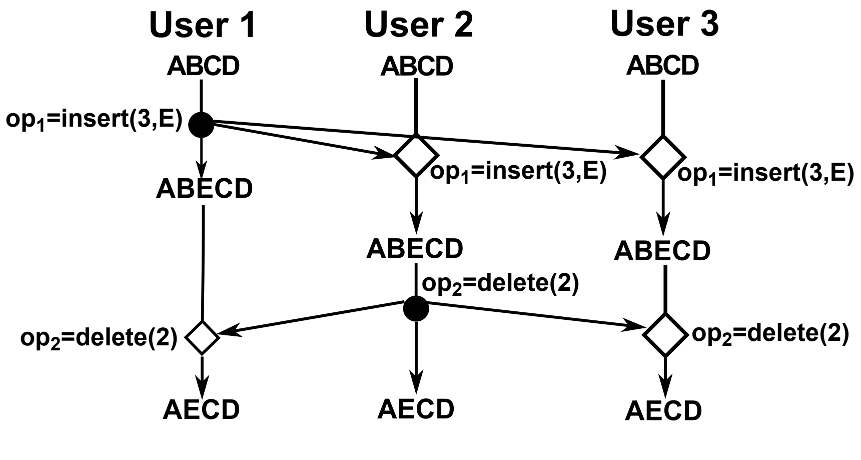


Figure 2.1: Causality Presevation Property

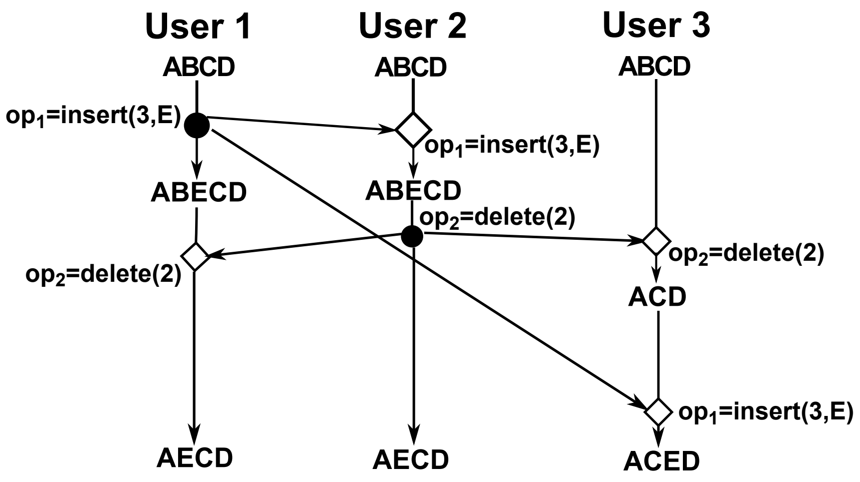


Figure 2.2: No Causality Presevation Property

Intention preservation property

The effect of an operation at all users must be the same as the intention of the user generated that operation. Operation intention for textual document is:

- Delete: If a character is deleted on a copy, it must be deleted on all other copies.
- Insert: A character inserted on a copy must be inserted on all other copies such that the order relation between it and other document characters is the same on all copies.

The example in Figure 2.3a presents the case in which the intention of a delete operation is not preserved. In this example, user 1 and user 2 generate two concurrent operations op_1 and op_2 on the same object "ABCD". User 1 generates $op_1 = insert(2,\text{`E'})$ with the intention of inserting 'E' between 'A' and 'B', and user 2 generates $op_2 = delete(3)$ with the intention of deleting 'C'. However, when op_2 is executed at user 1, deleted character is 'B', which is not the intention of op_2. That causes inconsistency.

The example in Figure 2.3b presents the case in which the intention of a insert operation is not preserved. User 1 generates $op_1 = insert(4,\text{`F'})$ with the intention of inserting 'F' between 'C' and 'D', and concurrently, user 2 generates $op_2 = insert(2, \text{`E'})$ with the intention of inserting 'E' between 'A' and 'B'. However, when op_1 is executed at user 2, 'F' is inserted between 'B' and 'C', which is not the intention of op_1. That causes inconsistency.

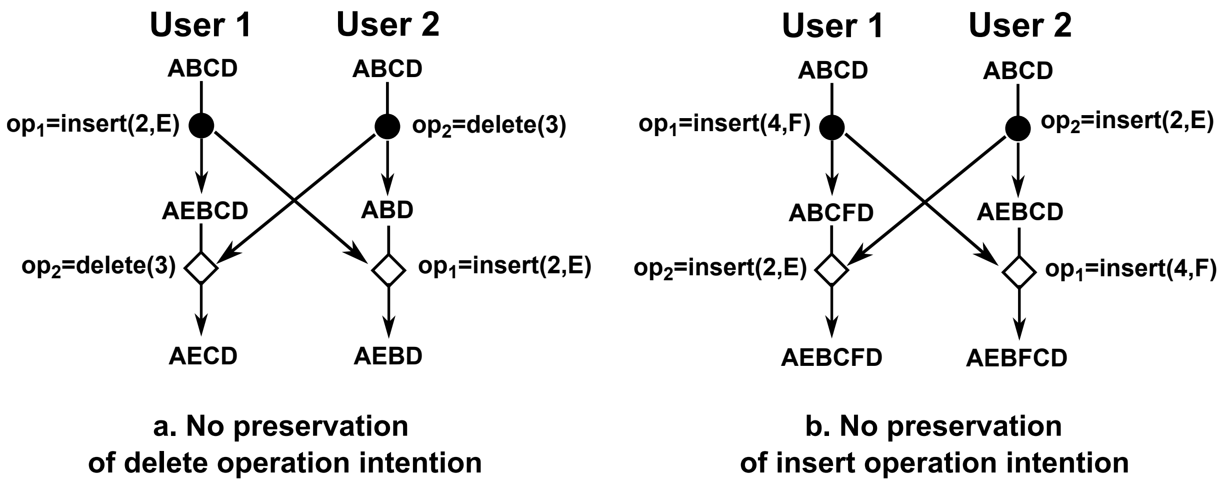


Figure 2.3: No Intention Presevation Property

Convergence property

After executing the same collection of operations, all users reach the same state. That is eventual consistency [12].

2.3 Operational Transformation

Operational Transformation introduced by Ellis and Gibbs [13] with the DOPT algorithm is well established for concurrency control in group editing. Currently, many products use this approach such as Gobby, SubEthaEdit, ACE and most recently Google Wave. This approach manages the document by a sequential array of characters and uses the index of the character in the array for its *posID*. Using this approach, the local operation is executed immediately on local copy to ensure responsiveness, and then it is broadcast to all other users. When a user receives a remote operation, it is transformed against all of its concurrent operations in the history by forward transformation procedure before it is executed to preserve its intention.

Consider the example in Figure 2.4, to preserve the intention, op_2 in the example shown in Figure 2.3a should be transformed forward against op_1 to become $op_2^{op_1} = delete(4)$ because after user 1 executes op_1, ‘C’ is in position 4 (Figure 2.4a), and op_1 in the example shown in Figure 2.3b should be transformed forward against op_2 to become $op_1^{op_2} = insert(5,‘F’)$ because after user 2 executes op_2, the position between ‘C’ and ‘D’ is position 5 (Figure 2.4b)

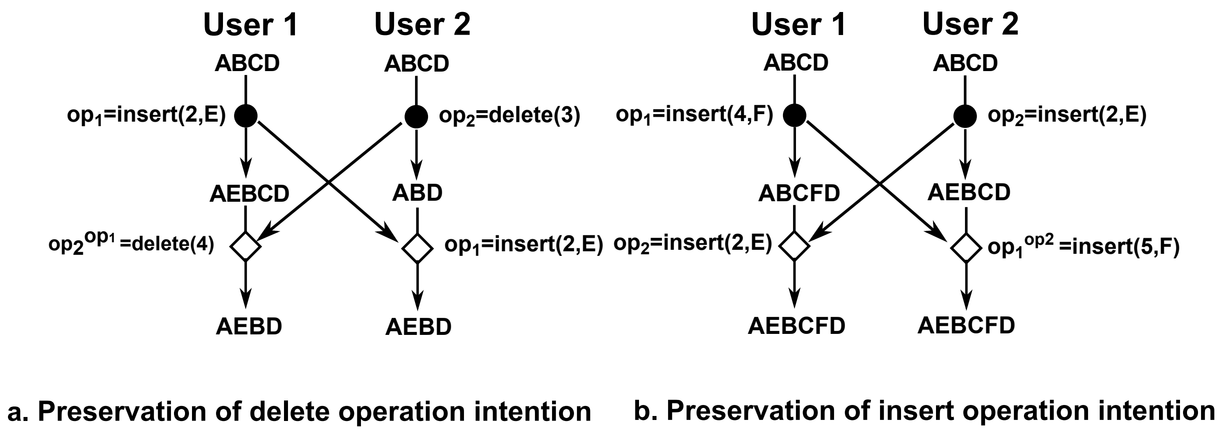


Figure 2.4: Operational Transformation

Specification 1 presents the forward transformation procedure.

Specification 1 Forward Transformation

```
    Forward transformation of $op_1 = delete(posID_1)$ against $op_2 = delete(posID_2)$.
 1: procedure FWTRANSFORM($op_1, op_2$)
 2:     if $posID_1 > posID_2$ then
 3:         $op_1 = delete(posID_1 - 1)$;
 4:     end if
 5:     if $posID_1 < posID_2$ then
 6:         $op_1 = delete(posID_1)$;
 7:     end if
 8:     if $posID_1 = posID_2$ then
 9:         $op_1 = null$;
10:     end if
11: end procedure
    Forward transformation of $op_1 = delete(posID_1)$ against $op_2 = insert(posID_2, newchar_2)$.
12: procedure FWTRANSFORM($op_1, op_2$)
13:     if $posID_1 \geq posID_2$ then
14:         $op_1 = delete(posID_1 + 1)$;
15:     else
16:         $op_1 = delete(posID_1)$;
17:     end if
18: end procedure
```

```
    Forward transformation of $op_1 = insert(posID_1, newchar_1)$ generated by $S_{op_1}$ against
    $op_2 = insert(posID_2, newchar_2)$ generated by $S_{op_2}$.
19: procedure FWTRANSFORM($op_1, op_2$)
20:     if $posID_1 > posID_2$ then
21:         $op_1 = insert(posID_1 + 1, newchar_1)$;
22:     end if
23:     if $posID_1 < posID_2$ then
24:         $op_1 = insert(posID_1, newchar_1)$;
25:     end if
26:     if $posID_1 = posID_2$ then
27:         if $newchar_1 = newchar_2$ then
28:             $op_1 = null$;
29:         else
30:             if $S_{op_1} > S_{op_2}$ then
31:                 $op_1 = insert(posID_1, newchar_1)$;
32:             else
33:                 $op_1 = insert(posID_1 + 1, newchar_1)$;
34:             end if
35:         end if
36:     end if
37: end procedure
    Forward transformation of $op_1 = insert(posID_1, newchar_1)$ against $op_2 = delete(posID_2)$
38: procedure FWTRANSFORM($op_1, op_2$)
39:     if $posID_1 > posID_2$ then
40:         $op_1 = insert(posID_1 - 1, newchar_1)$;
41:     else
42:         $op_1 = insert(posID_1, newchar_1)$;
43:     end if
44: end procedure
```

However, Ressel et al. in [14] showed that DOPT can not ensure document consistency in all cases. Concretely, inconsistent states happen when current operations are generated on different states. As depicted in Figure 2.5, user 1 generates op_1, op_3 and user 2 generates op_2. When user 2 receives op_1, it transforms op_1 against op_2 because $op_1||op_2$. Similarly, when user 2 receives op_3, it transforms op_3 against op_2 because $op_3||op_2$ and when user 1 receives op_2, it transforms op_2 against op_1 and op_3 because $op_2||op_1$ and $op_2||op_3$. However, in this case, two users are inconsistent after executing op_1, op_2, and op_3 because transforming op_3 against op_2 at user 2 can not preserve the intention of op_3. That is because op_1 and op_2 are concurrent and generated on the same state, while although op_3 is concurrent to op_2, they are generated on the different states. That is a typical case in the sense of partial concurrency [23] when an operation is concurrent to a sequence of operations.

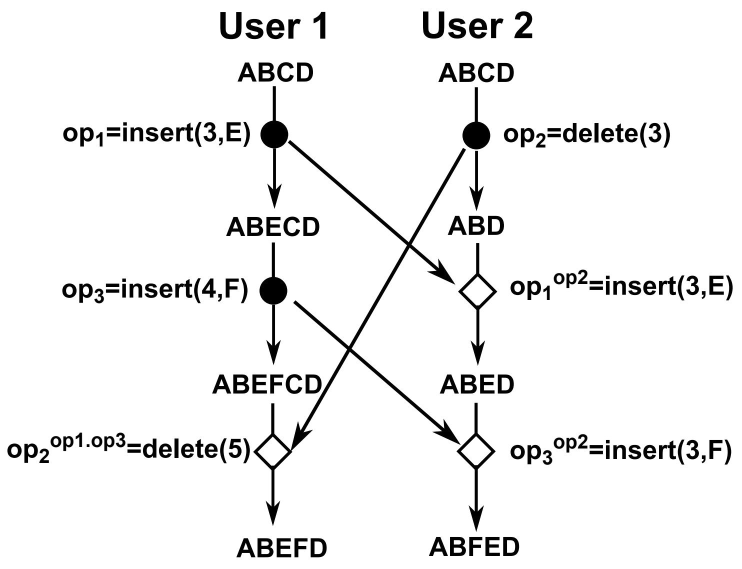


Figure 2.5: Inconsistency with Operational Transformation of DOPT Algorithm

This problem is solved by SOCT2 [15]. Concretely, when a remote operation is executed at a user, SOCT2 uses backward transformation technique to shift all of its preceding operations backward to the beginning of the history and all of its concurrent operations to the end of the history. After separating history, the remote operation will be transformed forward against the set of its concurrent operations. Figure 2.6 presents how SOCT2 solves the inconsistent case in Figure 2.5.

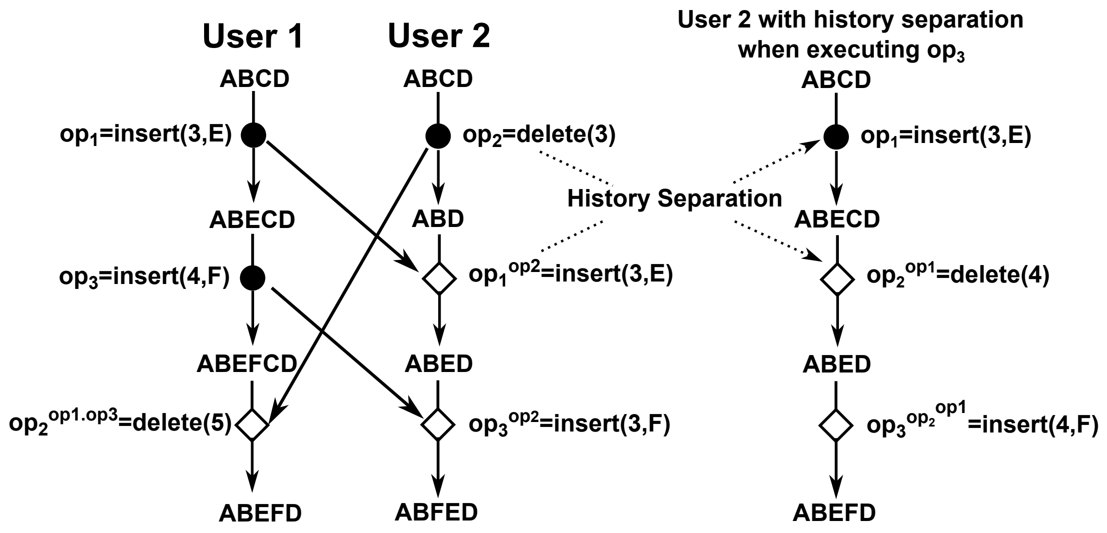


Figure 2.6: SOCT2 solves the Inconsistent Case in Operational Transformation of DOPT Algorithm

Specification 2 presents the backward transformation procedure.

Specification 2 Backward Transformation

```
    Backward transformation of $op_1 = delete(posID_1)$ against $op_2 = delete(posID_2)$.
 1: procedure BWTRANSFORM($op_1, op_2$)
 2:     if $posID_1 \geq posID_2$ then
 3:         $op_1 = delete(posID_1 + 1)$;
 4:     else
 5:         $op_1 = delete(posID_1)$;
 6:     end if
 7: end procedure
    Backward transformation of $op_1 = delete(posID_1)$ against $op_2 = insert(posID_2, newchar_2)$.
 8: procedure BWTRANSFORM($op_1, op_2$)
 9:     if $posID_1 > posID_2$ then
10:         $op_1 = delete(posID_1 - 1)$;
11:     else
12:         $op_1 = delete(posID_1)$;
13:     end if
14: end procedure
    Back transformation of $op_1 = insert(posID_1, newchar_1)$ against
    $op_2 = insert(posID_2, newchar_2)$.
15: procedure BWTRANSFORM($op_1, op_2$)
16:     if $posID_1 > posID_2$ then
17:         $op_1 = insert(posID_1 + 1, newchar_1)$;
18:     else
19:         $op_1 = insert(posID_1, newchar_1)$;
20:     end if
21: end procedure
    Backward transformation of $op_1 = insert(posID_1, newchar_1)$ against $op_2 = delete(posID_2)$.
22: procedure BWTRANSFORM($op_1, op_2$)
23:     if $posID_1 > posID_2$ then
24:         $op_1 = insert(posID_1 + 1, newchar_1)$;
25:     else
26:         $op_1 = insert(posID_1, newchar_1)$;
27:     end if
28: end procedure
```

However, Vidot et al. in [16] indicated that SOCT2 is inconsistent under a scenario when concurrent operations are not performed in the same order at all users. Consider the example in Figure 2.7, when operation op_3 generated by user 3 arrives at user 1 and user 2, it is transformed against its concurrent operations. Concretely, at user 1, op_3 is transformed in turn against op_1 and op_2, while at user 2, op_3 is transformed in turn against op_2 and op_1. Unfortunately, the results of transforming op_3 at user 1 and user 2 are different, which causes to be inconsistent. Therefore, to reach consistent state, the result of transforming an operation against a sequence of its concurrent operations must not depend on the orders of the execution of those operations at all users.

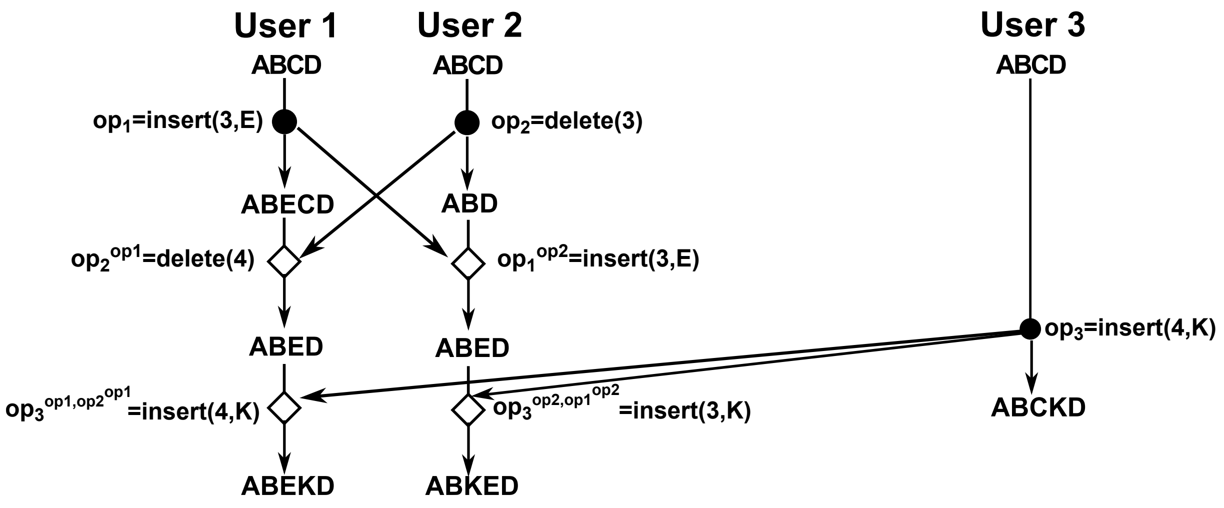


Figure 2.7: Inconsistency with SOCT2

In [16], Vidot et al. introduced SOCT3 to solve the inconsistent scenario of SOCT2 by an additional step to SOCT2. Concretely, like SOCT2 when a remote operation is received, it is transformed forward against the set of its concurrent operations obtained after the step of history separation. After that, operations in the history are rearranged by basing on the total ordering scheme by executing forward transformation and backward transformation. The rearrangement of the history after executing a remote operation ensures the order of operations in the history of all users to be the same. Figure 2.8 presents how SOCT3 solves the inconsistent case in Figure 2.7.

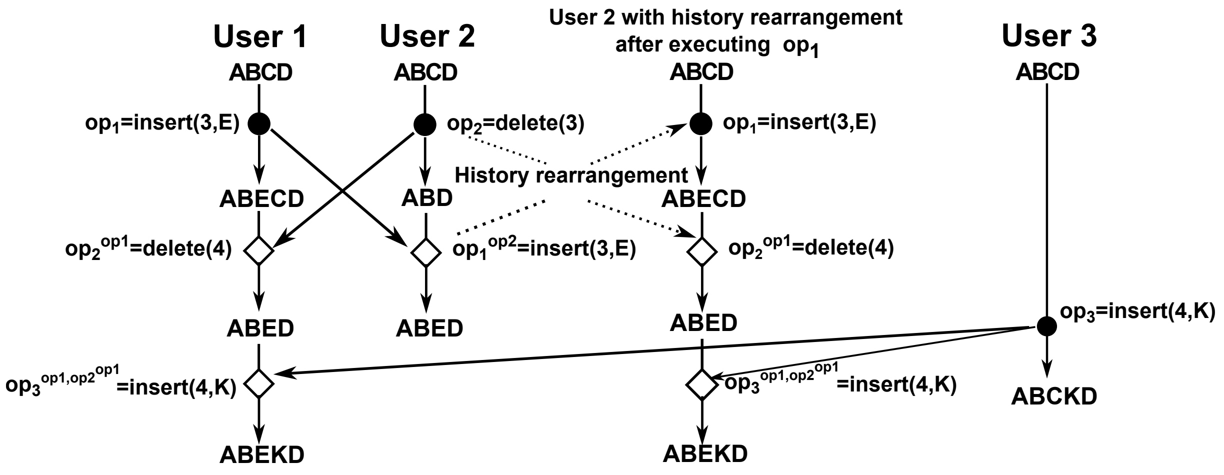


Figure 2.8: SOCT3 solves the Inconsistent Case in SOCT2

However, in SOCT3, the transformation procedure becomes very complex and error-prone [17]. Especially, with the large collaborative editing system in MANETs, many users and high network latency can cause a large number of concurrent operations while according this approach, every remote operation must be transformed against all of its concurrent operations to ensure document consistency. Therefore, it consumes a lot of processing power, while the process power of mobile device is low. Consequently, it is not suitable for implementation on mobile device in the large collaborative editing application.

Chapter 3

Document Consistency with TreeDoc

In this chapter, we present TreeDoc [19], a commutative replicated data type that supports eventually consistent information. In our application, we have adapted the concept of TreeDoc to our need to ensure convergence and intention preservation properties and use state vector technique to preserve causality property.

3.1 TreeDoc

TreeDoc [19] is a commutative replicated data type (CRDT) [26] designed for concurrent editing without concurrency control. It manages the document as a binary tree, and the content of document is defined by infix-order visit on the whole tree. Consider the example of Figure 3.1, the content of the document corresponding with TreeDoc is “ABCDEF”.

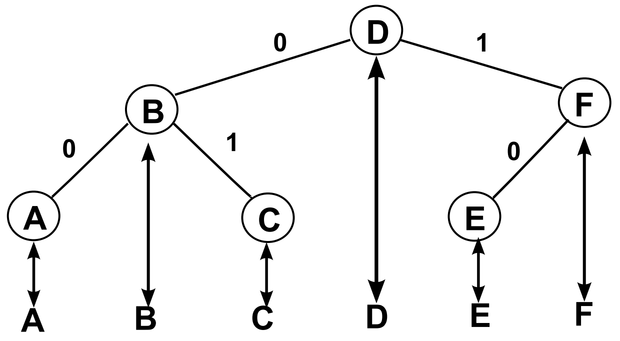


Figure 3.1: TreeDoc

Instead of using an array to manage the document and the index of the character in the array for its identifier like Operational Transformation approach, in TreeDoc, the identifier of the character(*posID*) in the document is the path of the corresponding node in the tree. The path of node is a sequence of bit {0;1} in which 0 stands for left branch and 1 stands for right branch on the tree. For example, *posID* of node A; B; C; E in Figure 3.1 are $00; 0; 01; 10$, respectively.

3.1.1 Local Operation

When inserting a new character, the generator generates the corresponding node to the tree at the position such that the correlation between its position and the position of all other characters on the tree is guaranteed. After that, the generator sends inserted character and its identifier which is the path of the corresponding node to all other users. For example, to insert character X at right position of character D on the tree of Figure 3.1, because node D has right child, which is node F, node X is inserted at the leftmost position of the subtree rooted at node F, and then the generator gets the path of node X, which is 100. After executing that insert operation, the tree becomes the tree in Figure 3.2.

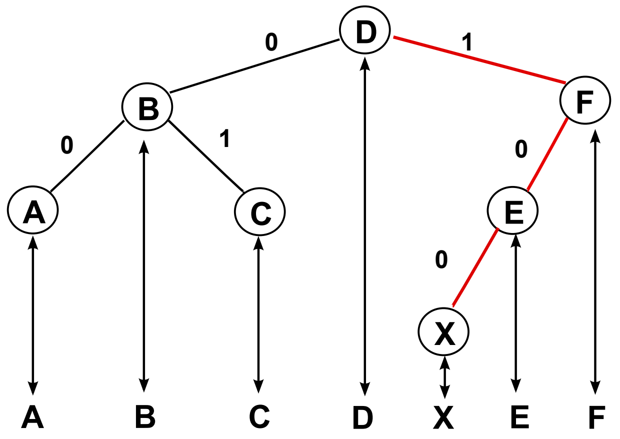


Figure 3.2: TreeDoc after inserting X at right position of character D

Another case is to insert character G at right position of character F on the tree of Figure 3.2. In this case, since node F has not right child, node G is right child of F as shown in Figure 3.3.

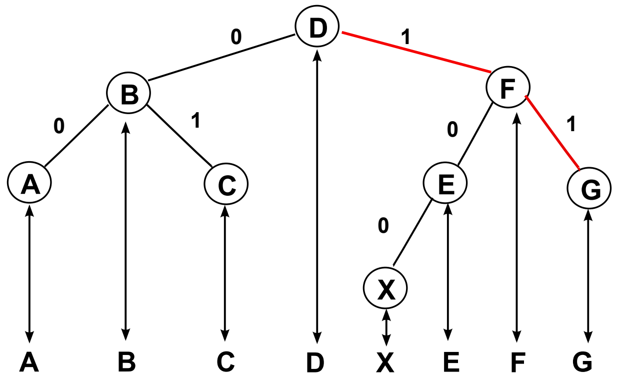


Figure 3.3: TreeDoc after inserting G at right position of character F

When deleting a character, the generator sets the corresponding node to empty node and sends the path of the corresponding node on the tree to all other users. For example, deleting character B in the tree of Figure 3.3 is executed by setting the corresponding node to empty node, and the tree becomes the tree of Figure 3.4.

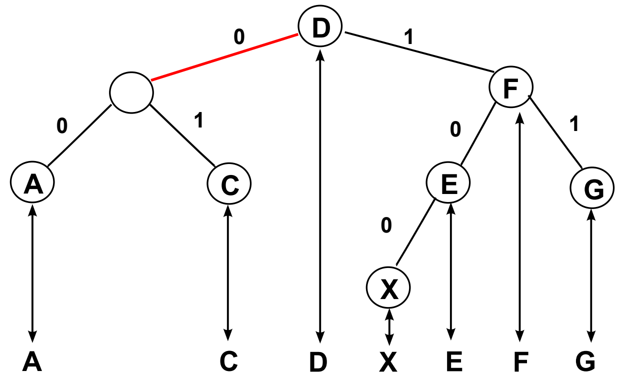


Figure 3.4: TreeDoc after deleting B

Specification 3 describes the procedures for generating node, applying local insert operation and local delete operation to TreeDoc.

Specification 3 Application of local operation to TreeDoc

```
    Generation of new node $N$ at right position node $M$
 1: procedure GenerateNode($M$)
 2:     if $M$ has right child then
 3:         Create node $N$ at the leftmost position of the subtree rooted at right child of node $M$;
 4:     else
 5:         Create node $N$ is the right child of node $M$;
 6:     end if
 7:     return node $N$;
 8: end procedure
    Application of locally deleting the character corresponding node $N$ on tree
 9: procedure ApplyLocalDeleteOperation($N$)
10:     Set node $N$ to empty node;
11:     Send the path of node $N$ to all other users;
12: end procedure
```

```
    Application of locally inserting character $newchar$ in right position of node $M$
13: procedure ApplyLocalInsertOperation($M$,$newchar$)
14:     Node $N$= GenerateNode($M$);
15:     Set the content of node $N$ to $newchar$;
16:     Send <the path of node $N$,$newchar$ > to all other users;
17: end procedure
```

3.1.2 Remote Operation

When executing remote delete operation, user gets the node at the position corresponding with the identifier of deleted character on the tree, and sets it into empty node. When executing remote insert operation, user creates the node at the position on the tree corresponding with the identifier of inserted character, and sets its content into inserted character. Specification 4 describes the procedures for applying remote insert operation and delete operation to tree.

Specification 4 Application of remote operation to TreeDoc

```
    Application of remote delete operation in which the path of deleted node is $posID$.
1:  procedure ApplyRemoteDeleteOperation($posID$)
2:      Get node $N$ whose path is $posID$
3:      Set node $N$ to empty node
4:  end procedure
    Application of remote operation in which inserted character is $newchar$ and path of
    inserted node is $posID$.
5:  procedure ApplyRemoteInsertOperation($posID$,$newchar$)
6:      Create node $N$ whose path is $posID$;
7:      Set the content of node $N$ to character $newchar$;
8:  end procedure
```

3.1.3 Commutative Replicated Data Type

For any pair of concurrent operations op_1 and op_2, op_1 is said to be commutative with op_2 if on an initial state, both the execution of op_1 before op_2 and the execution of op_2 before op_1 result in the same state in which the intention of op_1 and op_2 are preserved. To prove that with the execution of update operation on TreeDoc under the way as above, two concurrent operations op_1 and op_2 can commute with each other, we investigate the following cases:

Two concurrent insert operations ($op_1 = insert(posID_1, newchar_1)$, $op_2 = insert(posID_2, newchar_2)$)

When a user finishes executing op_1 and op_2, the share document contains both $newchar_1$ and $newchar_2$. Because inserting a node to a tree does not change the path of all other nodes on that tree, $posID_1$ and $posID_2$ is unique and does not depend on the orders of the execution of op_1 and op_2. In other word, the intention of op_1 and op_2 are always preserved. Hence, the final state is the same.

Two concurrent delete operations ($op_1 = delete(posID_1)$, $op_2 = delete(posID_1)$)

To delete a character on the document, TreeDoc sets the corresponding node on the tree to empty node, so executing delete operation does not change the path of all other nodes on the tree. Furthermore, the $posID$ of a node on the tree is unique, so the existence of two nodes having the same $posID$ is impossible. Therefore, after executing op_1 and op_2, the share document does not contain $deletedchar_1$ which is the deleted character of op_1 and $deletedchar_2$ which is the deleted character of op_2, and $posID$ of all other characters is the same as before. In other words, the final state is the same.

Concurrent delete operation and insert operation ($op_1 = delete(posID_1)$, $op_2 = insert(newchar_2, posID_2)$)

As indicated above, the $posID$ of a node on the tree is unique during the execution of insert and delete operation. After executing op_1 and op_2, the share document includes $newchar_2$ at the unique position $posID_1$ and does not contain $deletedchar1$ which is the deleted character of op_1. Therefore, the final state is the same.

3.1.4 Rebalancing

Besides the advantages, TreeDoc has the problem of overhead. Firstly, during the execution of insert operations, the tree can become unbalanced, which causes the path of inserted node to grow indefinitely. For instance, if a user always appends to the end, the path of node will grow with each new character. Secondly, the tree can accumulate many empty nodes because of executing delete operations. To alleviate this problem, rebalancing is executed periodically to make the tree balanced and discard empty nodes.

Rebalancing TreeDoc

Rebalancing process creates a binary balanced tree whose document content is the same as document content of initial tree. Concretely, with initial tree corresponding to the document consisting of k characters, this process operates on the initial tree by generating nodes, removing nodes, assigning again the content of each node such that it becomes complete binary tree including k nodes, whose height is equal to $\lceil \log_2(k+1) \rceil$-1, and document content is not changed. Figure 3.5 shows the example of rebalancing TreeDoc in Figure 3.5a to TreeDoc in Figure 3.5b.

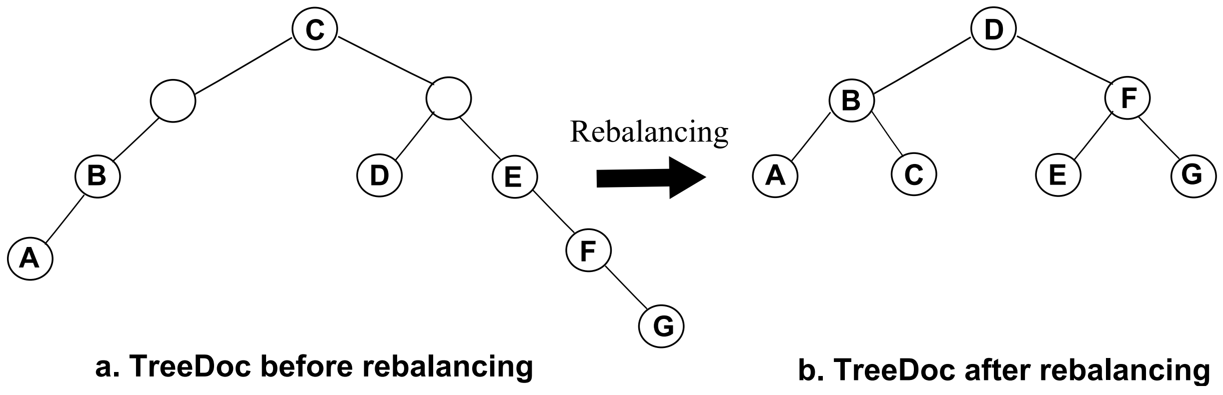


Figure 3.5: Rebalancing TreeDoc

Specification 5 describes the procedure for rebalancing TreeDoc.

Specification 5 Rebalancing TreeDoc

```
   Rebalancing TreeDoc $t$
1: procedure REBALANCE($t$)
2:     $k$= the number of nonempty nodes in $t$;
3:     $h = \lceil \log_2(k+1) \rceil$-1;
4:     Visit $t$ in infix order to create nodes, remove nodes to reach complete binary tree
   whose height is equal $h$, and assign again the content of $t$ to be the same as before;
5: end procedure
```

Rebalancing TreeDoc in the large-scale system

Although the sequential order of characters in the document is preserved after rebalancing, their identifiers is changed. Consequently, only updates on the copies executing rebalancing on the same state can be exchanged with each other. Therefore, before rebalancing tree, all users must agree on the same state of their trees by running a commitment protocol. However, it is difficult for commitment protocol to execute quickly, successfully in MANETs, which causes update operations to be blocked a long time.

Two-tier architecture

The solution for rebalancing tree in large-scale system is two-tier architecture introduced by Zawirski et al. in [20] in which all nodes are divided into two set, core set and nebular set:

- The core set includes well-connected nodes. Only core nodes participate in commitment protocol. With above property of core set, commitment protocol is executed quickly.
- Nebular set is dynamic, weakly connected, even disconnected. They only generate tree updates and do not participate in commitment protocol.

Every node defines an epoch e numbered sequentially for each rebalance.

Rebalance in core set

When a core node initiates rebalancing, it executes a commit protocol in which it plays the role of the coordinator to agree on the same final state in core set before executing rebalancing. If all core nodes agree on the same final state in epoch e, they execute rebalancing on that final state in epoch e to create the initial state in epoch $e + 1$.

Rebalance in nebular set

Because nebular nodes are not allowed to participate in commitment protocol for rebalancing, after core nodes execute rebalancing, nebula nodes are behind with core nodes at one or more epochs. Consequently, all updates generated by nebular nodes cannot be replayed at core nodes, although they can be replayed on all other nebular nodes in the same epoch. The catch-up protocol updates a nebula node to the next epochs.

Catch-up protocol

When a nebular node N in epoch e contacts a node C in epoch $e + 1$, N executes catch-up protocol to update its copy. Concretely, C sends all updates in epoch e to N. After receiving all updates in epoch e, noted by u_e, from C, N applies u_e to the initial state in epoch e to create the final state in epoch e and executes rebalancing on the final state in epoch e to create initial state in epoch $e + 1$. After that, N translates updates which it executed in epoch e but not belong to u_e into epoch $e + 1$, then applies and broadcasts them. Specification 6 describes the procedure for catching up.

Specification 6 Catching up nebular node in epoch e to epoch e+1

```
    Catching-up nebular node N in epoch $e$ after receiving all update in epoch $e$, noted
    by $u_e$, from node C in epoch $e + 1$
 1: procedure CATCH-UP($u_e$)
 2:     Apply $u_e$ to the initial state in epoch $e$ to reach the final state in epoch $e$;
 3:     Rebalance the final state in epoch $e$ to create the initial state in epoch $e + 1$;
 4:     Translate updates which is not in $u_e$ into epoch $e + 1$, then apply and send them
    to all other user;
 5: end procedure
```

3.2 Causality Preservation with State Vector

When a user updates its local copy, it sends the operation to all other users with the current value of its local vector. In [13], if each component of the state vector V_i, which is the state vector of user S_i, is greater than or equal to the corresponding component of the state vector V_j, which is the state vector of user S_j, user S_i has already executed all operations that have been executed by user S_j. Furthermore, to preserve causality property, when a remote operation *op* arrives at user S_i, *op* is only executed if all operations causally preceding it have already been executed by user S_i. Consequently, user S_i only executes *op* when each component of its state vector is greater than or equal to the corresponding component of the state vector V_{op}. Specification 7 presents the procedures for executing local operation and remote operation to preserve causality property by state vector technique.

Specification 7 Causality Preservation with State Vector technique

```
    Execution of local operation op at user $S_i$ with state vector $V_i$.
1:  procedure ExecuteLocalOperation(op)
2:      ApplyLocalOperation(op);
3:      $V_{op} = V_i$;
4:      $V_i[i]++$;
5:      Send <op,$V_{op}$> to all other user;
6:  end procedure
    Execution of remote operation op with state vector $V_{op}$ generated by user $S_j$ at user
    $S_i$ with state vector $V_i$.
7:  procedure ExecuteRemoteOperation(op,$V_{op}$,$S_j$)
8:      if $V_i[k] \geq V_{op}[k]$ , ($k$: $1 \leq k \leq N$) then
9:          ApplyRemoteOperation(op);
10:         $V_i[j]++$;
11:     else
12:         Wait until $V_i[k] \geq V_{op}[k]$ ,($k$: $1 \leq k \leq N$);
13:     end if
14: end procedure
```

Chapter 4

Implementation

4.1 System Architecture

In this section, we present the system architecture for every collaborative user. Each collaborative user consists of the following components as shown in Figure 4.1:

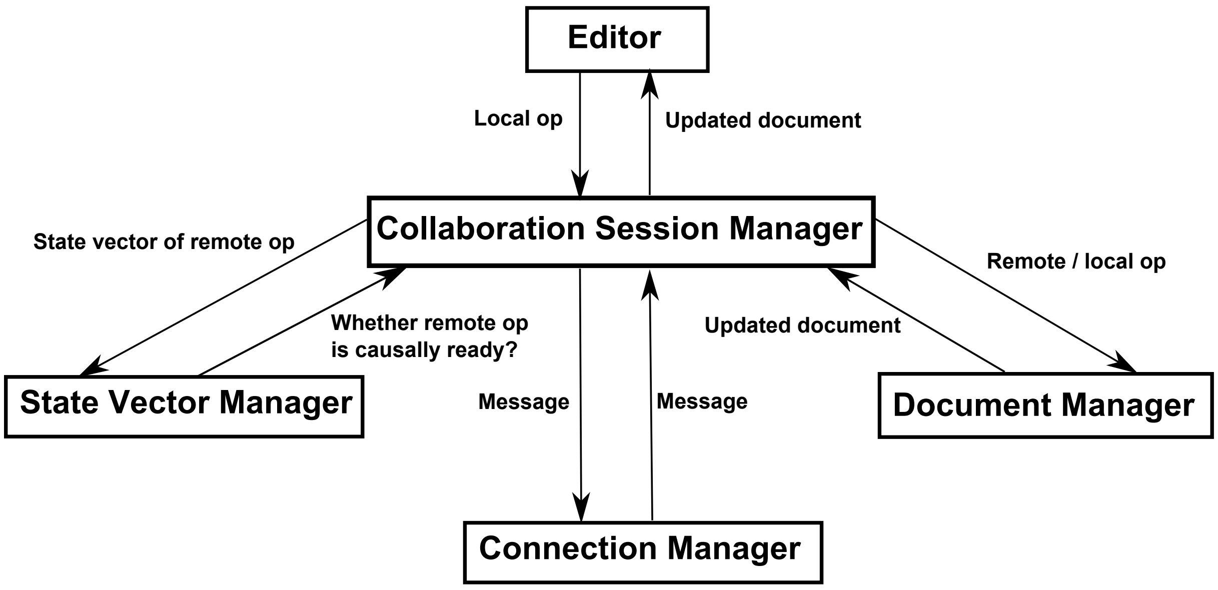


Figure 4.1: System Architecture

- Document manager is responsible for applying local operation/remote operation on the document structure, then returning new content of the document. In order to edit, rebalance in the collaboration, document management of user S_i in epoch n has to include as follow:
  - For ensuring document consistency on current document, current state x_n, structured by TreeDoc, is required.

  - For helping other users to perform catch-up protocol to update their documents to epoch n, S_i has to keep all operations that it executed before.
  - If S_i is a nebular user, in the future, it will perform catch-up protocol to update its document to the document in the epoch of core set. In order to do that, a nebular user in epoch n needs to keep the initial state and all operations that it is executed in the epoch n. Because the initial state of an epoch is balanced tree, it is managed by an array structure of characters to reduce overhead.

- State vector manager is responsible for determining whether remote operation is causally ready.

- Collaboration session manager is responsible for managing the entire collaboration. Concretely, when it receives a message related to the remote operation, it forwards state vector to state vector manager, operation to document manager, and the new content of the document to editor after executing the remote operation. Furthermore, when it receives local operation from editor, it forwards operation to document manager, updates state vector, and sends message related to that local operation to all other users through connection manager.

- Connection Manager is responsible for sending and receiving messages to and from all other uses

- Editor is the tool for editing the shared document.

4.2 Class Diagram

Figure 4.2 shows the class diagram of our collaborative editing application. In general, *TreeDoc* structure contains a *Node root*. *Node* in *TreeDoc* includes:

- *atom* is the character corresponding with the node
- *leftChild* is left child node
- *rightChild* is right child node
- *parent* is parent node.
- *Font* is the font of the node

In a *Node*, *Font* consists:

- *fontStyle* are Bold, Italic, Underline, Quote, Strike-through.
- *fontType* are Normal, Monospace, Serif, Sans-Serif.
- *fontSize* is the size of the character.
- *fonttextColor* is the color of the character.

- *fontbgColor* is the color of background of the character.

StateVector is structured by an array of states corresponding with an array of users in the system. When a remote operation is causally ready, the state corresponding with the user generating that operation will be incremented.

Epoch contains a list of executed operations to perform catch-up protocol in the future.

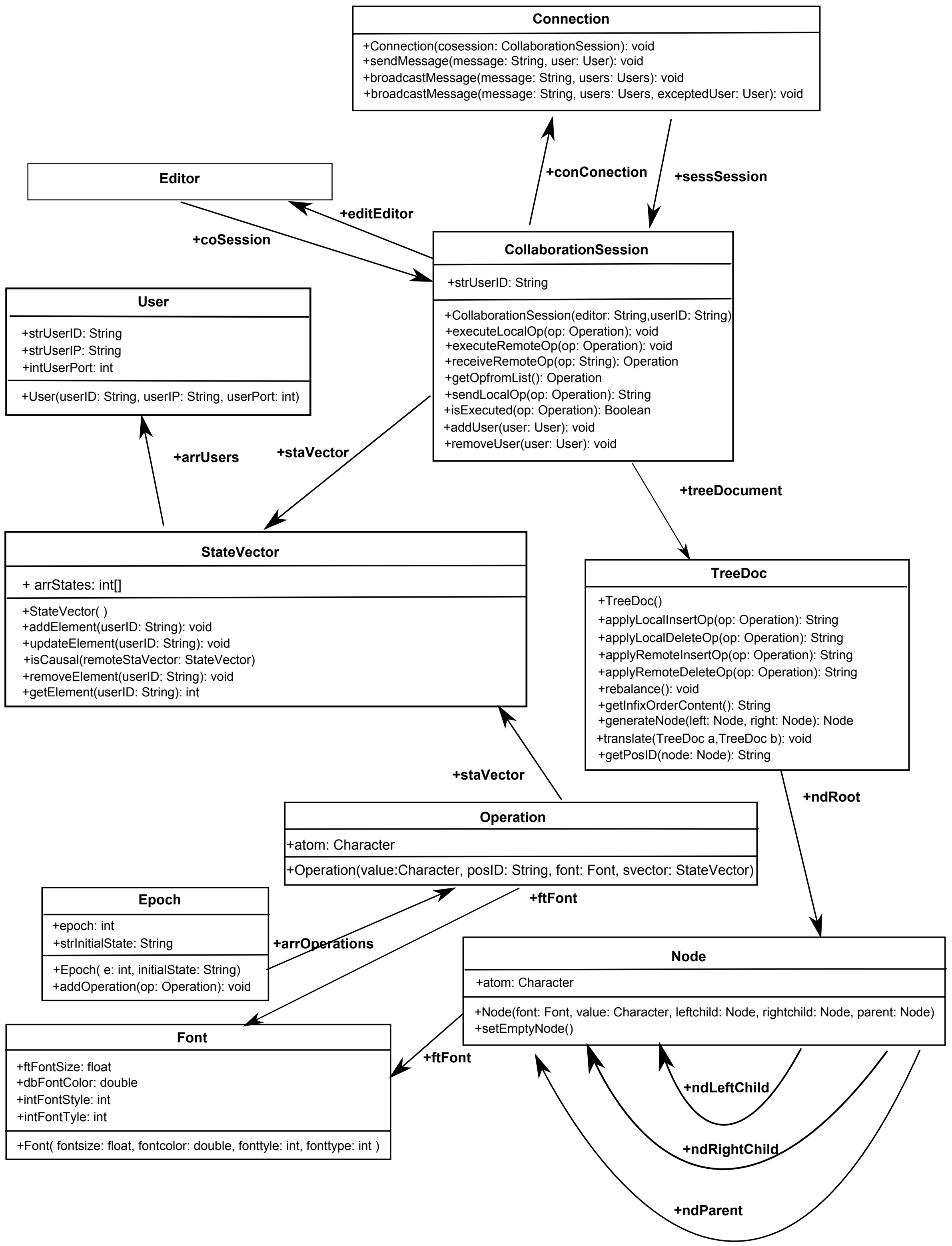

Connection
+Connection(cosession: CollaborationSession): void
+sendMessage(message: String, user: User): void
+broadcastMessage(message: String, users: Users): void
+broadcastMessage(message: String, users: Users, exceptedUser: User): void
+conConection
+sessSession
Editor
+editEditor
+coSession
CollaborationSession
+strUserID: String
+CollaborationSession(editor: String,userID: String)
+executeLocalOp(op: Operation): void
+executeRemoteOp(op: Operation): void
+receiveRemoteOp(op: String): Operation
+getOpfromList(): Operation
+sendLocalOp(op: Operation): String
+isExecuted(op: Operation): Boolean
+addUser(user: User): void
+removeUser(user: User): void
User
+strUserID: String
+strUserIP: String
+intUserPort: int
+User(userID: String, userIP: String, userPort: int)
+arrUsers
+staVector
+treeDocument
StateVector
+ arrStates: int[]
+StateVector( )
+addElement(userID: String): void
+updateElement(userID: String): void
+isCausal(remoteStaVector: StateVector)
+removeElement(userID: String): void
+getElement(userID: String): int
TreeDoc
+TreeDoc()
+applyLocalInsertOp(op: Operation): String
+applyLocalDeleteOp(op: Operation): String
+applyRemoteInsertOp(op: Operation): String
+applyRemoteDeleteOp(op: Operation): String
+rebalance(): void
+getInfixOrderContent(): String
+generateNode(left: Node, right: Node): Node
+translate(TreeDoc a,TreeDoc b): void
+getPosID(node: Node): String
+staVector
+ndRoot
Operation
+atom: Character
+Operation(value:Character, posID: String, font: Font, svector: StateVector)
Epoch
+epoch: int
+strInitialState: String
+Epoch(e: int, initialState: String)
+addOperation(op: Operation): void
+ftFont
+arrOperations
Node
+atom: Character
+Node(font: Font, value: Character, leftchild: Node, rightchild: Node, parent: Node)
+setEmptyNode()
Font
+ftFontSize: float
+dbFontColor: double
+intFontStyle: int
+intFontTyle: int
+Font(fontsize: float, fontcolor: double, fonttyle: int, fonttype: int)
+ftFont
+ndLeftChild
+ndRightChild
+ndParent

Figure 4.2: Class Diagram

4.3 Sequence Diagram

Figure 4.3 shows the sequence diagram for the process of a local operation. Concretely, when user S_i edits a character on local *Editor*, *CollaborationSession* gets that operation (*op*) from *Editor*, applies it to *TreeDoc*, and updates the element related to user S_i in *StateVector*. Finally, *op* and state vector of S_i when it generates *op* is sent asynchronously to all other users though *Connection*.

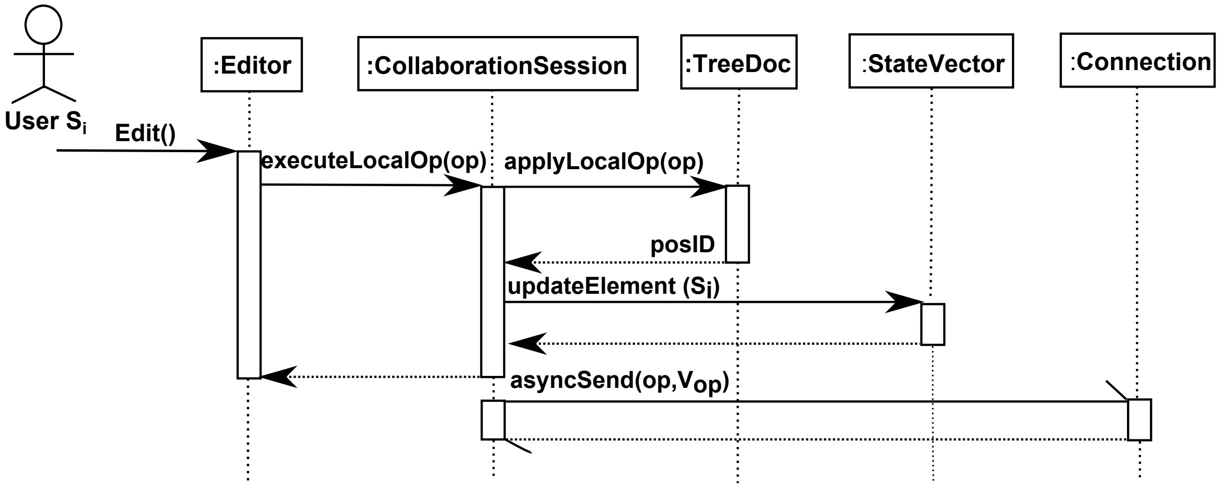


Figure 4.3: Sequence Diagram for the process of Local Operation

Figure 4.4 shows the sequence diagram for the process of a remote operation. When *Connection* receives a remote operation from other user, it is added to a list of operations that is waiting for be executed. *CollaborationSession* gets a operation from the list of operations, checks whether it is causally ready, and execute it on *TreeDoc* in the case of causal condition. After that, *CollaborationSession* sets the new content of the shared document to *Editor*.

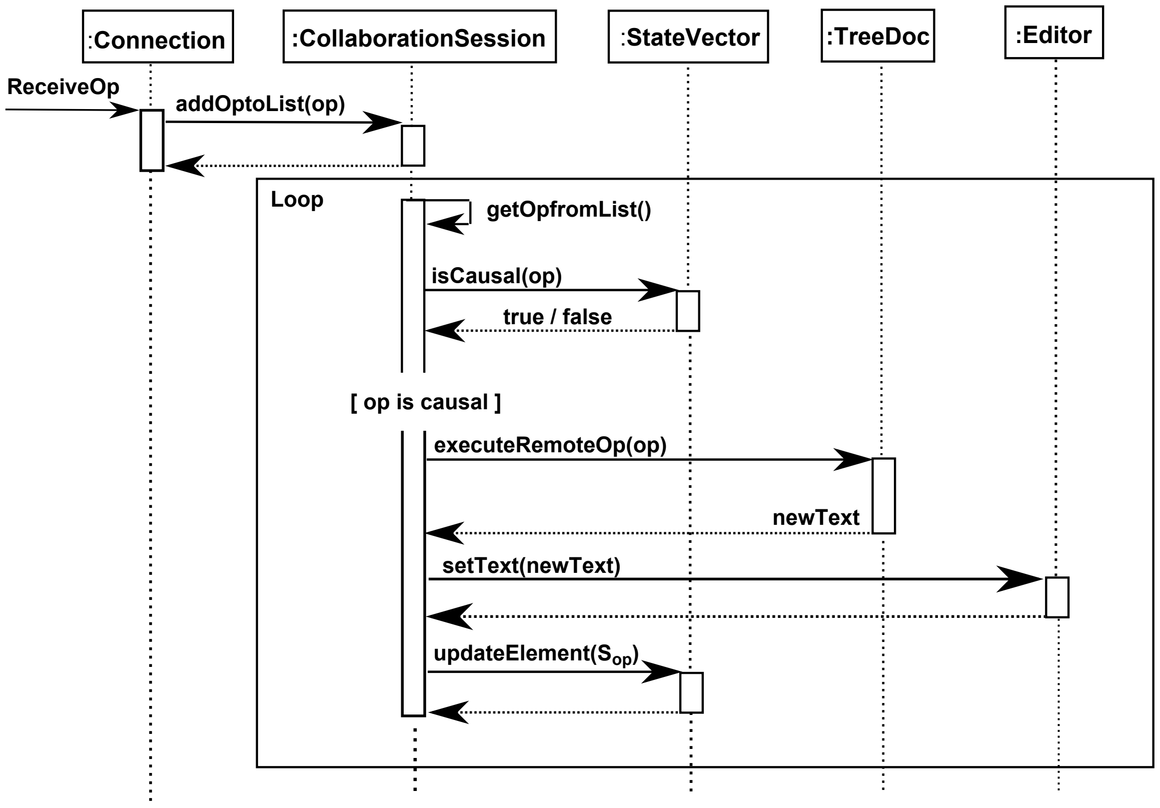


Figure 4.4: Sequence Diagram for the process of Remote Operation

Chapter 5

Evaluation

In this chapter, we compare the performance of two approaches for ensuring document consistency, TreeDoc and Operational Transformation in SOCT3 [16]. After that, we investigate the effect of rebalancing to the performance of TreeDoc.

In Operational Transformation approach, each remote operation must be transformed against all its concurrent operations in the history to preserve its intention. Therefore, operation time of this approach depends on the number of concurrent operations in the system.

In contrast to Operational Transformation, using TreeDoc, concurrent operations can commute with each other. Consequently, operation time of TreeDoc does not depend on the number of concurrent operations in the system and only depends on editing context of every operation. Concretely, in TreeDoc, time for executing local operation depends on time for walking on tree from the corresponding node to root to get *posID*, while time for executing remote operation depends on time for walking on tree from root to the node corresponding with *posID*. In other words, operation time of TreeDoc depends on the height of the node corresponding with every operation. Furthermore, total height of all nodes on the tree is minimal if the tree is always balanced during the execution of operations (operation i at height h=$\lceil \log_2(i+1) \rceil$-1, $i = 1...n$) like Figure 5.1. That is the best case for operation time of TreeDoc. The worst case for operation time of TreeDoc is the case of only editing at a certain position. In that case, the height of node increases steadily during the execution of operations (operation i at height $h = i - 1$, $i = 1...n$) like Figure 5.2, and total height of all nodes on the tree tree is maximal.

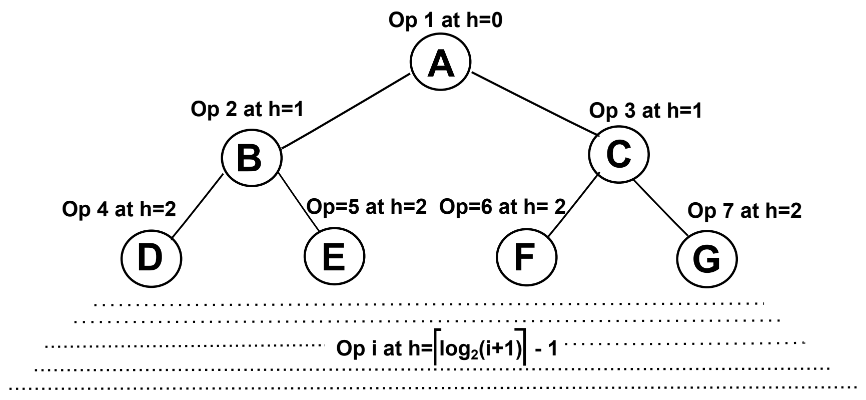


Figure 5.1: Best Case of TreeDoc: the tree is balanced, the depth is logarithmic

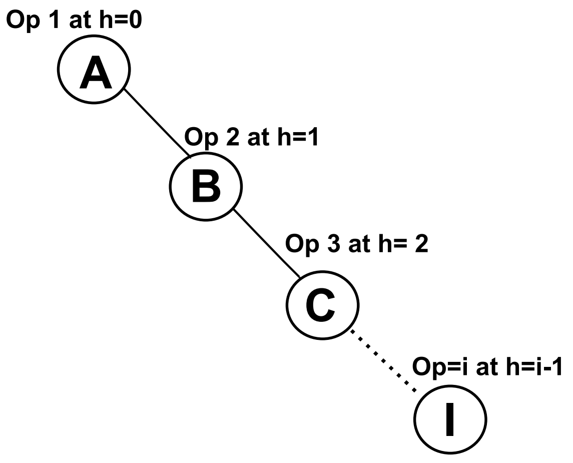


Figure 5.2: Worst Case of TreeDoc: the tree degenerates as a list with every new character added to the right, the depth is linear.

5.1 Experiment Setting

We experiment on 4 real android device Samsung GT-I9250 Galaxy Nexus (CPU speed: 1.2 GHz Dual Core Processor; Internal phone storage: 16GB; RAM: 1GB; OS version: Android OS 4.0). Because Android 4.0 does not officially build the ability to connect to ad-hoc networks, in our experiments, every mobile device is connected with each other by wireless network in which AirMac Extreme 802.11n Wi-Fi is the based station as shown in Figure 5.3. All mobile devices are located at the same position, and we use Ping command, we estimate network delay in this position by 77 ms.

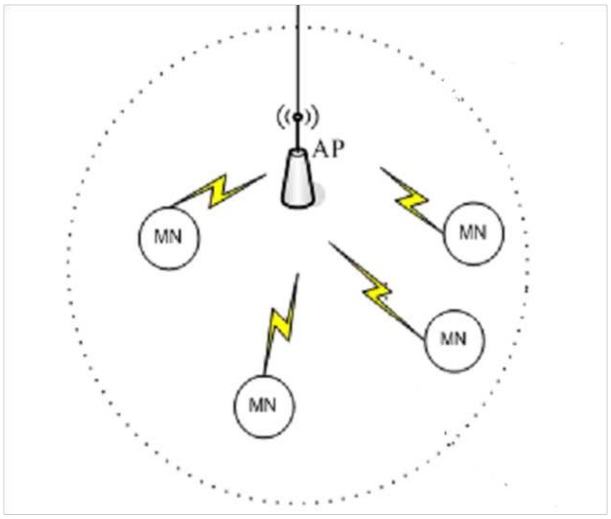


Figure 5.3: Experimental Environment

In our experiments, every device begins with the empty shared document as presented in Figure 5.4 and generates 250 operations. Therefore, every device will execute 1000 operations consisting of 250 local operations generated by it and 750 remote operations generated by all other devices. Figure 5.5 shows a consistent state of all devices after experiment. In every experiment, we get 30 measurements of each sample.

Figure 5.4: The Empty State in the Begining of Experiment

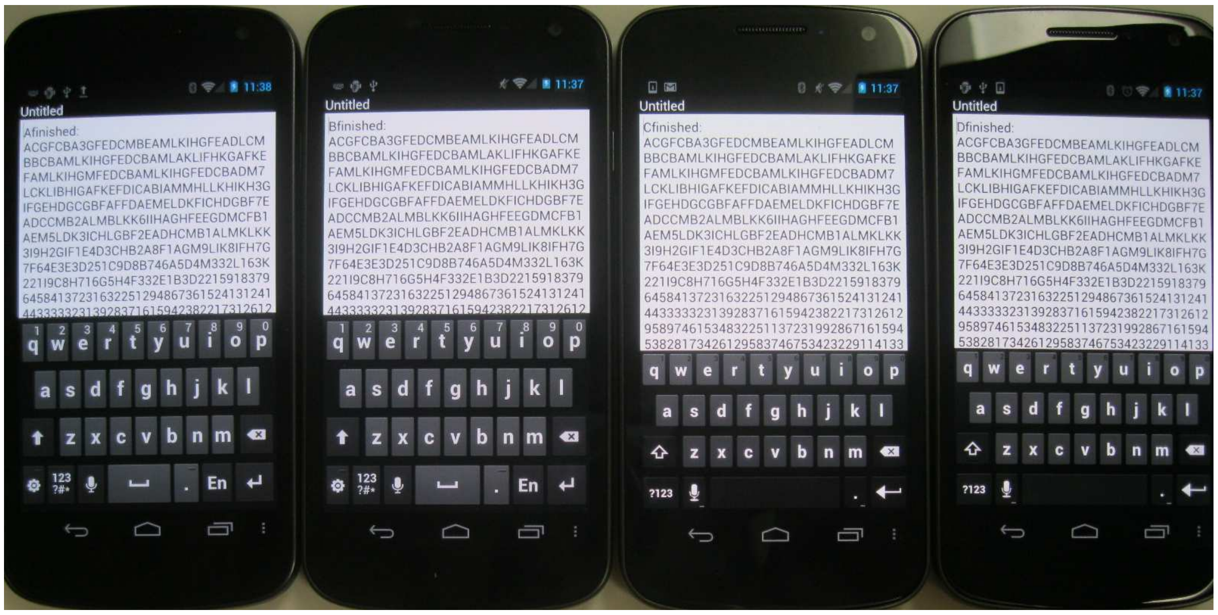


Figure 5.5: Consistent State in the End of Experiment

5.1.1 Operation Time

Firstly, we consider average operation time for a device to finish all operations as shown in Figure 5.6 (operation time = time for executing 250 local operations + time for executing 750 remote operations) in 4 cases:

- Operational Transformation
- The best case of TreeDoc
- The worst case of TreeDoc without rebalancing
- The average case of TreeDoc where editing positions are generated randomly in uniform distribution without rebalancing

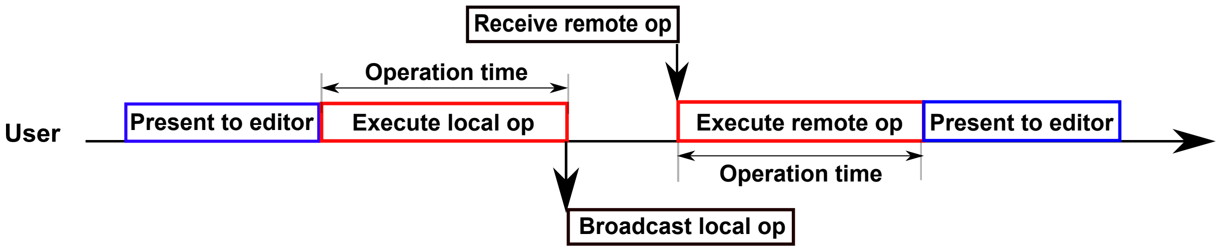


Figure 5.6: Operation Time

In this experiment, we manipulate the number of concurrent operations in the system by increasing operation interval, that is the period between two consecutive operations, in order to investigate the trend of operation time when the number of concurrent operations increases. Because every operation carries the state vector to reflect the status of the user generated it, we base on the state vector of every operations to compute the number of current operations in the system.

Figure 5.7 shows the result of this experiment. The number of concurrent operations decreases as operation interval increases, which results in the decrease of operation time of Operational Transformation, while operation time of TreeDoc is not changed. That is because for Operational Transformation, the more the number of concurrent operations is, the more the transformation procedure is executed, while for TreeDoc, concurrent operations can commute with each other without control. Especially, when operation interval is close to or greater than network delay, the number of current operations is fixed by 4 operations and operation time of Operational Transformation is 1553.3 ms. In the best case, operation time of TreeDoc is 457.57 ms, that is always better than operation time of Operational Transformation. However, in the worst case, operation time of TreeDoc is 3571.63 ms, that is only better than operation time of Operation Transformation if the number of current operations is greater than 10 operations.

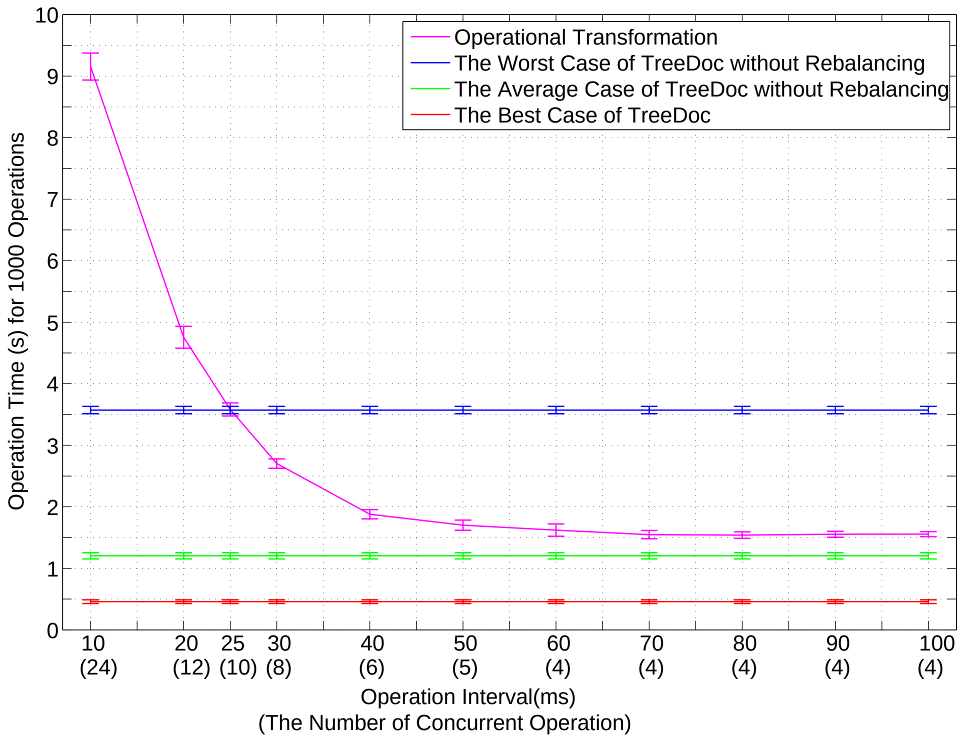


Figure 5.7: The number of Concurrent Opertions vs. Operation Time without Rebalancing

5.1.2 Rebalancing Time

With periodically executing rebalancing, the height of tree decreases, which results in the decrease of operation time in the worst case of TreeDoc. Next, we consider rebalancing time, operation time (2 cores, 2 nebulars) as depicted in Figure 5.8 in the worst case of TreeDoc when manipulating rebalancing period, that is the number of operations for a repeating rebalancing.

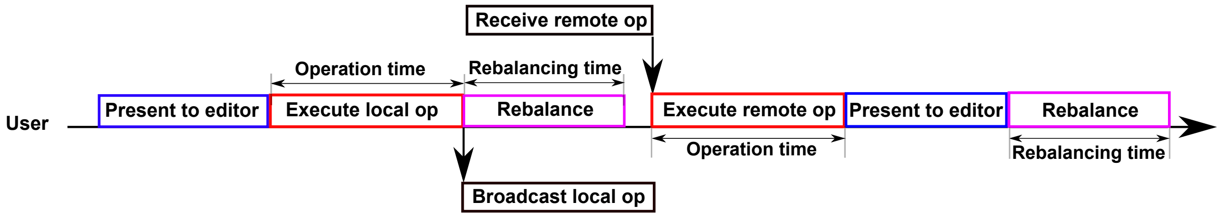


Figure 5.8: Rebalancing Time

The more the execution of rebalancing is, the more operation time decreases (Figure 5.9). However, the more the execution of rebalancing is, the more rebalancing time increases (Figure 5.9). With rebalancing period of 200 operations, total time consisting of rebalancing time and operation time is minimal (Figure 5.9) and operation time in the worst case of TreeDoc is reduced by 1100.26 ms that is always better than operation time of Operational Transformation (Figure 5.10). Therefore, 200 operations is the best value of rebalancing period in our experiments.

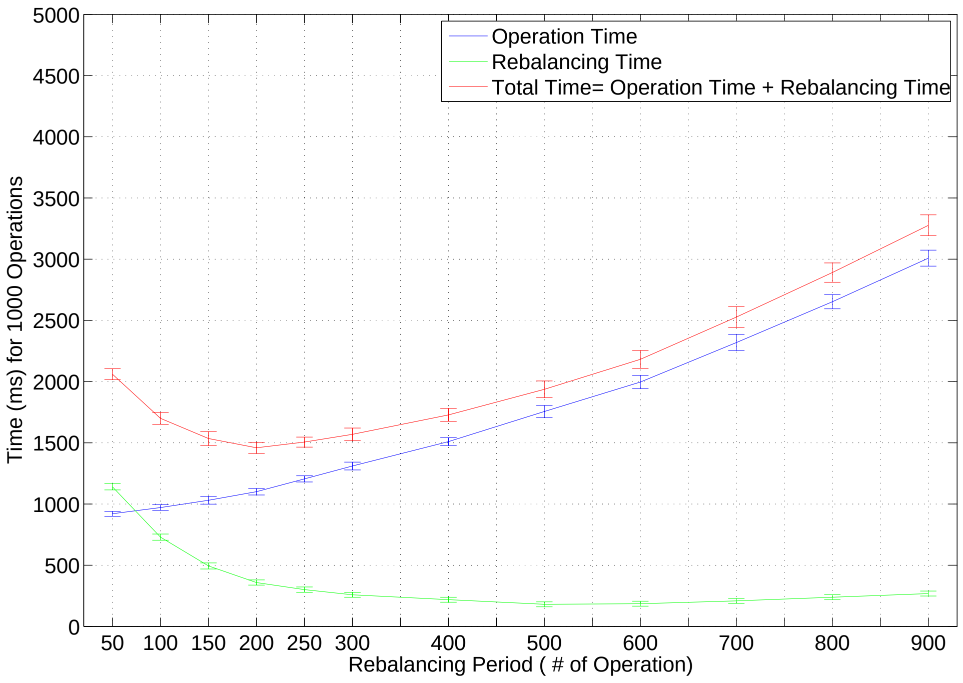


Figure 5.9: The Worst Case of TreeDoc vs. Rebalancing

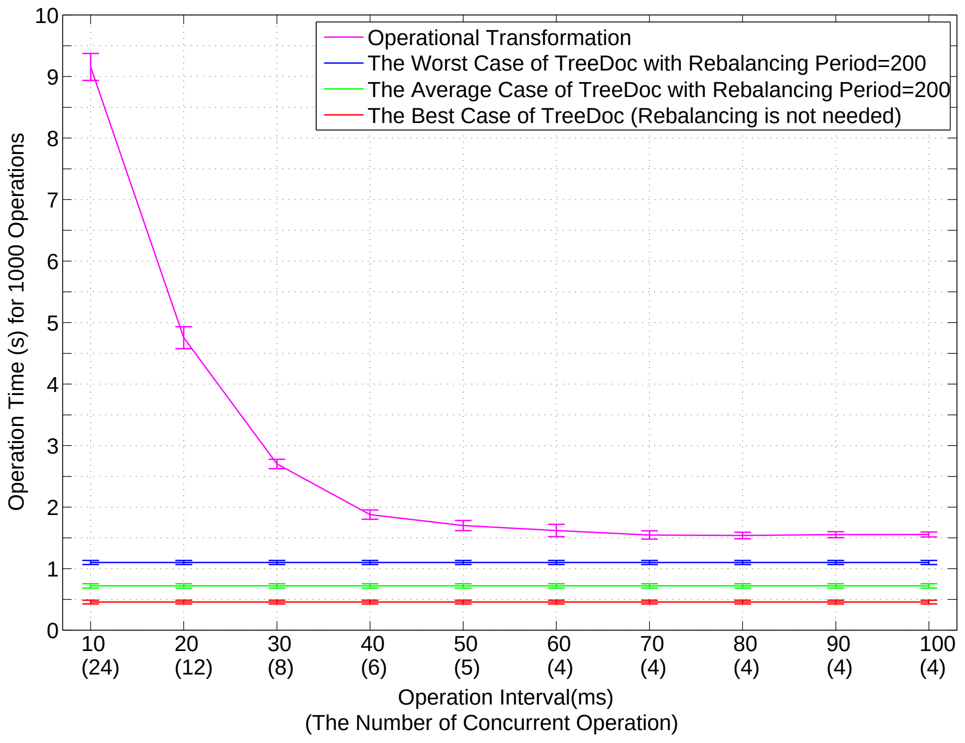


Figure 5.10: Operation Time vs. The number of Concurrent Opertions with Rebalancing

5.1.3 Operation Latency

Next, we consider latency of every operation, that is the interval from the time when a user edits a character on its editor to time when other user takes effect of that character like Figure 5.11.

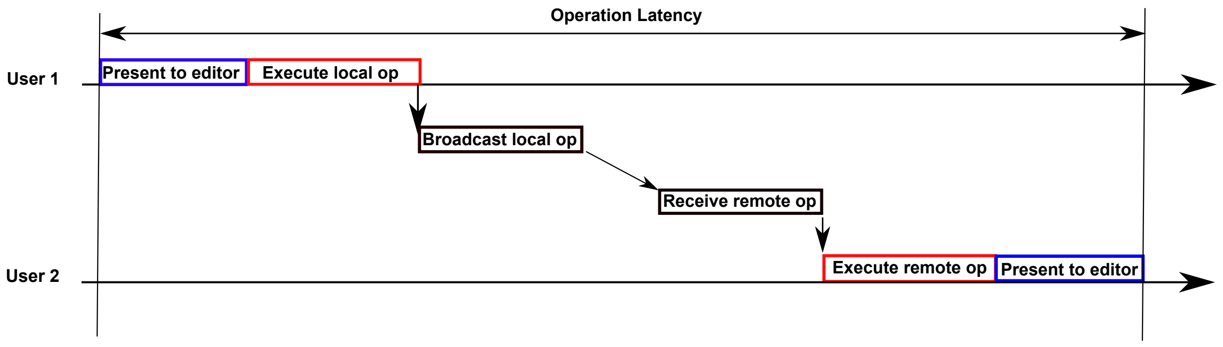


Figure 5.11: Operation Latency

In the worst case of TreeDoc, latency of every operation increases with operation ID because the height of the node corresponding with every operation increases steadily (operation i at height $h = i - 1$, $i = 1...n$). However, with rebalancing period of 200 operations, rebalancing is executed at operation ID:200; 400; 600; 800, which leads the height of nodes corresponding with operation ID: 201;401;601;801 to be reduced by 7;8;9;9, respectively and the decrease of nodes corresponding with operations after rebalancing. Therefore, latency falls dramatically after rebalancing as shown in Figure 5.12, and the maximum of operation latency is 146.4 ms.

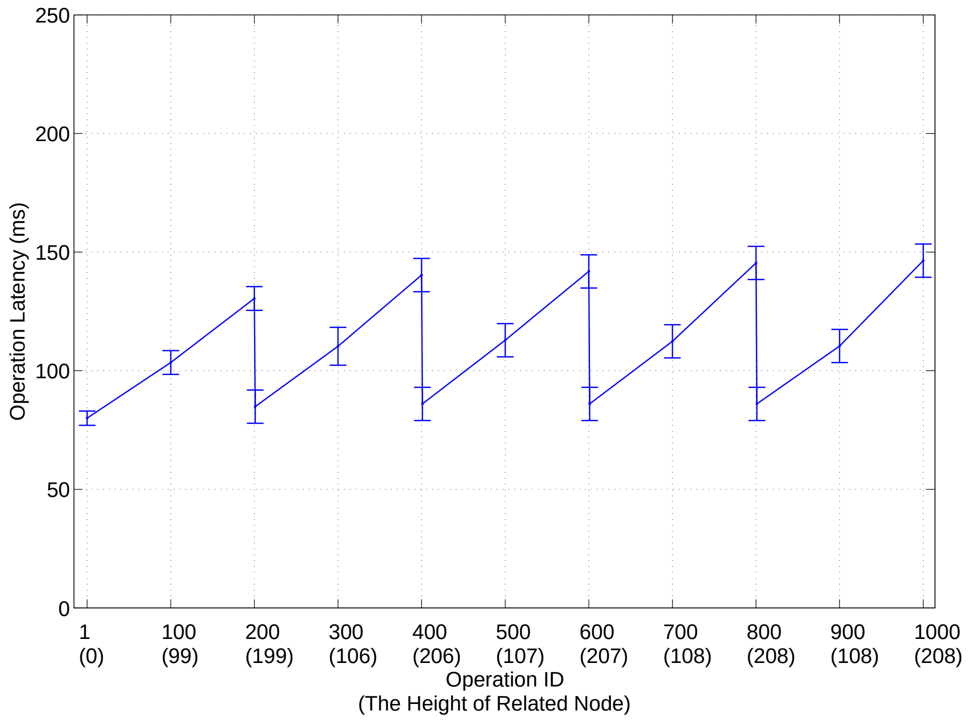


Figure 5.12: The Worst Case of TreeDoc vs. Operation Latency

5.1.4 Time Usage

Finally, Figure 5.14 shows time usage of a user during the collaboration session with rebalancing period of 200 operations and operation interval of 100 ms. Completion time is average time for a user to finish the collaboration session. It includes the following components as presented in Figure 5.13:

- Present time is time for updating editor
- Operation time is time for executing local/remote operation
- Rebalancing time is time for executing rebalancing
- Idle time

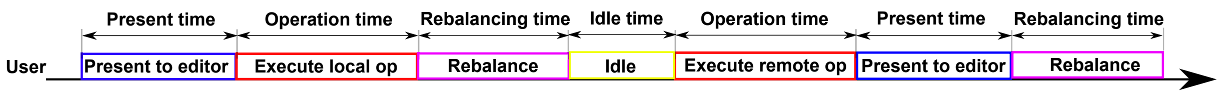


Figure 5.13: Completion Time

In the best case of TreeDoc, the tree is always balanced during the collaboration session, so it does not need rebalancing. Furthermore, in this case, operation time is minimal because total height of all nodes in the tree is minimal. Therefore, process time of this case only consists of present time and operation time and is relatively small at 3.3% (operation time=1.8%; present time=1.5%; rebalancing time=0%) of completion time.

Conversely, in the worst case of TreeDoc, the tree has to be periodically rebalanced to reduce operation time. Hence, process time of this case consists of present time, operation time, and rebalancing time. In this case, process time is the largest of three cases of TreeDoc and up to 7.3% (operation time=4.4%; present time=1.5%; rebalancing time=1.4%) of completion time.

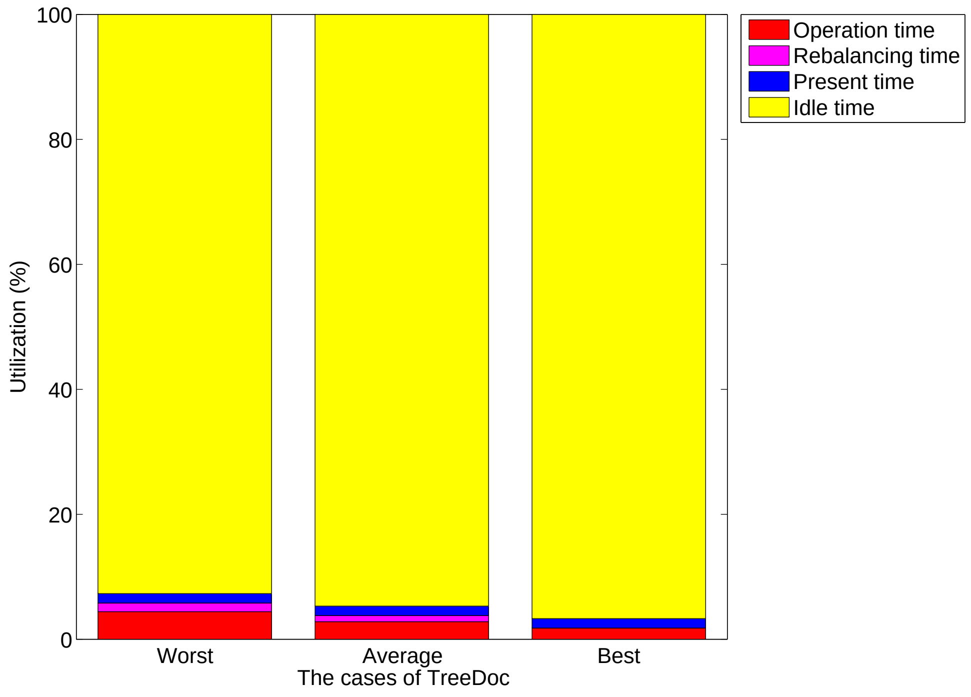


Figure 5.14: Time Usage of a User during the Collaboration Session

Chapter 6

Fault Tolerance

6.1 Disconnection

In MANETs, mobile device frequently gets disconnected from other devices. In the scope of this thesis, we concentrate on the disconnection in the case when mobile device goes too far out of the transmission range of other devices. When a site is disconnected, it can not receive and broadcast messages from/to sites that it is disconnected. Consequently, in the collaboration session, not only disconnected user misses messages but also other users miss messages generated by disconnected user during disconnection period.

Because of disconnection event, every user in the collaboration session has its own state which may be different from the state of other users. Therefore, when a user joins the collaboration session, it may miss some document operations to reach the last state of the collaboration session. Furthermore, when a user joins the collaboration session, if a current user in the collaboration session is disconnected from the joining user, it will miss the joining message broadcast by the joining user and document operations generated by the joining user before joining the collaboration session.

Furthermore, collaboration session has to ensure causality preservation property. Consequently, a message may be blocked at a user, if that user misses messages causally preceding it.

In order to tolerate disconnection event, the system has to solve missing messages (operation document, leaving, joining). Concretely, the system must satisfy the following property:

- If a user S_i generates a message m (operation document, leaving, joining), eventually, all users receive m either directly from S_i or indirectly from other users.

6.2 Related Works

One approach is that all users have to agree on the same state when a user joins the collaboration session. Therefore, joining user is able to get the last state of the share document easily and all current users in the collaboration session do not miss joining message and document operation messages generated by joining user before joining the collaboration session.

Concretely, in Ycab [21], all users agree on the same state by running commitment protocol in which joining user is the coordinator. Concretely, when a user joins the collaboration session, it broadcasts joining message and its current state to all users of the collaboration session and waits for the replies about their states. If all users have the same state, joining users broadcasts the message to notify all users of processing joining messages and adding joining user to their participant list.

Contrary to Ycab, in CoWord [22], all users agree on the same state by broadcasting synchronization message to push all operations to arrive, be executed at their destinations, and temporarily block users from generating new operations, which is called quiescence state. If all users reach quiescence state, they have identical document state. Therefore, in order to join the collaboration session successfully, joining user only needs to get the document and state vector of a current user.

This approach requires all users to be well-connected together during the process of joining session to ensure that all messages arrive at their destinations. Therefore, the agreement is executed correctly, quickly and the collaboration session is not blocked a long time. However in MANETs, above condition is not always guaranteed. In this chapter, we will provide a solution to solve missing message problem for the process of joining, leaving, editing in a collaboration session.

6.3 Tolerating Missing Messages

Each message in a collaboration session consisting of joining message, document operation message, leaving message has to carry a state vector to reflect the status of the user generated it. For each user, the first message is a joining message, the next messages are document operation messages, and the last message is a leaving message. In the collaboration session, every user has to ensure the execution of those messages in causal order by basing on their state vectors to detect which of the message is causally ready.

Consequently, when a message m generated by user S_i arrives user S_j, S_j compares the state vector of m, which shows the state of S_i when it generated m, and the state vector of S_j, which shows the state of S_j when m arrives S_j, in order to detect message k that causally precedes m and S_j misses, and then, S_j sends a request to ask S_i to resend missing message k.

The following sections discuss how user S_j detects missing message by comparing its state vector and the state vector of received message m generated by user S_i in two cases. After that, we specify two error cases in which the comparison between state vectors cannot detect missing message and provide solutions for those cases.

6.3.1 Detection in the Normal Case

In this case, S_j detects missed messages generated by user S_t whose related component appears in both the state vector of S_j, noted by V_{S_j}, and the state vector of message m, noted by V_m. As described in the section 2.3, $V_{S_j}[S_t]$ indicates the number of messages generated by S_t that has already been executed by S_j, while $V_m[S_t]$ indicates the number of messages generated by S_t that has already been executed by user S_i when it generates m. Therefore, if $V_m[S_t]$ is greater than $V_{S_j}[S_t]$, S_j realizes that it misses messages k=< $V_m[S_t]$-1,$V_m[S_t]$-2,....,$V_{S_j}[S_t]$ > generated by S_t.

Figure 6.1 shows the example in which user S_j is disconnected from S_t, S_i. Therefore, S_j can not receive $op_{S_{t_0}}$, $op_{S_{i_0}}$ generated by S_t, S_i, respectively during disconnection period of S_j. When S_j reconnects, it receives $op_{S_{i_1}}$ generated by S_i. By basing on state vector of $op_{S_{i_1}}$, that is state vector of S_i when it generates $op_{S_{i_1}}$, S_j knows that S_i has already executed one operation generated by S_t because $V_{S_i}[S_t] = 1$ and one operation generated by S_i because $V_{S_i}[S_i] = 1$, while S_j has yet to execute any operations generated by S_t and S_j. Consequently, S_j detects missing messages consisting of message 0 generated by S_t ($op_{S_{t_0}}$), message 0 generated by S_i ($op_{S_{i_0}}$), and sends a request to ask S_i to resend those messages.

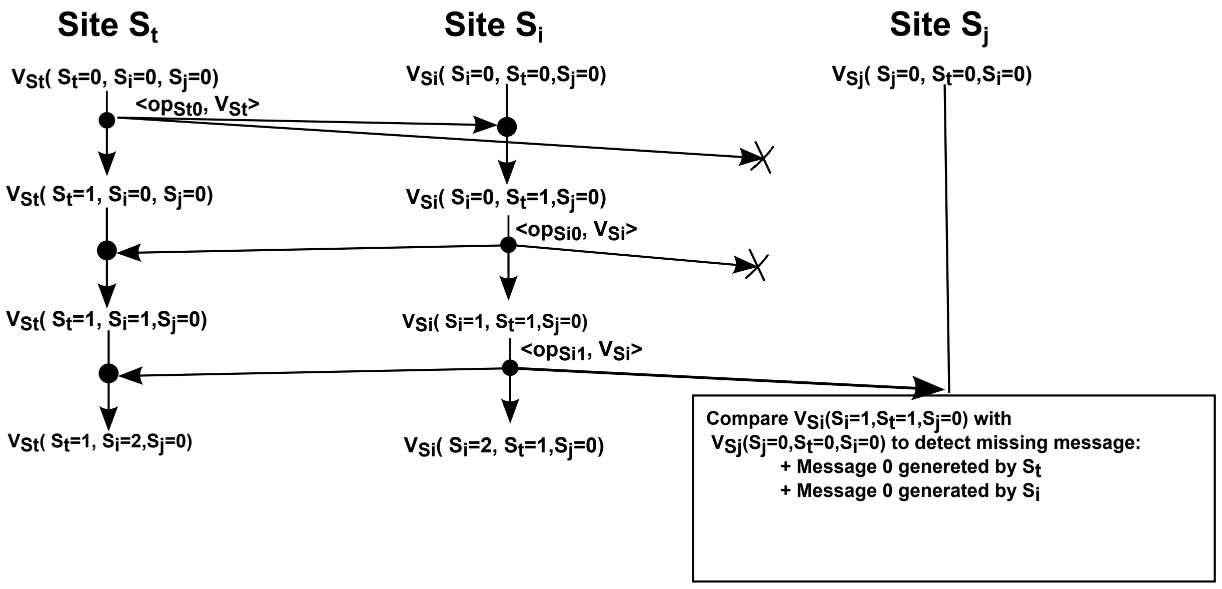

Site S_t
Site S_i
Site S_j
V_{St}(S_t=0, S_i=0, S_j=0)
V_{Si}(S_i=0, S_t=0,S_j=0)
V_{Sj}(S_j=0, S_t=0,S_i=0)
<op_{St0}, V_{St}>
V_{St}(S_t=1, S_i=0, S_j=0)
V_{Si}(S_i=0, S_t=1,S_j=0)
<op_{Si0}, V_{Si}>
V_{St}(S_t=1, S_i=1,S_j=0)
V_{Si}(S_i=1, S_t=1,S_j=0)
<op_{Si1}, V_{Si}>
V_{St}(S_t=1, S_i=2,S_j=0)
V_{Si}(S_i=2, S_t=1,S_j=0)
Compare V_{Si}(S_i=1,S_t=1,S_j=0) with
V_{Sj}(S_j=0,S_t=0,S_i=0) to detect missing message:
+ Message 0 genereted by S_t
+ Message 0 generated by S_i

Figure 6.1: Detection in the Normal Case

6.3.2 Detection in the Special Case: Missing Joining or Leaving Messages

In this case, S_j detects missing messages generated by S_t whose related component appears in state vector of S_j and disappears in state vector of message m. This case happens because two reasons:

- The first reason is that S_i executed joining message of S_t before generating m, while S_j missed joining message of S_t. In this case, S_j misses all of message generated by

S_t: $< V_m[S_t]$-1,$V_m[S_t]$-2,...,0 $>$. This context is shown in the example in Figure 6.2. In this example, when receiving $op_{S_{i_0}}$ generated by S_i, S_j knows that it misses all messages generated by S_t consisting of joining message and $op_{S_{t_1}}$ of S_t.

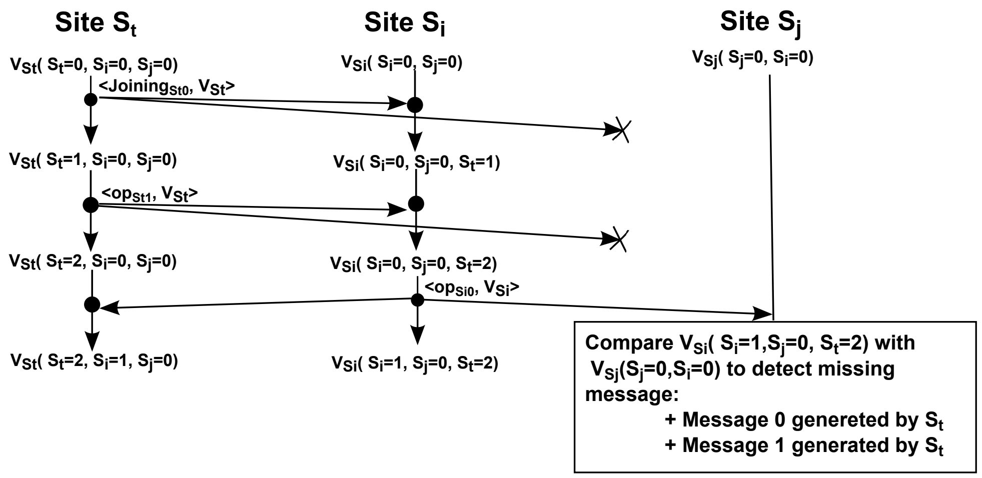

Site S_t
Site S_i
Site S_j
V_{St}(S_t=0, S_i=0, S_j=0)
V_{Si}(S_i=0, S_j=0)
V_{Sj}(S_j=0, S_i=0)
<Joining$_{St0}$, V_{St}>
V_{St}(S_t=1, S_i=0, S_j=0)
V_{Si}(S_i=0, S_j=0, S_t=1)
<op$_{St1}$, V_{St}>
V_{St}(S_t=2, S_i=0, S_j=0)
V_{Si}(S_i=0, S_j=0, S_t=2)
<op$_{Si0}$, V_{Si}>
V_{St}(S_t=2, S_i=1, S_j=0)
V_{Si}(S_i=1, S_j=0, S_t=2)
Compare V_{Si}(S_i=1,S_j=0, S_t=2) with V_{Sj}(S_j=0,S_i=0) to detect missing message:
+ Message 0 genereted by S_t
+ Message 1 generated by S_t

Figure 6.2: Detection in the Special Case: Missing Joining Messages

- The second reason is that S_i missed leaving message of S_t before generating m, while S_j executed leaving message of S_t. In this case, S_j does not miss any messages generated by S_t. Consider the example in Figure 6.3, when receiving $op_{S_{i_0}}$ generated by S_i, S_j knows that it does not miss any messages generated by S_t because it executed leaving message of S_t.

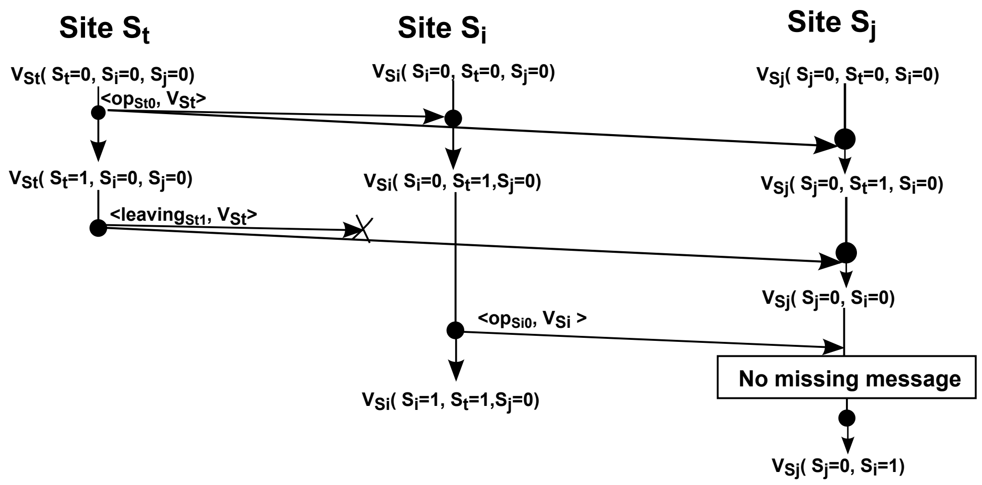

Site S_t
Site S_i
Site S_j
V_{St}(S_t=0, S_i=0, S_j=0)
V_{Si}(S_i=0, S_t=0, S_j=0)
V_{Sj}(S_j=0, S_t=0, S_i=0)
<op$_{St0}$, V_{St}>
V_{St}(S_t=1, S_i=0, S_j=0)
V_{Si}(S_i=0, S_t=1,S_j=0)
V_{Sj}(S_j=0, S_t=1, S_i=0)
<leaving$_{St1}$, V_{St}>
V_{Sj}(S_j=0, S_i=0)
<op$_{Si0}$, V_{Si} >
No missing message
V_{Si}(S_i=1, S_t=1,S_j=0)
V_{Sj}(S_j=0, S_i=1)

Figure 6.3: Detection in the Special Case: Missing Leaving Messages

Specification 8 describes the procedure for detecting missing messages.

Specification 8 Detection of Missing Messages

S_j with state vector V_{S_j} detects missing message after receiving message m with state vector V_m generated by S_i.

```
1: procedure DectectMissingMessages($m$,$V_m$,$S_i$)
2:     for all $S_t \in V_m$ do
3:         if $S_t \in V_{S_j}$ then
4:             if $V_{S_j}[S_t] < V_m[S_t]$ then
5:                 Send a request to $S_i$ for asking missing messages $k$=$< V_m[S_t]-1, V_m[S_t]-2, ..., V_{S_j}[S_t] >$;
6:             end if
7:         else
8:             if $S_j$ already executed leaving message of $S_t$ then
9:                 No missing messages;
10:            else
11:                Send a request to $S_i$ for asking missing messages $k$=$< V_m[S_t]-1, V_m[S_t]-2, ..., 0 >$;
12:            end if
13:        end if
14:    end for
15: end procedure
```

6.3.3 Fault Case 1: Missing All Messages from a User

Consider the context in Figure 6.4 where user S_j misses all messages (joining, operation, leaving) generated by S_t from the time when S_t joins the collaboration to the time when S_t leaves the collaboration, while user S_i receives and executes all of those messages. In this context, when message $op_{S_{i_0}}$ generated by S_i arrives S_j, S_j can not detect missing messages generated by S_t by comparing its state vector with state vector of S_i because now, the component associated with S_t is no longer in state vector of S_i after S_i executed leaving message of user S_t.

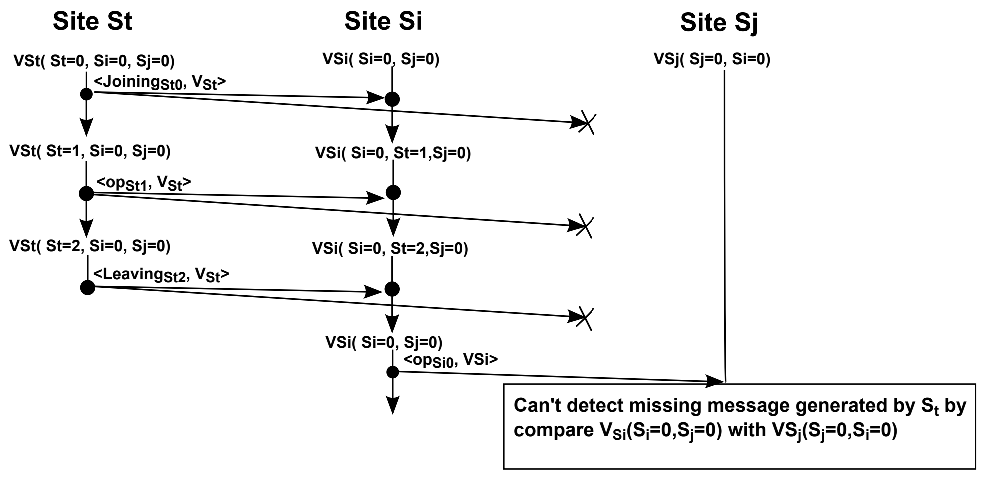


Figure 6.4: Fault Case when Missing All Messages from a User

In order to take into account this context, S_i has to delay executing leaving message until it generates a message. Concretely, in Figure 6.5, after receiving leaving message of S_t, S_i does not execute that leaving message immediately. After S_i generates message $op_{S_{i_0}}$ and sends it to all other users, S_i executes leaving message of S_t by removing S_t from its state vector. Consequently, when S_j receives message $op_{S_{i_0}}$, it can detect missing messages generated by S_t because the component associated with S_t is still in the state vector of $op_{S_{i_0}}$. In the worst case where S_j misses message $op_{S_{i_0}}$, although the next messages generated S_i do not help S_j to detect missing messages generated by S_t, they help S_j to detect missing message $op_{S_{i_0}}$ and after that, S_j bases on state vector of $op_{S_{i_0}}$ to detect missing messages generated by S_t.

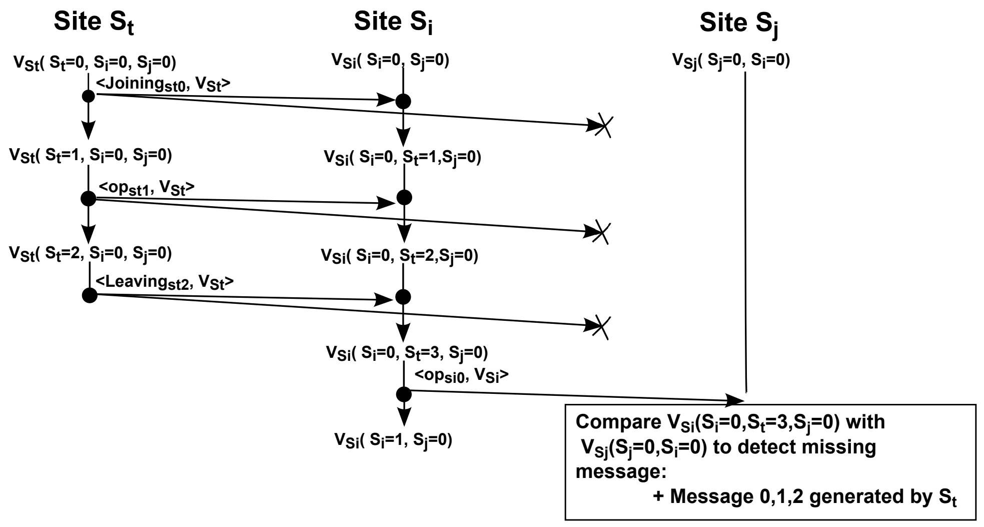

Site S$_t$
Site S$_i$
Site S$_j$
V$_{St}$(S$_t$=0, S$_i$=0, S$_j$=0)
V$_{Si}$(S$_i$=0, S$_j$=0)
V$_{Sj}$(S$_j$=0, S$_i$=0)
<Joining$_{st0}$, V$_{St}$>
V$_{St}$(S$_t$=1, S$_i$=0, S$_j$=0)
V$_{Si}$(S$_i$=0, S$_t$=1,S$_j$=0)
<op$_{st1}$, V$_{St}$>
V$_{St}$(S$_t$=2, S$_i$=0, S$_j$=0)
V$_{Si}$(S$_i$=0, S$_t$=2,S$_j$=0)
<Leaving$_{st2}$, V$_{St}$>
V$_{Si}$(S$_i$=0, S$_t$=3, S$_j$=0)
<op$_{si0}$, V$_{Si}$>
V$_{Si}$(S$_i$=1, S$_j$=0)
Compare V$_{Si}$(S$_i$=0,S$_t$=3,S$_j$=0) with V$_{Sj}$(S$_j$=0,S$_i$=0) to detect missing message:
+ Message 0,1,2 generated by S$_t$

Figure 6.5: Solution for Fault Case 1

6.3.4 Fault Case 2: Silent User

In the context that if requests for asking to resend messages are lost, missing messages will not be resent. Especially, if users fall to silent status in which they do not generate any messages, other users can not detect missing messages.

Our solution is that if a user does not generate any messages for a period T, it has to broadcast a state notification containing its state vector to all other users to detect missing messages.

6.4 Evaluation

In this section, we continue considering the experiment as presented in the Chapter 2. In order for a message which is operation document, leaving or joining to arrive its destination under disconnection event, the system may send many packets including original or resent message, requests for asking to resend message, state notifications for detecting messing message. Therefore, in our experiment, when a user generates a message, for that message to arrive at 3 other users, in the case of no disconnection, it needs to send 3 packets, each of which is for every user, while under disconnection event, it has to send more than 3 packets.

In this experiment, we manipulate loss probability, which is the probability for a packet not arriving at its destination. We simulate packet loss by randomly generating a number in uniform distribution and, after that, we base on the generated number and the loss probability to determine whether a packet is lost. For example, with loss probability of 50%, before a user sends a packet, it generates a random number in the range of 1 to 100. If the random number is less than or equal to 50, the user makes the packet lost by not sending this packet to its destination.

We investigate the following metrics in the worst case of TreeDoc (2 core, 2 nebular) with rebalancing period of 200 operations, operation interval of 100 ms, and period T for broadcasting state notification of 500 ms:

- The average number of packets sends by a user for all other users to receive all messages (the number of packets generated)
- Average completion time for a user to finish the collaboration session (completion time)
- Average time for a user to detecting its missing messages (detection time)

If loss probability is equal to 0%, the number of packets is equal to the number of messages, detection time is 0, and completion time is 27.3 s. The more loss probability is, the more the number of missing messages is, so a user has to send more packets, and take more detection time in order for all users to get their missing messages, which results in the increase of the number of packets, detection time, and completion time as shown in Figure 6.6, Figure 6.7, Figure 6.8. Furthermore, the increase of detection time leads to the increase of operation latency which is delay between time when a user presses a character and time when this character appears on the screen of other user. In practice, there are some studies about acceptable latency for users. For multiplayer games, latency affects significantly the play and result. The acceptability of more or less latency for a game depends on its type. For example, with First Person Shooters (FPS) which is a game for shooting, Claypool et al. in [24] showed that s the accuracy of shooting is greatly decreased with even modest (75-100 ms) amounts of latency, while RTS (real time strategy) games has a fairly acceptable performance even at latency of 500 ms [25]. Obviously, acceptable latency in collaborative text editing is higher than acceptable latency in game. When loss probability reaches 75%, the number of sent messages reaches is 11021 packets (955.8% of the number of messages needed to be sent), detection time is 73.7 s and completion time is 2559.5 s. The difference between the completion time in the case of the loss probability of 0% and that of 75%, which corresponds to 2532.2s (2559.5 − 27.3), is the period that a user detects the missing messages and waits for these messages being resent. In other words, that is the increasing of total latency of all operations in those two cases. Because every user in our experiment has to execute 1000 operations, average latency for every operation is greater than 2.5322 s calculated by 2532.2 s/1000, which is very far larger than operation interval (100 ms). Especially, when loss probability is greater than 75%, the collaboration session cannot terminate within 2 hours.

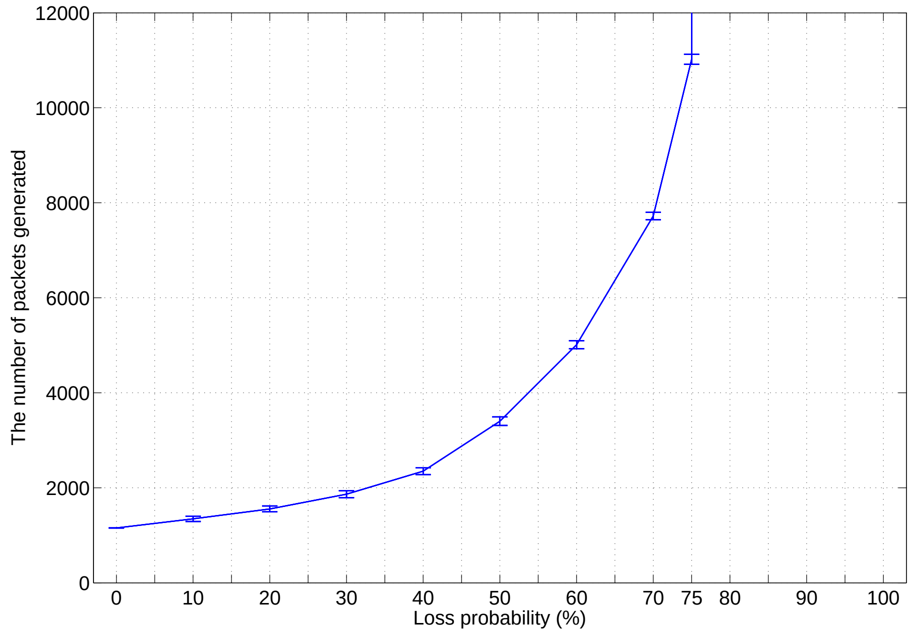


Figure 6.6: The number of Packets vs. Loss Probability

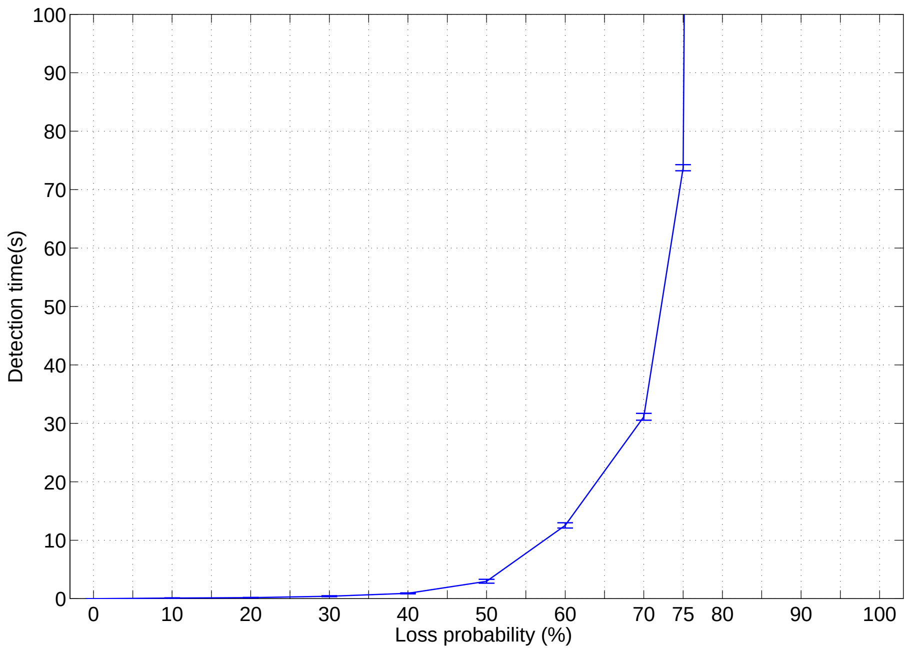


Figure 6.7: Detection Time vs. Loss Probability

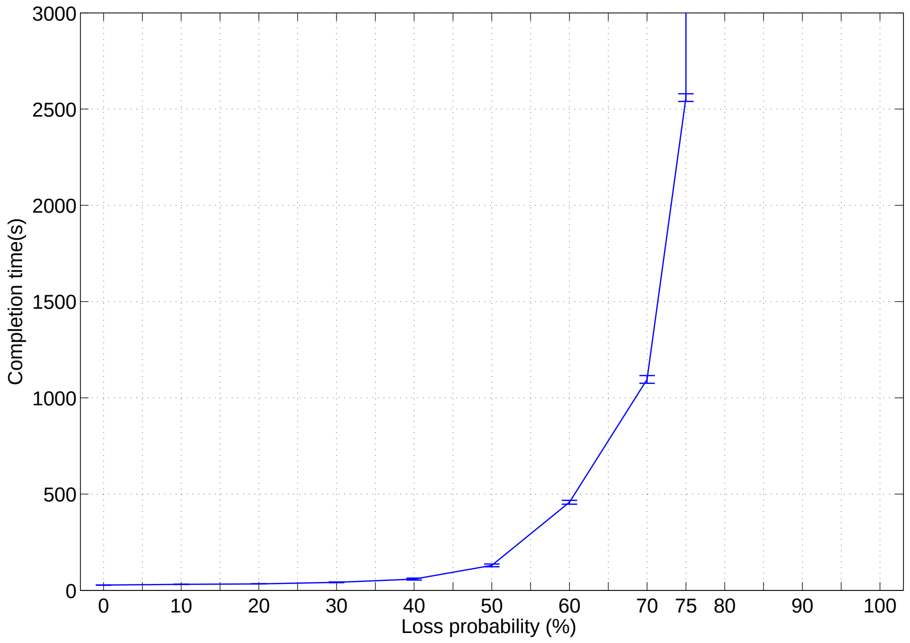


Figure 6.8: Completion Time vs. Loss Probability

Chapter 7

Application

7.1 Editing Function

In this section, we show the photos of editing functions in our application. Every photo demonstrates a sequence of consecutive actions of a function. Our application provides the following editing functions for user:

- File (Figure 7.1)
  - New: Create new document (Figure 7.2).
  - Open: Open a existing document (Figure 7.3).
  - Save/Save As: Save a document (Figure 7.4).

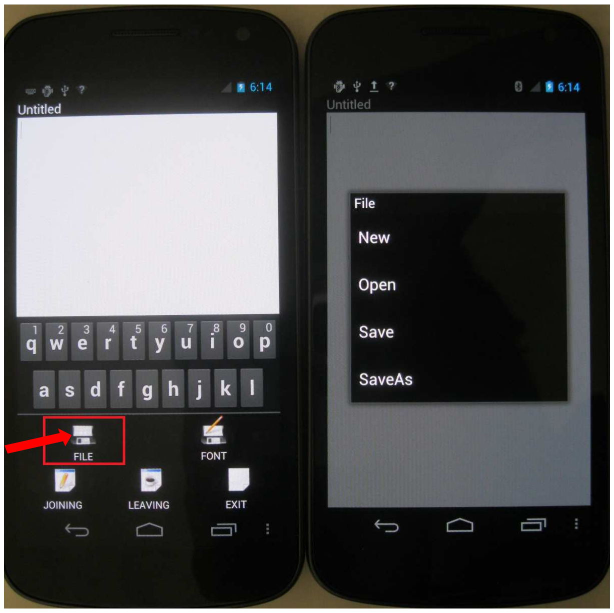


Figure 7.1: Menu File

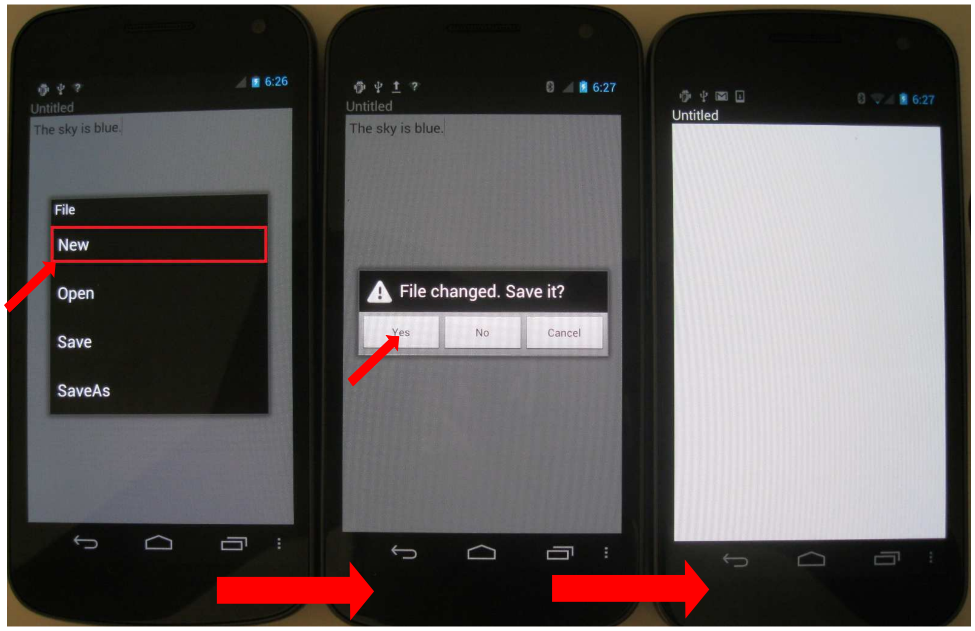


Figure 7.2: New File (1. “New” in Menu File; 2. “No” in Dialog of Save File)

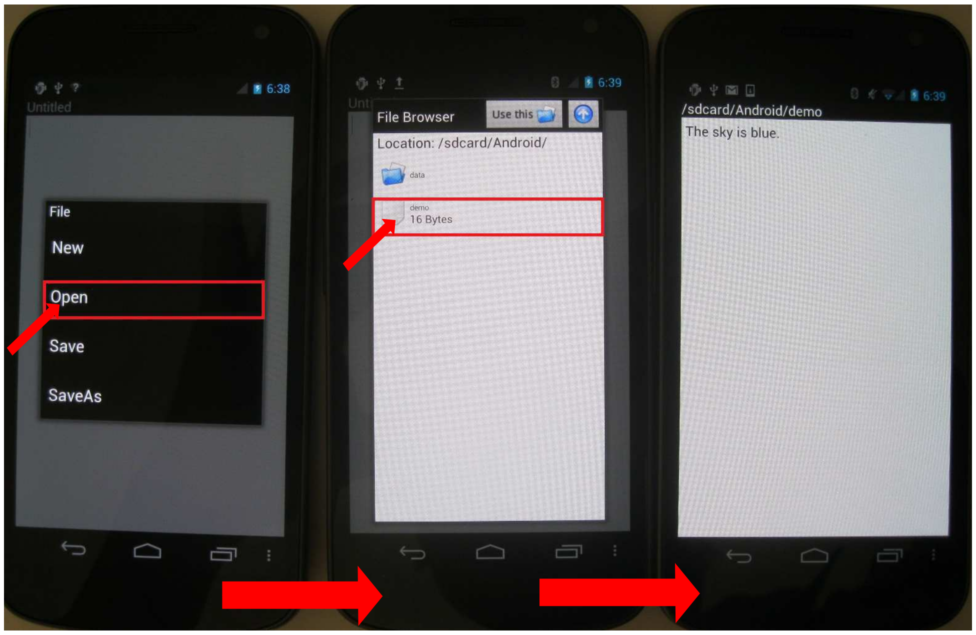


Figure 7.3: Open File (1. “Open” in Menu File; 2. Select file “Demo” in File Browser)

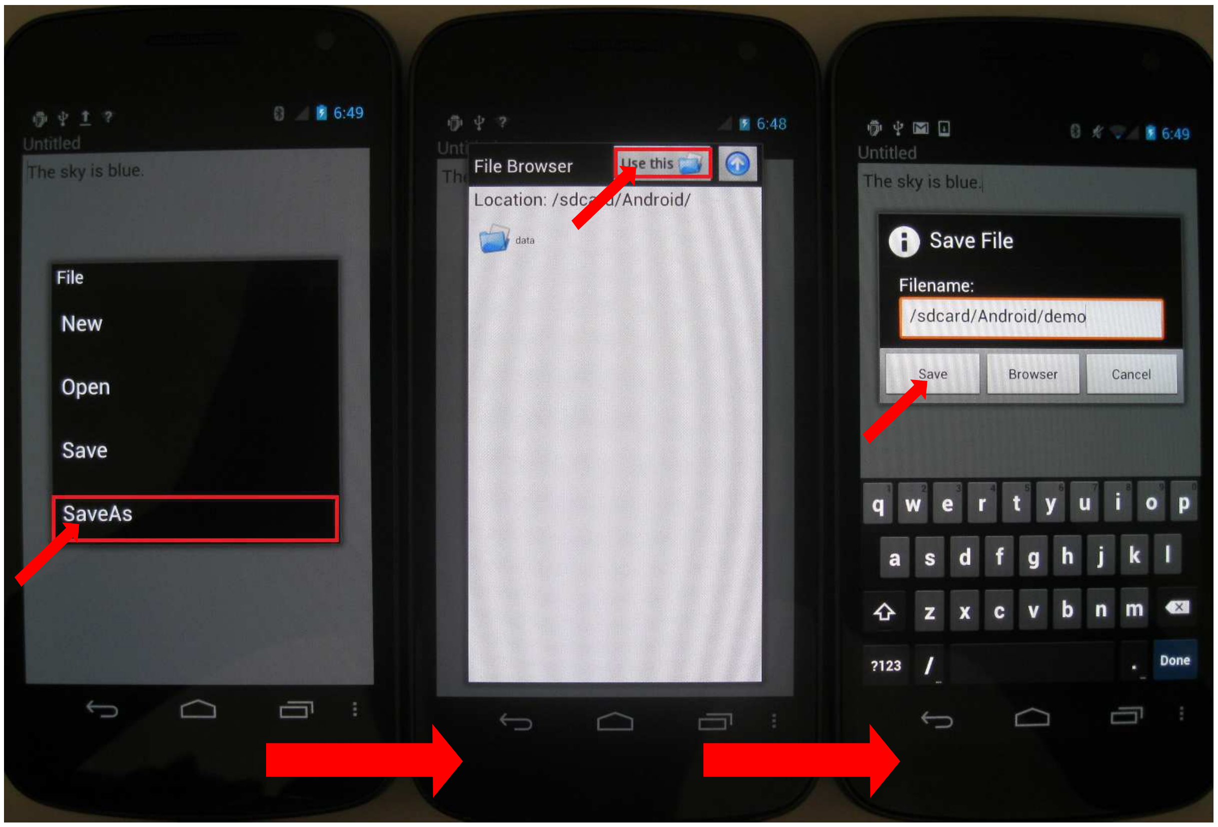


Figure 7.4: Save as (1. "SaveAs" in Menu File; 2. Select Folder in File Browser; 3. Save in Dialog of Save File)

- Font (Figure 7.5)
  - Font Style: Change font style (Bold/Italic/Underline/Quote/Strike-through) of selected text (Figure 7.6).
  - Font Type: Change font type (Normal/Monospace/Serif/Sans-Serif) of selected text (Figure 7.7).
  - Text Color: Change text color of selected text (Figure 7.8).
  - Background Color: Change background color of selected text (Figure 7.9).

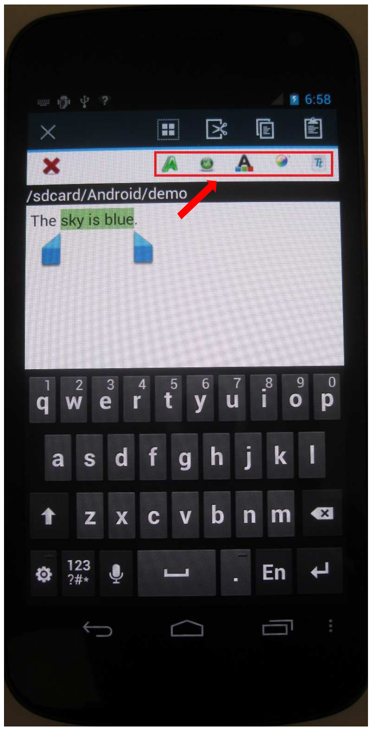


Figure 7.5: Menu Font

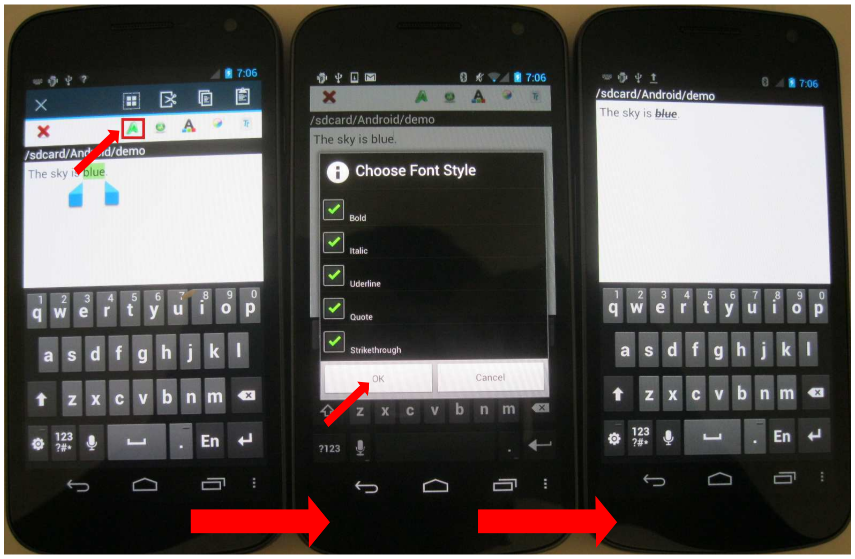


Figure 7.6: Font Style (1. Select text; 2. Select font style)

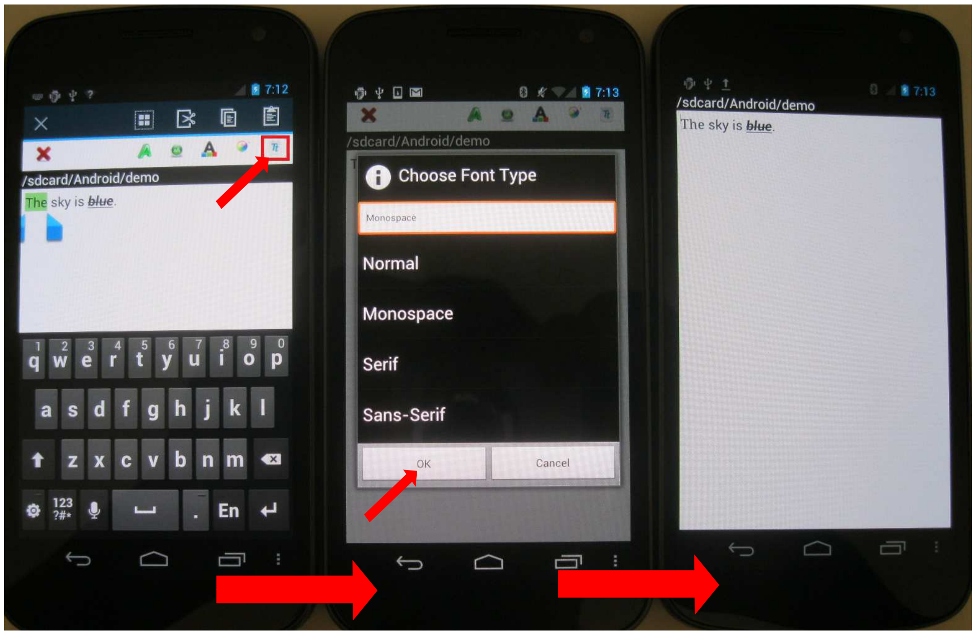


Figure 7.7: Font Type (1. Select text; 2. Select font type)

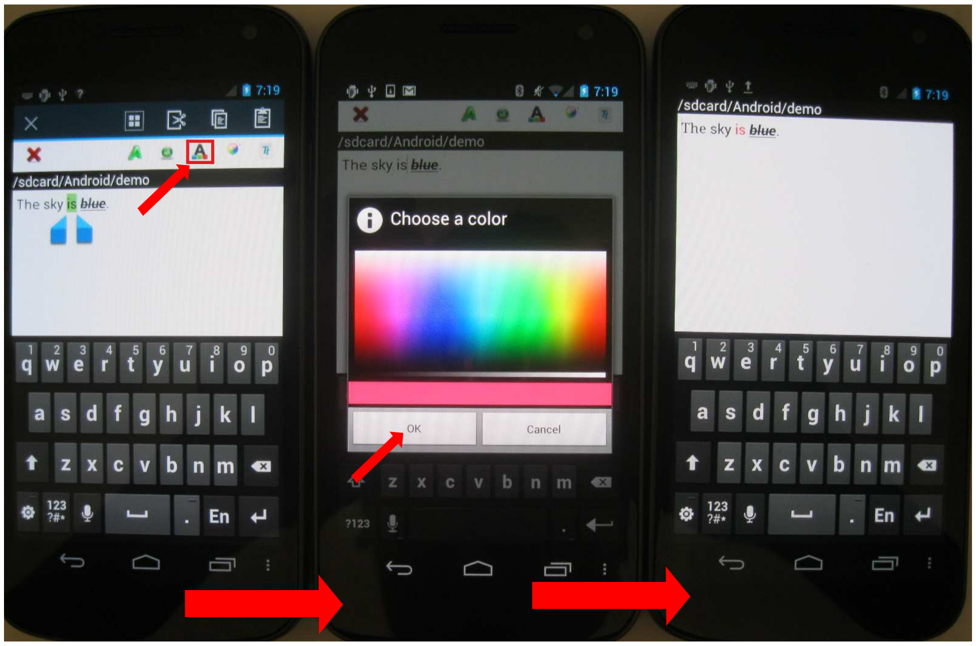


Figure 7.8: Text Color (1. Select text; 2. Select text color)

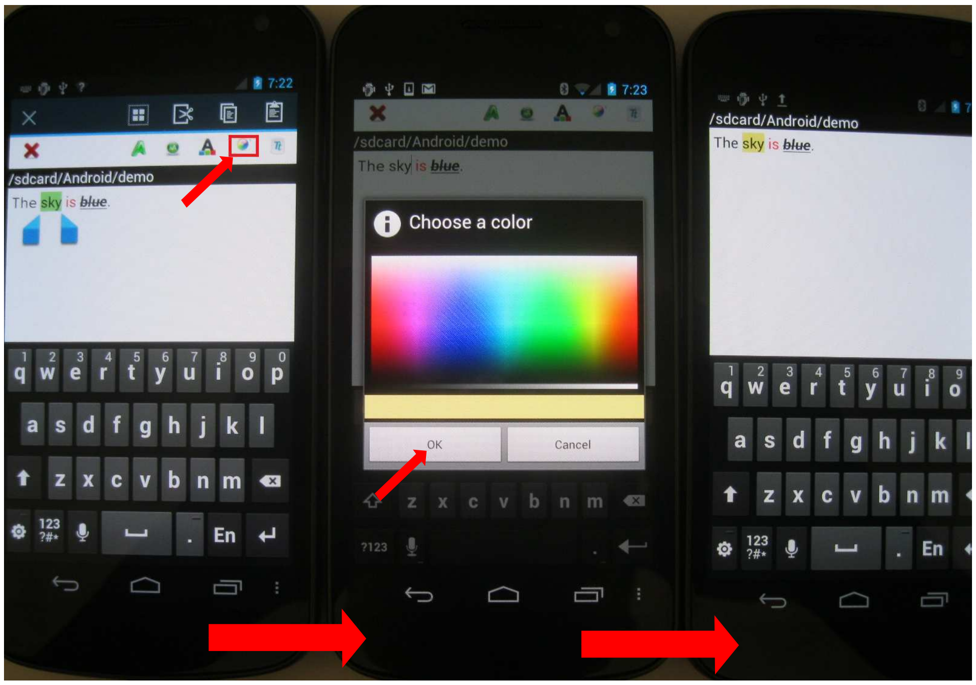


Figure 7.9: Background Color (1. Select text; 2. Select background color)

- Copy, Cut, Paste (Figure 7.10)

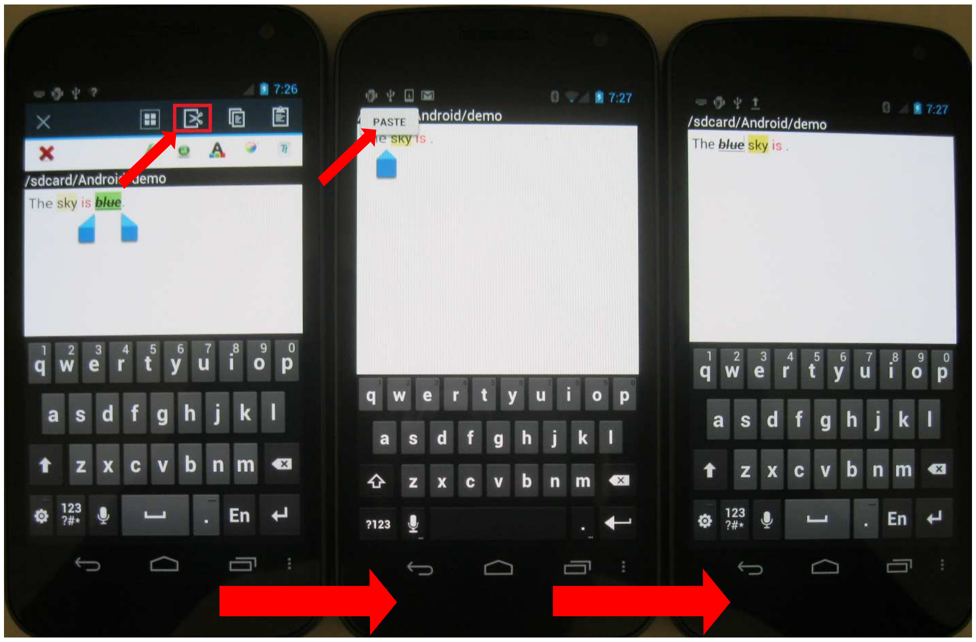


Figure 7.10: Copy, Cut, Paste (1. Cut; 2. Paste)

7.2 Collaboration Function

In order to start a collaboration session, every user who wants to join the collaboration session in the future will be distributed a configuration file for that session. Configuration file describes the name, the initial content of the shared document, the list of users in core set for the collaboration session.

Initially, collaboration session only consists of the users in core set described in its configuration file. Other sites taking the effect of the configuration file of the collaboration session can separately edit the initial content of the shared document of the collaboration session without sending and receiving updates to/from other users until they join the collaboration session to become nebular users. Figure 7.11 shows that 3 users, user A, B, and C, take effect of the configuration file of a collaboration session named "demo" in which user A and B are core users.

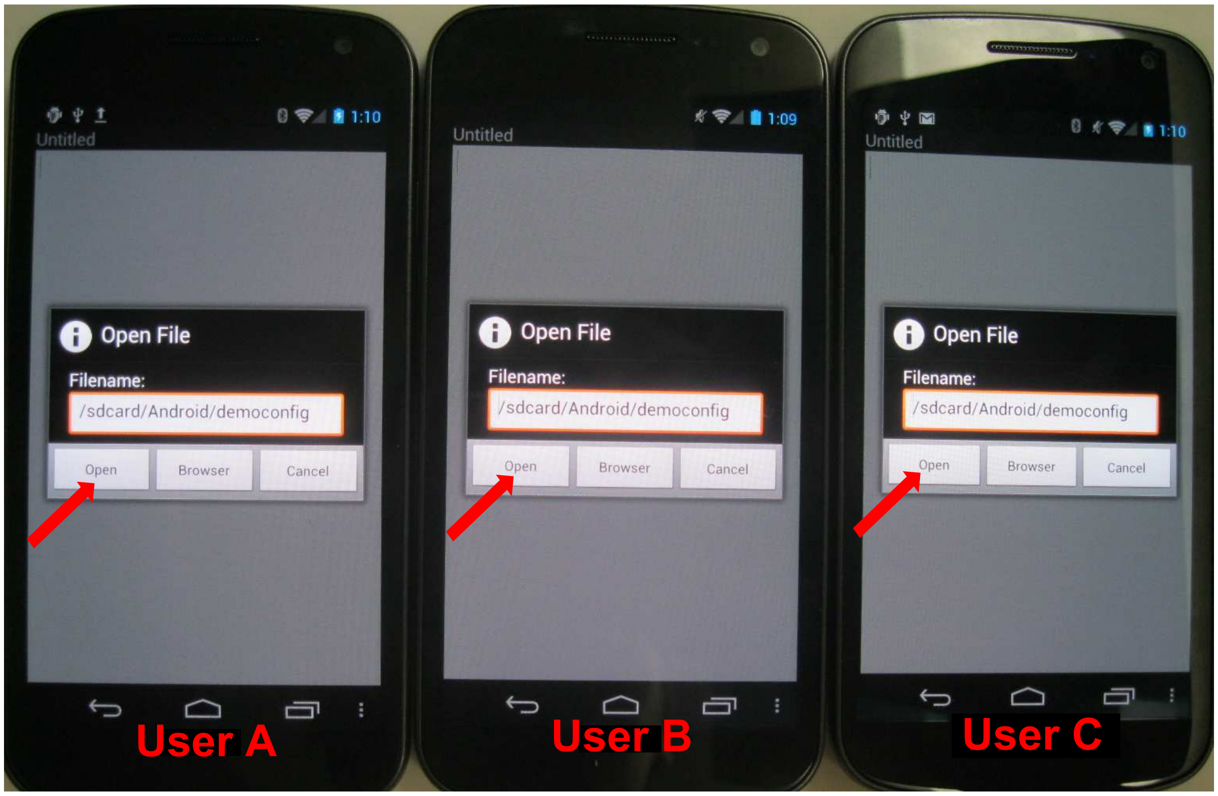
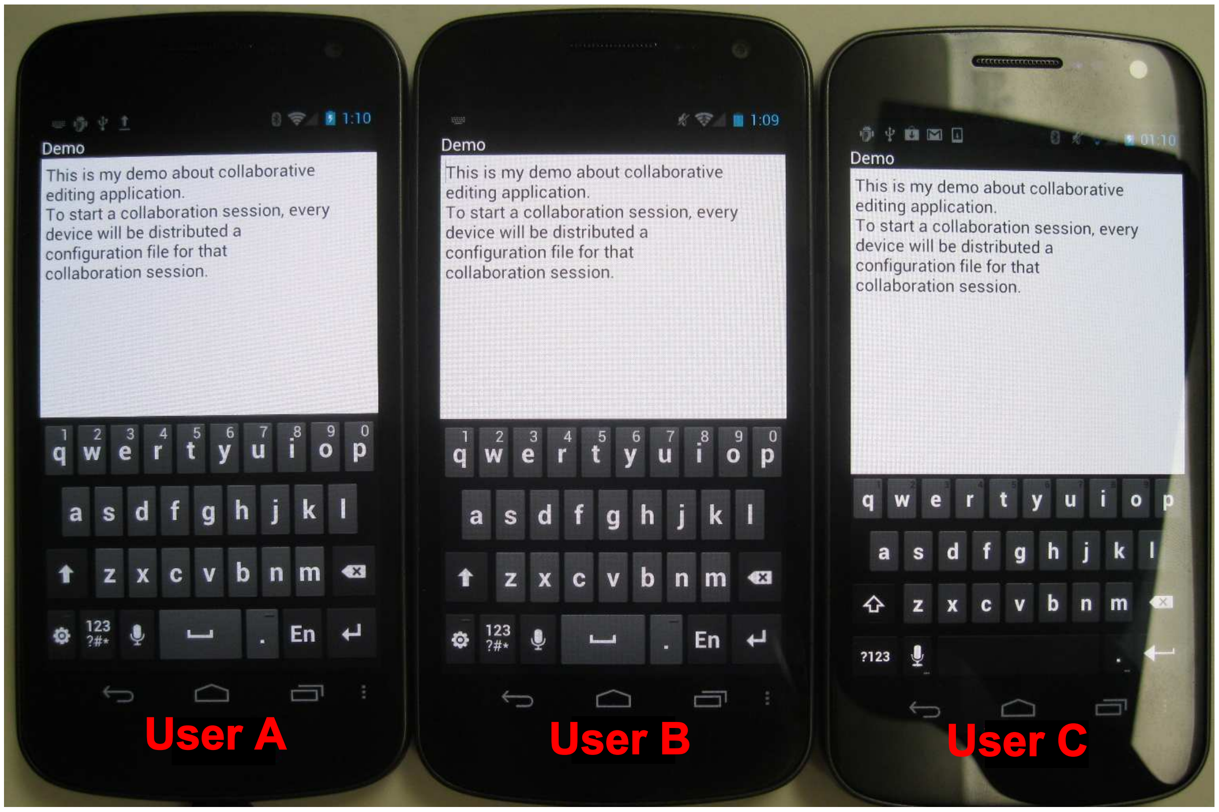


Figure 7.11: Users take effect of a collaboration configuration

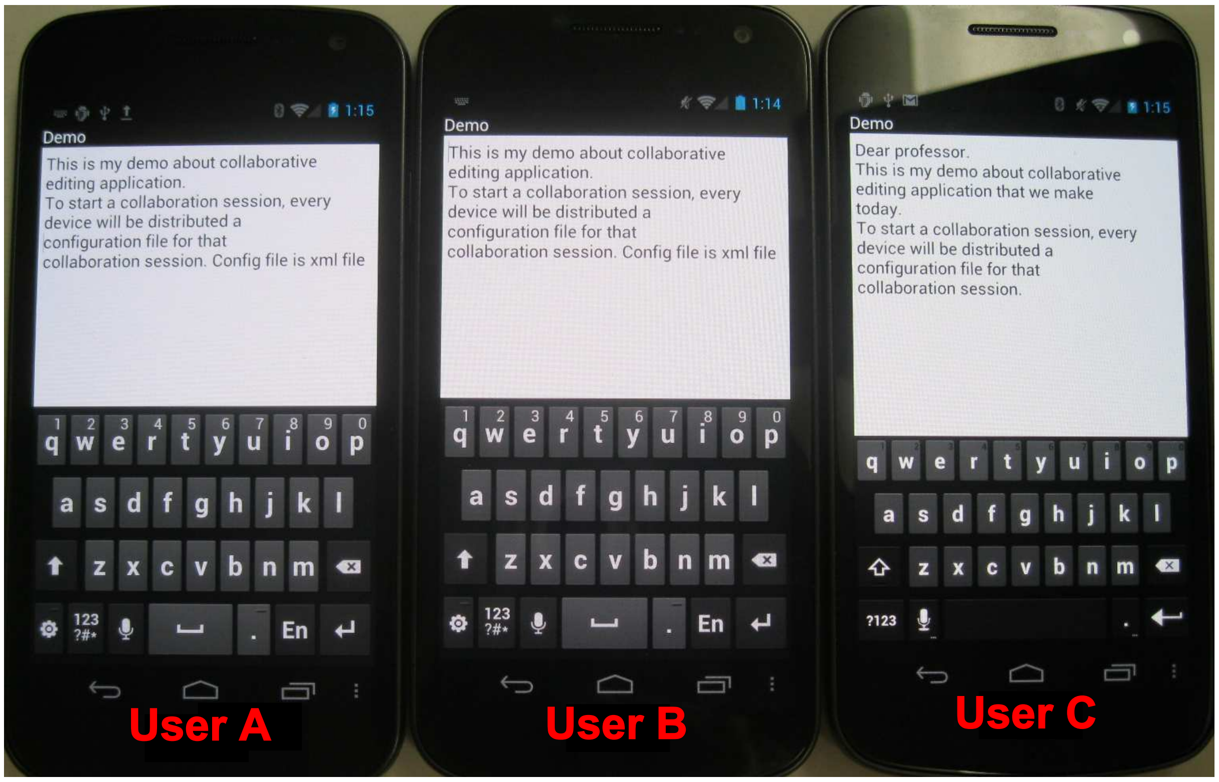
Initially, user A, B, and C begin at the same state described in the configuration file as shown in Figure 7.12.


Demo

This is my demo about collaborative editing application.
To start a collaboration session, every device will be distributed a configuration file for that collaboration session.

Figure 7.12: A initial state of users in the collaboration

Now, user A and B become core users of the collaboration session "Demo" and every update of A and B will be sent to each other to maintain a consistent state, while user C has yet to be a user in the collaboration session, and it is able to edit the shared document of that session separately. Figure 7.13 shows the states of user A, B and C after a while.



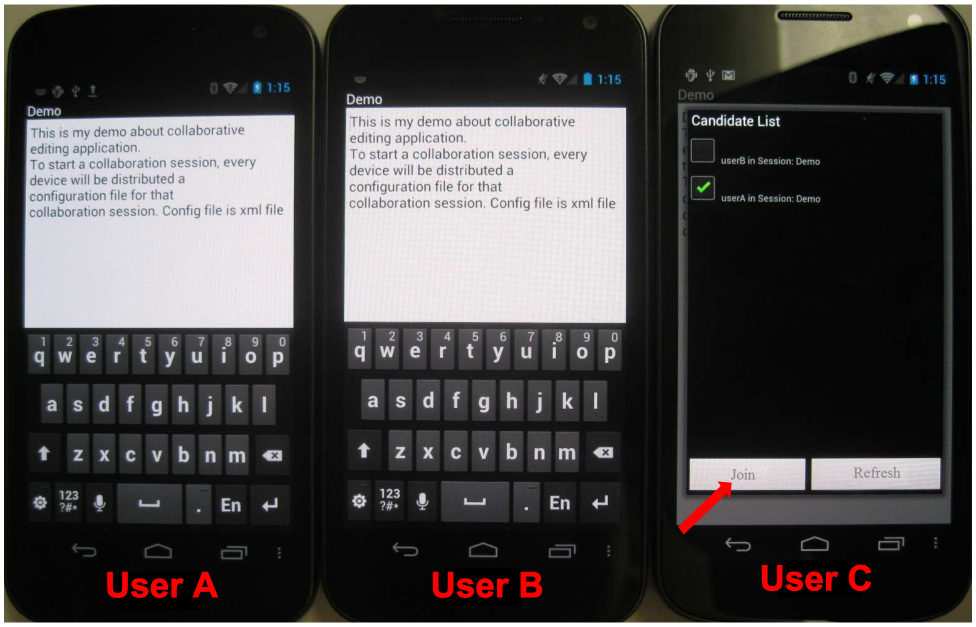
This is my demo about collaborative
editing application.
To start a collaboration session, every
device will be distributed a
configuration file for that
collaboration session. Config file is xml file

This is my demo about collaborative
editing application.
To start a collaboration session, every
device will be distributed a
configuration file for that
collaboration session. Config file is xml file

Dear professor.
This is my demo about collaborative
editing application that we make
today.
To start a collaboration session, every
device will be distributed a
configuration file for that
collaboration session.

Figure 7.13: A consistent state of users in the collaboration

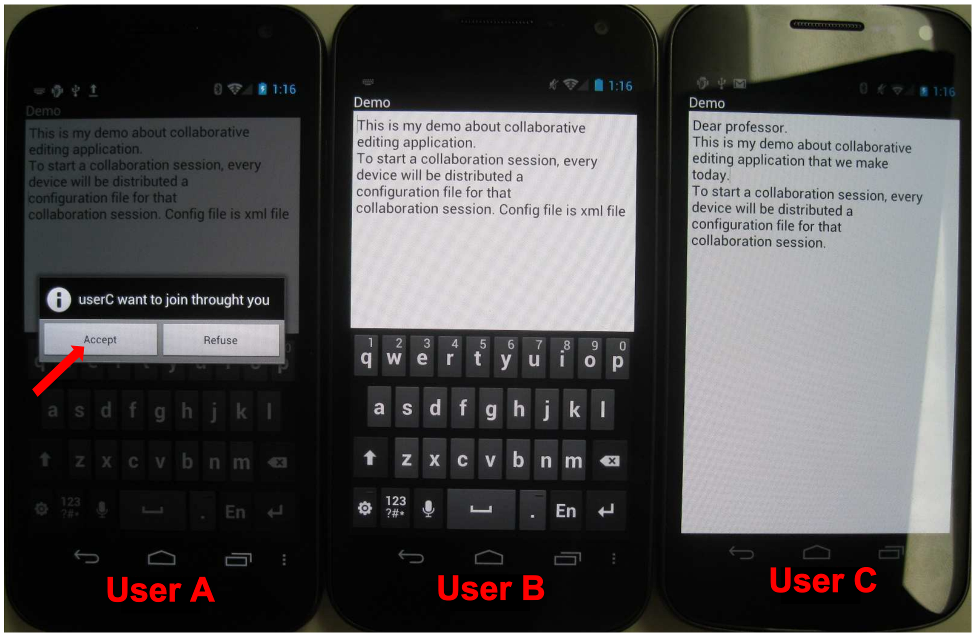
A user can join a collaboration session whenever it is connected with a current user of that collaboration session. At that time, its document is synchronized with the document of the user that it is connected with and gradually reaches the last state of the collaboration session. To continue considering the scenario in Figure 7.13, user C decides to join the collaboration session "Demo" via user A as depicted in Figure 7.14 and Figure 7.15. Figure 7.16 shows the consistent state after synchronizing the document of C with the current document of the collaboration session "Demo".



This is my demo about collaborative editing application.
To start a collaboration session, every device will be distributed a configuration file for that collaboration session. Config file is xml file

userB in Session: Demo
userA in Session: Demo

Figure 7.14: Join collaboration session

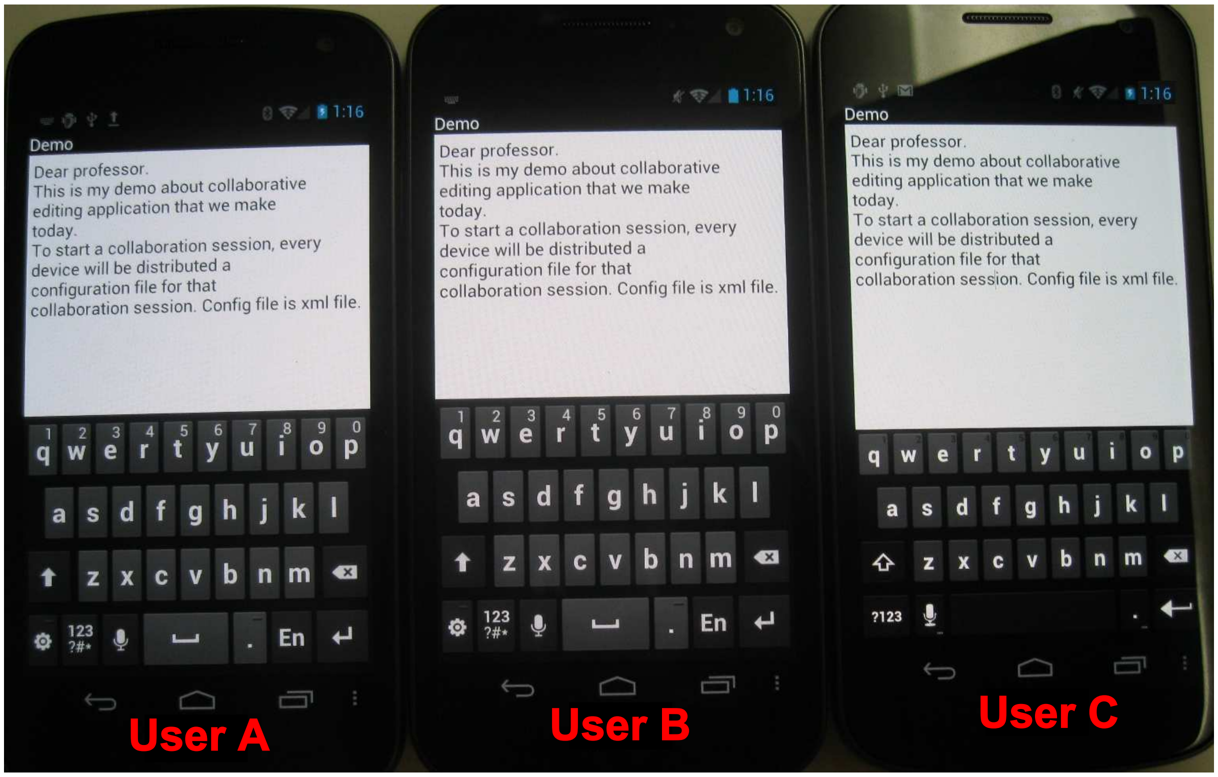


userC want to join throught you

This is my demo about collaborative editing application.
To start a collaboration session, every device will be distributed a configuration file for that collaboration session. Config file is xml file

Dear professor.
This is my demo about collaborative editing application that we make today.
To start a collaboration session, every device will be distributed a configuration file for that collaboration session.

Figure 7.15: Accept joining request



Dear professor.
This is my demo about collaborative
editing application that we make
today.
To start a collaboration session, every
device will be distributed a
configuration file for that
collaboration session. Config file is xml file.

Dear professor.
This is my demo about collaborative
editing application that we make
today.
To start a collaboration session, every
device will be distributed a
configuration file for that
collaboration session. Config file is xml file.

Dear professor.
This is my demo about collaborative
editing application that we make
today.
To start a collaboration session, every
device will be distributed a
configuration file for that
collaboration session. Config file is xml file.

Figure 7.16: The synchronization of joining user and current users in the collaboration session

Chapter 8

Conclusion

8.1 Summary

In our thesis, we concern collaborative editing application in MANETs. With the limitations of MANETs, decentralized architecture is more appropriate for collaborative editing application in MANETs because using this architecture, collaborative editing application does not need a dedicated server to manage all aspects of the collaboration. That promotes the concurrency, independence and operation in the case of disconnection. However, besides greater flexibility, decentralized collaborative application in MANETs has many problems because mobile device can move freely outside of the transmission range of other devices, while it have to receive and process correctly all updates from other devices especially concurrent updates. The major contribution of our work is the development of an Android-based application for collaborative editing in which we address the problems on document consistency and fault tolerance.

Firstly, our application replies the concept of TreeDoc, a data structure that supports eventually consistent information to ensure document consistency. With TreeDoc, concurrent operations can commute with each other, so a large number of concurrent operations do not affect application performance. We carry out the experiment on real environment with 4 android mobile devices, each of which executes 1000 operations (250 local operations + 750 remote operations). In this experiment, we compare operation time of TreeDoc with Operation Transformation, an approach used by many collaborative editing applications. In the best case and average case, operation time of TreeDoc is always better than operation time of Operational Transformation. However, in the worst case, operation time of TreeDoc is only better than operation time of Operational Transformartion if the number of current operations is greater than 10 operations. With periodically executing rebalancing, the height of TreeDoc decreases, which results in the decrease of operation time in the worst case of TreeDoc. By experimenting, we determine that the best value of rebalancing period is equal to 200 operations because with that rebalancing period, total time in the worst case of TreeDoc consisting of rebalancing time and operation time is minimal. With rebalancing period of 200 operations, operation time in the worst case of TreeDoc is reduced and always better than operation time of Operational

Transformation.

Secondly, we consider disconnection event in MANETs. In MANETs, mobile device frequently gets disconnected from other devices. In the scope of this thesis, we concentrate on the disconnection in the case where mobile device goes too far out of the transmission range of other devices. When a site is disconnected, it stops receiving and broadcasting message from/to sites that it is disconnected. Consequently, in the collaboration session, when a user is disconnected, it misses messages consisting of joining messages, leaving messages, and document operation messages from/to users that it is disconnected from. However, disconnected user is still able to work on local document. In order to tolerating disconnection event, we provide a solution in which every user bases on state vector appended with each message to detect its missing messages. Consequently, every message generated by a user in the collaboration session eventually arrives at all others users regardless of disconnection event.

8.2 Open Questions

Optimizations on TreeDoc

The main problem of TreeDoc is unbalance, which causes the path of node to grow indefinitely. Therefore, it is necessary to provide optimizations to TreeDoc in order to reduce the degree of unbalance or even design new commutative replicated data type to alleviate this problem. Besides insert, delete operation, TreeDoc needs to provide more kinds of operation such as undo, redo, lock, unlock...

User Identification

In a wider environment, the configuration for collaboration session could be information provided by a document server. In that case, user should be identified by some credentials rather than IP address. Because IP address can change in a mobile environment, identifying a user (a device) by its IP address is a poor choice.

Message Broadcast on MANETs

In our current application, we did not pay attention on the message broadcast scheme on MANETs. In the future, it is necessary to develop mechanisms for broadcasting message on MANETs to reduce message loss in the system.

Bibliography

[1] J. F. Patterson, R. D. Hill, S. L. Rohall and S. W. Meeks, Rendezvous: an architecture for synchronous multi-user applications, In Proceedings of the 1990 ACM conference on Computer Supported Cooperative Work, pp. 317-328, Los Angeles, California, United States. ACM Press, 1990.

[2] N. A. Streitz, J. Geiler, J. M. Haake and J. Hol, DOLPHIN: integrated meeting support across local and remote desktop environments and LiveBoards, In Proceedings of the 1994 ACM conference on Computer Supported Cooperative Work, Chapel Hill, North Carolina, United States. ACM Press, 1994.

[3] D. A. Nichols, P. Curtis, M. Dixon and J. Lamping, High-latency, low-bandwidth windowing in the Jupiter collaboration system, In Proceedings of the 8th annual ACM symposium on User interface and software technology, pp. 111-120, Pittsburgh, Pennsylvania, United States. ACM Press, 1995.

[4] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning and L. Suchman, Beyond the chalkboard: computer support for collaboration and problem solving in meetings, Communications of the ACM, 30(1): 32-47, 1987.

[5] C. A. Ellis, S. J. Gibbs and G. L. Rein, Design and Use of a Group Editor, In Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, pp. 13-28, Napa Valley, California. Elsevier, 1989.

[6] C. Sun and C. Ellis, Operational transformation in real-time group editors: issues, algorithms, and achievements, In Proceedings of the 1998 ACM conference on Computer Supported Cooperative Work, pp. 59-68, Seattle, Washington, United States. ACM Press, 1998.

[7] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen, Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems, ACM Transactions on Computer-Human Interaction (TOCHI), 5(1): 63-108, 1998.

[8] A. Karsenty, C. Tronche and M. Beaudouinlafon, GroupDesign: shared editing in a heterogeneous environment, Usenix Journal of Computing Systems, 6(2): 167-195, 1993.

[9] C. Sun and D. Chen, Consistency maintenance in real-time collaborative graphics editing systems, ACM Transactions on Computer-Human Interaction (TOCHI), 9(1): 1-41, 2002.

[10] C. L. Ignat and M. C. Norrie, Draw-together: graphical editor for collaborative drawing, In Proceedings of the 2006 20th anniversary conference on Computer supported Cooperative Work, pp. 269-278, Banff, Alberta, Canada. ACM Press, 2006.

[11] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Comm. ACM, vol. 21, no. 7, pp. 558-565, 1978.

[12] P.R. Johnson and R.H. Thomas, RFC 677: Maintenance of Duplicate Databases, http://www.faqs.org/rfcs/rfc677.html, Jan. 1975.

[13] C. A. Ellis and S. J. Gibbs, Concurrency control in groupware systems, In Proceedings of the 1989 ACM SIGMOD international conference on Management of data, pp. 399-407, Portland, Oregon, United States. ACM Press, 1989.

[14] M. Ressel, D. Nitsche-Ruhland and R. Gunzenhuser, An integrating, transformation-oriented approach to concurrency control and undo in group editors, In Proceedings of the 1996 ACM conference on Computer Supported Cooperative Work, pp. 288-297, Boston, Massachusetts, United States. ACM Press, 1996.

[15] M. Suleiman, M. Cart and J. Ferri, Serialization of concurrent operations in a distributed collaborative environment, In Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge, pp. 435-445, Phoenix, Arizona, United States. ACM Press, 1997.

[16] N. Vidot, M. Cart, J. Ferri and M. Suleiman, Copies convergence in a distributed real-time collaborative environment, In Proceedings of the 2000 ACM conference on Computer Supported Cooperative Work, pp. 171-180, Philadelphia, Pennsylvania, United States. ACM Press, 2000.

[17] G. Oster, P. Urso, P. Molli, and A. Imine, Proving correctness of transformation functions in collaborative editing systems, LORIAINRIA Lorraine, Rapport de recherche RR-5795, Dec. 2005.

[18] F. Mattern, Virtual time and global states of distributed systems, In Proceedings of the International Workshop on Parallel and Distributed Algorithms, pp. 215-276. Elsevier Pub., 1989.

[19] Nuno Preguia, Joan Manuel Marqus, Marc Shapiro, and Mihai Letia, A commutative replicated data type for cooperative editing, In Int. Conf. on Distributed Comp. Sys. (ICDCS), pages 395403, Montral, Canada, June 2009.

[20] Marek Zawirski, Marc Shapiro, and Nuno Preguia, Asynchronous rebalancing of a replicated tree, Conference Franaise de Systmes d'Exploitation (CFSE), Saint-Malo, France, May 2011.

[21] D. Buszko, W. Lee and A. Helal, Decentralized ad-hoc groupware API and framework for mobile collaboration, In Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work, pp. 5-14, Boulder, Colorado, USA. ACM Press, 2001.

[22] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen and W. Cai, Transparent adaptation of single-user applications for multi-user real-time collaboration, ACM Transactions on Computer-Human Interaction (TOCHI), 13(4): 531 - 582, 2006.

[23] Allison C., Concurrency Control for Real Time Groupware, CE94: Concurrent Engineering Research and Applications, pp. 163170, A global Perspective, Pittsbourg, August 1994.

[24] Mark Claypool, Tom Beibeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, The Effects of Loss and Latency on User Performance in Unreal Tournment 2003, Computer Science Department at Worchester Polytechnic Institute, 2004.

[25] Paul Bettner, Mark Terrano, 1500 archers on a 28.8: Network programming in Age of Empires and beyond, the game developer conference proceedings, 2001.

[26] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski, A comprehensive study of Convergent and Commutative Replicated Data Types, Rapport de recherche 7506, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011.

Appendix A

System Architecture

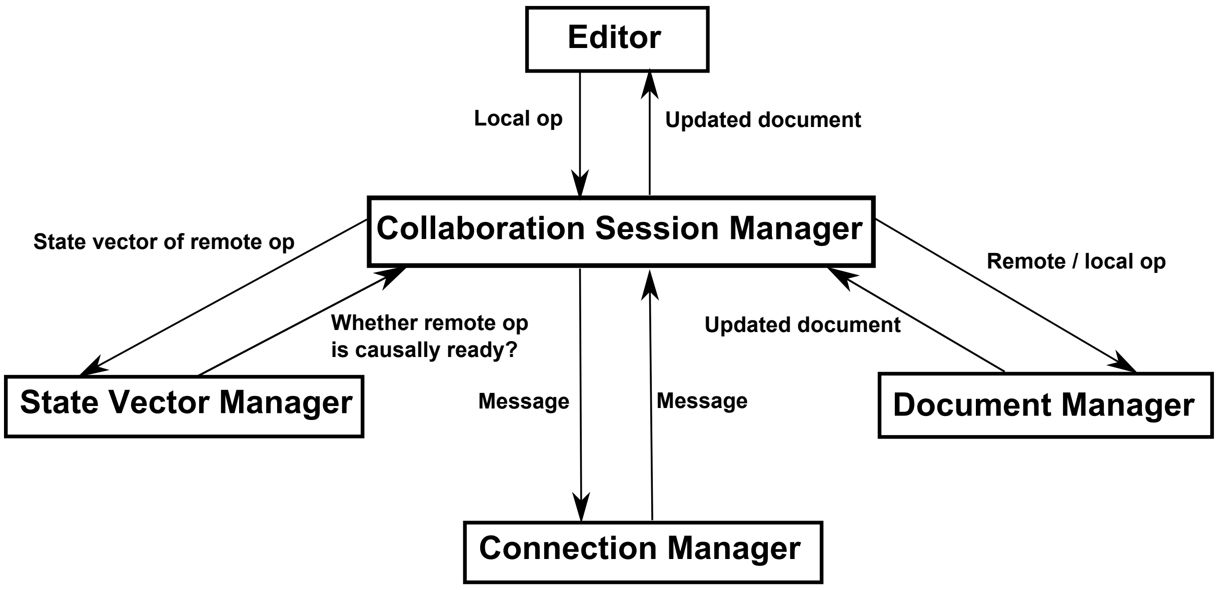


Figure A.1: System Architecture

Appendix B

Class Diagram

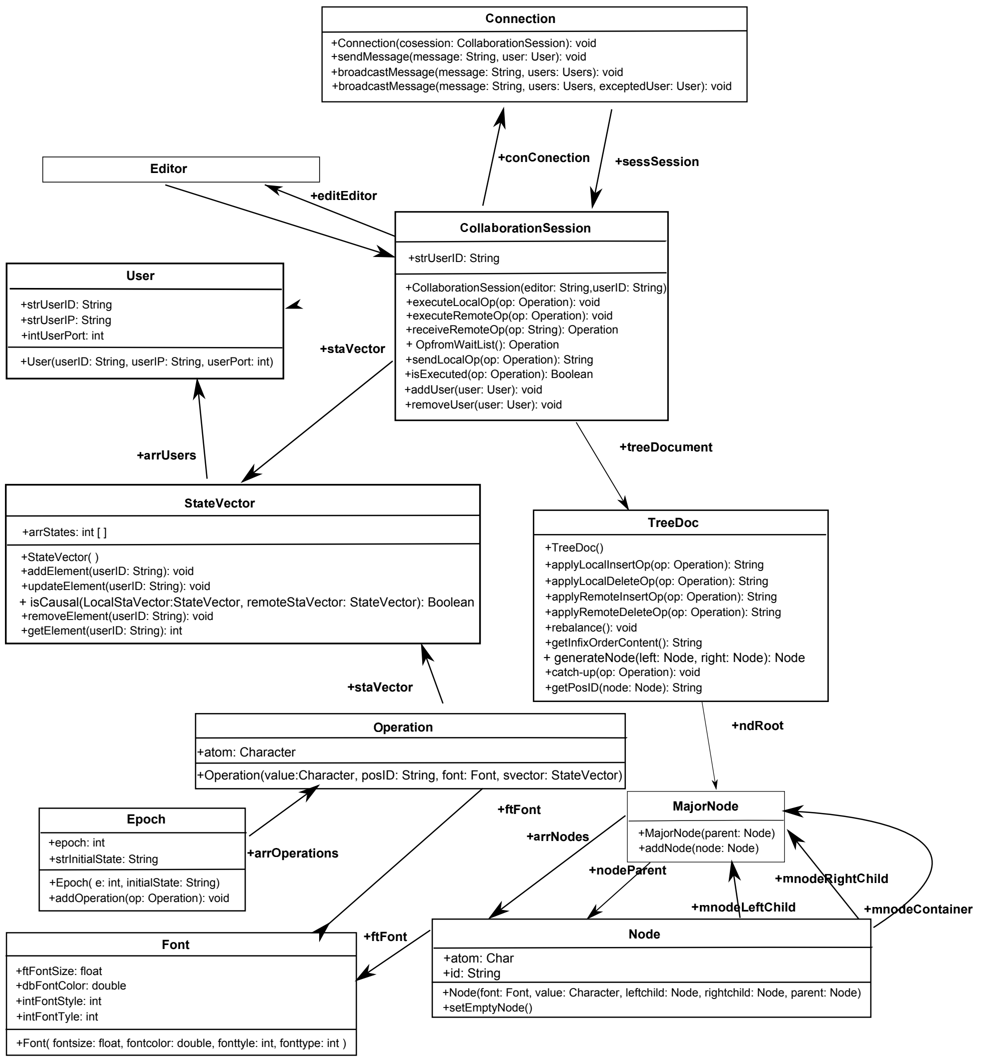

Connection
+Connection(cosession: CollaborationSession): void
+sendMessage(message: String, user: User): void
+broadcastMessage(message: String, users: Users): void
+broadcastMessage(message: String, users: Users, exceptedUser: User): void
+conConection
+sessSession
Editor
+editEditor
CollaborationSession
+strUserID: String
+CollaborationSession(editor: String,userID: String)
+executeLocalOp(op: Operation): void
+executeRemoteOp(op: Operation): void
+receiveRemoteOp(op: String): Operation
+ OpfromWaitList(): Operation
+sendLocalOp(op: Operation): String
+isExecuted(op: Operation): Boolean
+addUser(user: User): void
+removeUser(user: User): void
User
+strUserID: String
+strUserIP: String
+intUserPort: int
+User(userID: String, userIP: String, userPort: int)
+staVector
+arrUsers
+treeDocument
StateVector
+arrStates: int [ ]
+StateVector( )
+addElement(userID: String): void
+updateElement(userID: String): void
+ isCausal(LocalStaVector:StateVector, remoteStaVector: StateVector): Boolean
+removeElement(userID: String): void
+getElement(userID: String): int
TreeDoc
+TreeDoc()
+applyLocalInsertOp(op: Operation): String
+applyLocalDeleteOp(op: Operation): String
+applyRemoteInsertOp(op: Operation): String
+applyRemoteDeleteOp(op: Operation): String
+rebalance(): void
+getInfixOrderContent(): String
+ generateNode(left: Node, right: Node): Node
+catch-up(op: Operation): void
+getPosID(node: Node): String
+staVector
+ndRoot
Operation
+atom: Character
+Operation(value:Character, posID: String, font: Font, svector: StateVector)
+ftFont
MajorNode
+MajorNode(parent: Node)
+addNode(node: Node)
+arrNodes
Epoch
+epoch: int
+strInitialState: String
+Epoch(e: int, initialState: String)
+addOperation(op: Operation): void
+arrOperations
+nodeParent
+mnodeRightChild
+mnodeLeftChild
+mnodeContainer
+ftFont
Font
+ftFontSize: float
+dbFontColor: double
+intFontStyle: int
+intFontTyle: int
+Font(fontsize: float, fontcolor: double, fonttyle: int, fonttype: int)
Node
+atom: Char
+id: String
+Node(font: Font, value: Character, leftchild: Node, rightchild: Node, parent: Node)
+setEmptyNode()

Figure B.1: Class Diagram

Appendix C

TreeDoc Source Code

```
import java.util.ArrayList;
public class TreeDoc {
  MajorNode root;
  /**
   * Apply remote delete operation
   * @param op: delete operation
   * @return deleled node
   */
  public DocNode applyRemoteDeleteOp(Operation op) {
    DocNode u = getNode(op.posID);
    u.setAtom(null);
    return u;
  }
  /**
   * Apply remote insert operation
   * @param op: insert operation
   * @return inserted node
   */
  public DocNode applyRemoteInsertOp(Operation op) {
    DocNode u = getNode(op.posID);
    u.setAtom(op.atom);
    u.setFont(op.font);
    return u;
  }
  /**
   * Get the node corresponding with a posID
   * @param posID: posID of the node
   * @return the node corresponding with posID
   */
  public DocNode getNode(String posID) {
    ArrayList<String> pathdis = new ArrayList<String>();
    ArrayList<Integer> pathbit = new ArrayList<Integer>();
    decodePosID(posID, pathdis, pathbit);
    DocNode u = null;
    for (int i = 0; i < pathdis.size(); i++) {
      String disg = pathdis.get(i);
      int direction = pathbit.get(i);
      u = gotoNode(u, direction, disg);
    }
    return u;
  }
  public DocNode gotoNode(DocNode u, int direction, String disg) {
    MajorNode tempMj = null;
    if (u == null)
      tempMj = root;
    else {
      if (direction == 1)
        tempMj = u.right;
      else
        tempMj = u.left;
    }
    if (tempMj != null) {
      int i;
      for (i = 0; i < tempMj.arrDocNode.size(); i++) {
        if (disg.equals(tempMj.arrDocNode.get(i).id) == true) {
          u = tempMj.arrDocNode.get(i);
          break;
        }
        if (tempMj.arrDocNode.get(i).id.compareTo(disg) > 0) {
          DocNode uu = new DocNode(null, false, null, disg);
          uu.majorNode = tempMj;
          tempMj.arrDocNode.add(i, uu);
          u = uu;
          break;
        }
      }
      if (i == tempMj.arrDocNode.size()) {
        DocNode uu = new DocNode(' ', false, null, disg);
        uu.majorNode = tempMj;
        tempMj.arrDocNode.add(uu);
        u = uu;

      }
    } else {
      DocNode uu = new DocNode(' ', false, null, disg);
      MajorNode mm = new MajorNode();
      mm.add(uu);
      mm.parrent = u;
      if (direction == 1)
        u.right = mm;
      else if (direction == 0)
        u.left = mm;
      else
        root = mm;
      u = uu;
    }
    return u;
  }
  /**
```

```
 * Generate a node on the tree
 * @param left: the node at right position of inserted node
 * @param right: the node at left position of inserted node
 * @param u: inserted node
 * @return inserted node
 */
public DocNode generateNode(DocNode left, DocNode right, DocNode u) {
  if (root == null) {
    MajorNode um = new MajorNode();
    um.add(u);
    root = um;
    return u;
  }
  if (left == null && right == null) {
    u = addToMajorNode(root, u);
    return u;
  }
  if (left == null) {
    if (right.left == null) {
      MajorNode um = new MajorNode();
      um.add(u);
      right.left = um;
      um.parrent = right;
    } else {
      u = addToMajorNode(right.left, u);
    }
    return u;
  }
  if (right == null) {
    if (left.right == null) {
      MajorNode um = new MajorNode();
      um.add(u);
      left.right = um;
      um.parrent = left;
    } else {
      u = addToMajorNode(left.right, u);
    }
    return u;
  }
  if (left.isancestor(right) == true) {

    if (right.left == null) {
      MajorNode um = new MajorNode();
      um.add(u);
      right.left = um;
      um.parrent = right;
    } else {
      u = addToMajorNode(right.left, u);
    }

  } else {
    if (left.right == null) {
      MajorNode um = new MajorNode();
      um.add(u);
      left.right = um;
      um.parrent = left;
    } else {
      u = addToMajorNode(left.right, u);
    }
  }
  return u;
}
/**
 * Apply local insert operation
 * @param before: the node at right position of inserted node
 * @param after: the node at left position of inserted node
 * @param u: inserted node
 * @return posID of inserted node
 */
public String applyLocalInsertop(DocNode before, DocNode after, DocNode u
    )
{
  u=generateNode(before, after, u);
  String p = getPosID(u);
  return p;
}
/**
 * Apply local delete operation
 * @param u: deleted node
 * @return posID of deleted node
 */
public String applylocalDeleteOperation(DocNode u) {
  u.setAtom(null);
  String p = getPosID(u);
  return p;
}
public DocNode addToMajorNode(MajorNode mj, DocNode u) {
  int i;
  for (i = 0; i < mj.arrDocNode.size(); i++) {
    if (mj.arrDocNode.get(i).id.compareTo(u.id) > 0) {
      u.majorNode = mj;
      mj.arrDocNode.add(i, u);
      return u;
    }
  }
  if (i == mj.arrDocNode.size()) {
    u.majorNode = mj;
    mj.arrDocNode.add(u);
    return u;
  }
  return null;
}
long desiredheight;
long extranode;
```

```
static int currentheight;
int addedindex;
/**
 * Rebanlance Tree
 * @param NodeList: the list of nodes of rebalanced Tree
 */
public void rebalancing(ArrayList<DocNode> NodeList) {
  for (int y = NodeList.size() - 1; y >= 0; y--) {
    if (NodeList.get(y).getAtom() == null) {
      NodeList.remove(y);
    }
  }
  desiredheight = ((long) (Math.log(NodeList.size() + 1) / Math.log(2)))
     - 1;
  extranode = (long) (NodeList.size() - Math.pow(2, desiredheight + 1) +
     1);
  if (extranode > 0) {
    desiredheight++;
  }
  currentheight = 0;
  addedindex = 0;
  Rebalancing(root, NodeList);
}
void Rebalancing(MajorNode u, ArrayList<DocNode> NonemptyNdList) {
  while (u.arrDocNode.size() > 1) {
    u.arrDocNode.remove(1);
  }
  if (u.arrDocNode.get(0).left != null) {
    if (currentheight < desiredheight) {
      currentheight++;
      Rebalancing(u.arrDocNode.get(0).left, NonemptyNdList);
      currentheight--;
    } else {
      u.arrDocNode.get(0).left = null;
    }
  } else {
    if (currentheight < desiredheight) {
      DocNode uu = new DocNode(' ', true, null, "1");
      MajorNode mm = new MajorNode();
      mm.add(uu);
      mm.parrent = u.arrDocNode.get(0);
      u.arrDocNode.get(0).left = mm;
      currentheight++;
      Rebalancing(u.arrDocNode.get(0).left, NonemptyNdList);
      currentheight--;
    }
  }
  u.arrDocNode.get(0).id = "1";
  u.arrDocNode.get(0).setAtom(NonemptyNdList.get(addedindex).getAtom());
  u.arrDocNode.get(0).rebalance = true;
  addedindex++;
  if (currentheight == desiredheight && extranode > 0) {
    extranode--;
    if (extranode == 0) {
      desiredheight--;
    }
  }
  if (u.arrDocNode.get(0).right != null) {
    if (currentheight < desiredheight) {
      currentheight++;
      Rebalancing(u.arrDocNode.get(0).right, NonemptyNdList);
      currentheight--;
    } else {
      u.arrDocNode.get(0).right = null;
    }
  } else {
    if (currentheight < desiredheight) {
      DocNode uu = new DocNode(' ', true, null, "1");
      MajorNode mm = new MajorNode();
      mm.add(uu);
      mm.parrent = u.arrDocNode.get(0);
      u.arrDocNode.get(0).right = mm;
      currentheight++;
      Rebalancing(u.arrDocNode.get(0).right, NonemptyNdList);
      currentheight--;
    }
  }
}
static int nonEmptyNode;
static int EmptyNode;
  /**
   * Translate nodes that don't belong to epoch n to epoch n+1
   * @param a: Tree before rebalancing
   * @param b  Tree after rebalancing
   */
public static void Translate(TreeDoc a, TreeDoc b) {
  currentheight = 0;
  nonEmptyNode = -1;
  EmptyNode = 0;
  ArrayList<String> posIDPair = new ArrayList<String>();
  visitforTranslate(a, a.root, b, posIDPair);
}
static public void visitforTranslate(TreeDoc a, MajorNode r, TreeDoc b,
    ArrayList<String> posIDPair) {
  for (int i = 0; i < r.arrDocNode.size(); i++) {
    if (r.arrDocNode.get(i).rebalance == true
        && r.arrDocNode.get(i).left != null) {
      currentheight++;
      visitforTranslate(a, r.arrDocNode.get(i).left, b, posIDPair);
      currentheight--;
    }
    DocNode temp = r.arrDocNode.get(i);
    if (temp.rebalance == true) {
      if (temp.getAtom() != null) {
        nonEmptyNode++;
        EmptyNode = 0;
      } else {
        EmptyNode++;
      }
    } else {
      String oldPosID = a.getPosID(temp);
      applytoRebalanceTree(currentheight, EmptyNode, nonEmptyNode,
          temp, b);
      String newPosID = b.getPosID(temp);
      posIDPair.add(oldPosID);
      posIDPair.add(newPosID);
    }
    if (r.arrDocNode.get(i).rebalance == true
        && r.arrDocNode.get(i).right != null) {
      r.arrDocNode.get(i).right.parrent = r.arrDocNode.get(i);
      currentheight++;
      visitforTranslate(a, r.arrDocNode.get(i).right, b, posIDPair);
      currentheight--;
    }
  }
}
public static void applytoRebalanceTree(int Height, int IndexEmptyNode,
    int NotEmptyNode, DocNode temp, TreeDoc b) {
  String dis = "0";
  char strHeigh = (char) Height;
  if (IndexEmptyNode == 0) {
    dis = dis + "|" + "+" + strHeigh;
  } else {
    char strindexEmptyNode = (char) IndexEmptyNode;
    dis = dis + "|" + "-" + strindexEmptyNode + "|" + "+" + strHeigh;
  }
  if (temp.id.equals("") == false) {
    dis = dis + "|" + temp.id;
  }
  temp.id = dis;
  if (NotEmptyNode == -1) {
    DocNode u = b.FindNode(0);
    if (u.left == null) {
      MajorNode mm = new MajorNode();
      mm.add(temp);
      temp.majorNode = mm;
      u.left = mm;
      mm.parrent = u;

    } else {
      temp.majorNode = u.left;
      if (u.left.arrDocNode.get(u.left.arrDocNode.size() - 1).id
          .equals("1") == true) {
        u.left.arrDocNode.add(u.left.arrDocNode.size() - 1, temp);

      } else {
        u.left.arrDocNode.add(temp);
      }
    }
  } else {
    DocNode u = b.FindNode(NotEmptyNode);
    if (u.right == null) {
      MajorNode mm = new MajorNode();
      mm.add(temp);
      temp.majorNode = mm;
      u.right = mm;
      mm.parrent = u;
    } else {
      temp.majorNode = u.right;
      if (u.right.arrDocNode.get(u.right.arrDocNode.size() - 1).id
          .equals("1") == true) {
        u.right.arrDocNode.add(u.right.arrDocNode.size() - 1, temp);

      } else {
        u.right.arrDocNode.add(temp);
      }
    }
  }
}
/**
 * Get posID of a node on the tree
 * @param u: the node on the tree
 * @return posID of parameter node
 */
public String getPosID(DocNode u) {
  String pathbit = "";
  String pathdis = "";
  byte a = 1;
  byte onebyte = 0;
  int numberofbit = 0;
  DocNode temp = u;
  while (temp.majorNode != root) {
    String d = temp.id;
    MajorNode majorNode = temp.majorNode;
    temp = temp.majorNode.parrent;
    if (temp.right == majorNode) {
      onebyte = (byte) (onebyte | a);
    }
    numberofbit++;
    a = (byte) (a << 1);
    if (numberofbit % 8 == 0) {
      pathbit = pathbit + (char) ((onebyte << 8) >>> 8);
      a = 1;
      onebyte = 0;
      numberofbit = 0;
    }
    pathdis = d + "-" + pathdis;
  }
  return pathbit + ":" + pathdis;
}
/**
 * Decode posID
 * @param receivedposID
 * @param pathdis: the array of disambiguous of posID
 * @param pathbit: the array of bit of posID
 */
void decodePosID(String posID, ArrayList<String> pathdis,
    ArrayList<Integer> pathbit) {
  int index = posID.indexOf(":");
  String disarr = posID.substring(0, index);
  String bitarr = posID.substring(index + 1, posID.length());
  index = disarr.indexOf("-");
  while (index != -1) {
    pathdis.add(disarr.substring(0, index));
    disarr = disarr.substring(index + 1, disarr.length());
    index = disarr.indexOf("-");
  }
  pathdis.add(disarr);
  kk = bitarr;
  ii = -1;
  for (int i = 0; i < pathdis.size() - 1; i++) {
    pathbit.add(0, getNext());
  }

}
String kk;
int ii,jj;
byte dd;
public int getNext() {

  int k = 0;
  if (ii == -1) {
    jj = 0;
  }
  if (ii == 8 || ii == -1) {
    dd = (byte) kk.charAt(jj);
    jj++;
    ii = 0;
  }
  if (ii < 8) {
    k = (int) (dd & 1);
    dd = (byte) (dd >> 1);
    ii++;
  }
  return k;
}
}
```

Listing C.1: `TreeDoc.class`

```java
import java.util.ArrayList;
public class MajorNode {
  public DocNode parrent;
  public ArrayList<DocNode> arrDocNode;
  public MajorNode() {
    arrDocNode = new ArrayList<DocNode>();
    parrent = null;
  }
  /**
   * add a DocNode to major node
   * @param u: DocNode
   */
  public void add(DocNode u) {
    u.majorNode = this;
    arrDocNode.add(u);
  }
}
```

Listing C.2: `MajorNode.class`

```java
public class DocNode {
  public Character atom;
  public String id;
  public MajorNode majorNode;
  public MajorNode left;
  public MajorNode right;
  public boolean rebalance;
  public Font font;
  public DocNode(Character a, boolean s,Font ft,String dis) {
    atom = a;
    left = null;
    right = null;
    id=dis;
    font=ft;
    rebalance=false;
  }
}
```

Listing C.3: `DocNode.class`

```java
import java.util.ArrayList;
public class Operation {
  public int  epoch;
  public String generator;
  public StateVector stateVector;
  public int optype;
  public String posID;
  public Character atom;
  public Font font;
  public Operation(String gen,StateVector vers,int opetype, String pos,Font
      f, int e, Character a)
  {
        generator=gen;
        stateVector=vers;
        optype=opetype;
        atom=a;
        font=f;
        epoch=e;
        posID=pos;
    }
}
```

Listing C.4: `Operation.class`