A Document Server for Collaborative Concurrent Editing

Y. QIN & X. WEI

July 30, 2014

1 Introduction

A collaborative concurrent editor enables multiple users to edit on a same document. In this project, we have realized a document server in Java that receives and executes concurrent editing operations from multiple clients and executes them in real-time. Our main work is the following:

Firstly, we adopt a method called Operation Transformation which can handle a sequence of operations representing concurrent editing operations of a large number of clients on a same document. As previous transformation leads to incorrect results in certain cases, we then introduce the Tombstones Transformation Functions (TTF), which have been proved to be correct, to improve the performance of our program in terms of validity. Next, a server is developed to execute operations of clients in real-time by using synchronized Java threads. We also design an UI for server and clients. Finally, we realize two extensions of our program: 1. extending input of insert and delete operations from a single character to a string; 2. adding an update operation that enables a character to be modified in-place.

The two operation transformations mentioned above are presented in the following two parts(Part 2 and Part 3). Then we provide our implementation in Part 4, by explaining the algorithms and data structures that we adopt in the program. The realization of the two extensions are presented the Part 5. Finally, demonstrations of the program in different cases will be given. We will also discuss possible improvements of our present work. The user guide of our program can be found at the end of this report.

2 Operational Transformation

Collaborative editing systems allow users to edit the same document. Different clients may intend to execute parallel modifications on a same sentence or word and therefore potential conflicts may occur. Thus, the server has to handle correctly such modifications by using more complex algorithms instead of a simple merge of clients' edits. There are two ways to deal with the problem of inconsistency: pessimistic approach and optimistic approach. Pessimistic approach does not allow users to edit the same piece of text at the same time, whereas optimistic approach enables users to edit on their own versions of a document and then server merges theirs edits correctly, making it more suitable for collaborative editing.

A well-known optimistic approach is called **Operational Transformations** . It considers one server and N clients. Each client owns a version of the document. The server maintains a master copy of the document and a log of clients' operations. For the moment, we only consider the following operations for clients:

`insert (uid,ver,char,pos)`, where `uid` is the unique identifier of the user who generated the operation, `ver` is the document version number at this user, and `char`is the single character that the user wants to insert at position `pos`.

`delete(uid,ver,pos)`, where `uid` and `ver` have the same signification as those in insert, and `pos`is the position from which a character is to be deleted.

We add another operation which can be useful later:

`noop(uid,ver)`, an identity operation that does nothing to the document.

The key idea of the Operational Transformation approach is transformation function. A transformation **T** takes two parameters `op1` and `op2`, which has the same version. The function computes a new operation **T(op1,op2)** that is equivalent to `op1` but has a version number after `op2`. Suppose that after **n** operations, the version number is **n** and the log contains **n** operations $O_1, ..., O_n$. When the server receives a new operation **O** from user **U** who has document version **i** , it transforms this operation with the operations `Oi, ..., On` in the log to obtain $O_{n+1} = T(T(...(T(O, O_{i+1})), ...O_{n-1}), O_n)$. Next, `On+1` is executed and added to the log. Consequently, the current version becomes **n+1**.

A transformation function **T** proposed by *Ressel et. al.* is as follows:

```
T(insert(u1,v,c1,p1),insert(u2,v,c2,p2)) =
    if p1 < p2 then insert(u1,v+1,c1,p1)
    else if p2 < p1 then insert(u1,v+1,c1,p1+1)
        else if c1 = c2 then noop(u1,v+1)
            else if u1 < u2 then insert(u1,v+1,c1,p1)
                else insert(u1,v+1,c1,p1+1)

T(insert(u1,v,c1,p1),delete(u2,v,p2)) =
    if p1 <= p2 then insert(u1,v+1,c1,p1)
    else insert(u1,v+1,c1,p1-1)

T(delete(u1,v,p1),insert(u2,v,c2,p2)) =
    if p1 < p2 then delete(u1,v+1,p1)
    else delete(u1,v+1,p1+1)

T(delete(u1,v,p1),delete(u2,v,p2)) =
    if p1 < p2 then delete(u1,v+1,p1)
    else if p2 < p1 then delete(u1,v+1,p1-1)
        else noop(u1,v+1)

T(noop(u1,v),O) = noop(u1,v+1)

T(insert(u1,v,c1,p1),noop(u2,v)) = insert(u1,v+1,c1,p1)

T(delete(u1,v,p1),noop(u2,v)) = delete(u1,v+1,p1)
```

We can thus implement the operational transformation in Java. Our program is tested against a series of operation sequences. We find that the program works well in the case of two concurrent editors. But problems may occur when there are more than two editors, which will be discussed in the next part.

3 Tombstone Transformation Functions

We have found that previous operational transformation fails to give a correct result in certain cases. One typical example is as following.

The problem involves three concurrent operations:

```
delete(1,0,2)
insert(2,0,'x',3)
insert(3,0,'y',2)
```

The original text is "**abc**". The document at version 3 should be "**ayxc**" but according to the previous function, it becomes "**axyc**". This problem derives from the fact that the character '**b**' should serve as a landmark to separate the two insert positions but it is erased by the delete operation.

A solution called **Tombstone Transformation Functions (TTF)** is proposed by *Oster et. al.* The idea is to keep the deleted character ('**b**' in our example) as a tombstone. The deleted character remains at its position but its content does not appear in the resulting text. Thus, there are two different texts: a **Model** and a **View** . The **Model** contains all the characters including tombstones, while the **View** contains only visible characters. According to this approach, the document evolves as shown in the following figure and the previous problem no longer exists. The Tombstone Transformation function is presented in Figure 1.

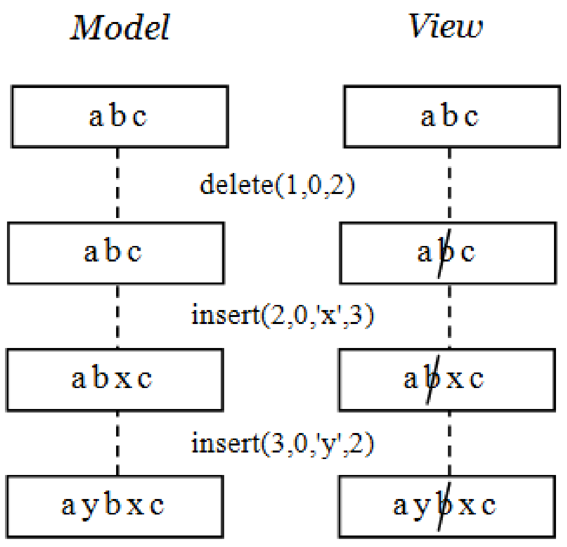


Figure 1: Tombstone Transformation Function

```
T(insert(u1,v,c1,p1),insert(u2,v,c2,p2)) =
    if p1 < p2 then insert(u1,v+1,c1,p1)
    else if p1 = p2 and u1 < u2 then insert(u1,v+1,c1,p1)
        else insert(u1,v+1,c1,p1+1)

T(insert(u1,v,c1,p1),delete(u2,v,p2)) =
    insert(u1,v+1,c1,p1)

T(delete(u1,v,p1),insert(u2,v,c2,p2)) =
    if p1 < p2 then delete(u1,v+1,p1)
    else delete(u1,v+1,p1+1)
```

```
T(delete(u1,v,p1),delete(u2,v,p2)) =
    delete(u1,v+1,p1)
```

The Tombstone Transformation function is based on the text with tombstones, which means the position parameter in the function is defined on the **Model** . However, operations provided by users are based on the visible text, namely the **View** . Thus, a conversion from **View** to **Model** has to be performed before using the transformation function. Considering that an insert operation is to be executed at the position **pos** on the **View** . The program has to skip all tombstones and find out the position right after **pos** visible characters (and skip all tombstones after this position).

4 Implementation

4.1 Implementation of the Tombstone Mechanism by a "mask"

We can use a **mask** to represent visibility of each character. The mask is a string with the same length as the *model text*, consisting only of '0' and '1'. If a character at a certain position of mask is '0', then the character at the same position of Model is visible. And if it's '1', then the character in Model is invisible (a **tombstone**).

For example, at one time, the *model text* is `ABCD`, and the mask is `0010`, therefore, the *view text* seen by user is `ABD`, as the third character, 'C', is masked.

4.2 Representation of Different Operations

We used java to implement the algorithm of operational transformation. To represent the operations, we constructed an abstract class, `Operation`:

```
abstract class Operation {
    int uid;
    int ver;
    int pos;
    abstract String execute(String text) throws Exception;//execution of operation
    abstract String executeMask(String mask_text) throws Exception;//execution on mask
    abstract Operation trans( Operation op2);//operational transformation
    abstract void check(String text) throws Exception;
}
```

The field `uid`, `ver` and `pos` represent user's id, user's local text version and the position to perform the operation. We put the 3 fields in the father class `Operation` because any operations will have these three fields .

And then we declared some functions for `Operation`:

- The function `execute(String text)` will make the desired operation on the string and return the transformed text.
- To implement the `TTF` algorithm, we added the function `executeMask(String mask_text)`, it will update the corresponding mask string, and return the updated mask string.

- To implement the operational transformation, we declared the function `trans( Operation op2)`, for two Operations `op1`, `op2`, `op1.trans(op2)` will return a new operation which is the result of `T(op1,op2)` as described in the previous section.
- Sometimes an operation is not valid, because the parameters are not properly given. For example, the `pos` parameter is not valid if it's longer than the length of current text. To check such cases, we declared the function `check(String text)`, if the operation is not compatible with `text`, the function will throw out an Exception, and the associated exception message(something like "Wrong parameter") will be displayed to the client.

All these 4 functions are declared as `abstract`, so that any class that inherit `Operation` will have to implement these functions, cause different operations correspond to different implementation of these functions.

Then we declared the classes `InsertOperation` and `DeleteOperation` (and for the extension part, the class `UpdateOperation`) which are inherited from `Operation` class, representing insert and delete (and update) operation.

Apart from the three fields inherited from `Operation`, these classes will have their own fields. For example, an `InsertOperation` will have an additional field which contains the text to insert.

And they have to implement the 4 functions declared as `abstract`, for example, when `InsertOperation` implement the `trans()` function, it will generate a new `InsertOperation` with proper parameters, instead, for `DeleteOperation`, it will generate a new `DeleteOperation`.

A point to mention is the problem in distinguishing the type of an `Operation` object **at runtime**, we used java's *reflection mechanism*: we get an operation's class name by calling:

```
op.getClass().getSimpleName()
```

Once we know the type of the operation, we can make the operational transformation correctly, so the code to implement `trans()` will contain a switch clause like below:

```
switch (op2.getClass().getSimpleName()) {
case "InsertOperation":
    //...
case "DeleteOperation":
    //...
case "UpdateOperation":
    //...
default:
    System.out.println("Unrecognized operation!!");
}
```

In TTF, we have to make a *ViewToModel Transformation* to get the correct position to perform operation, to do this, we add a static function into `Operation`:

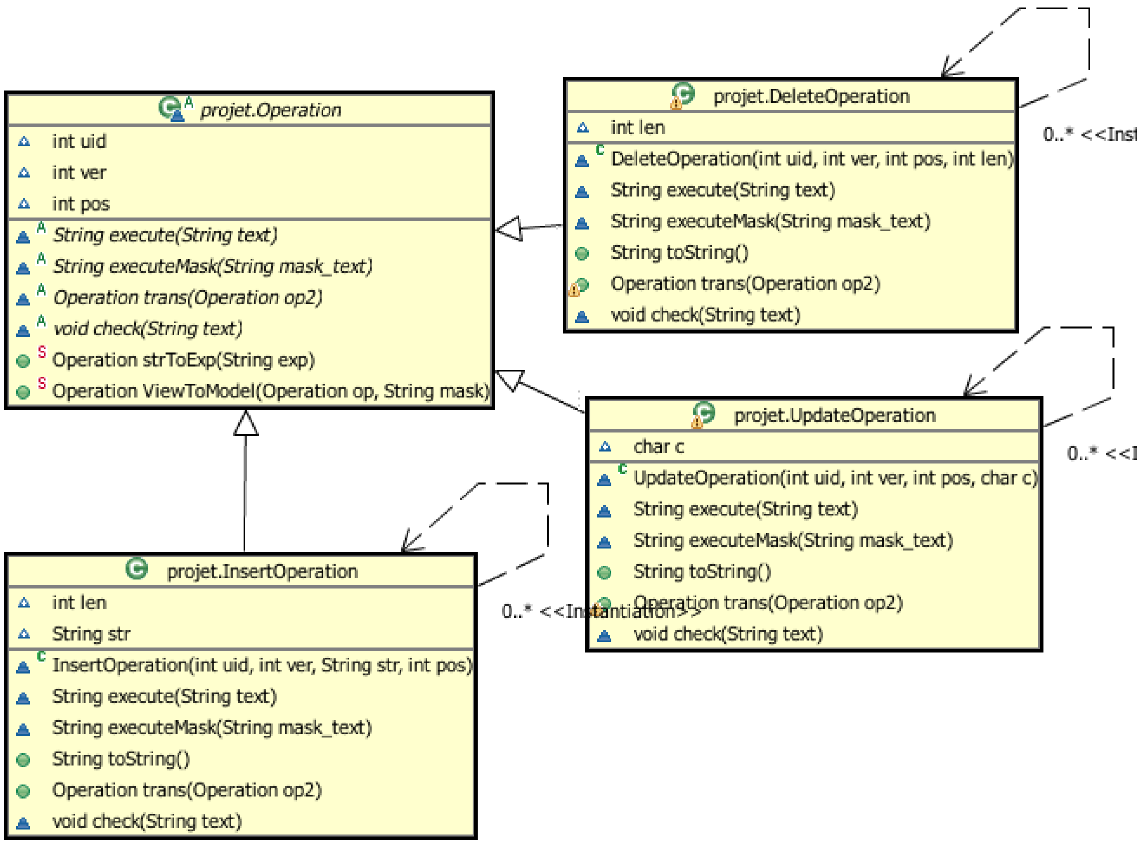
```
public static Operation ViewToModel(Operation op,String mask){
    //ViewToModel Transformation according to mask
}
```

And it is often required to parse an operation string into an `Operation` object, so we added a static function into `Operation` to perform this parse process:

```
public static Operation strToExp(String exp) throws Exception {
    //parse string into operation...
}
```

As `Operation` is declared as `abstract`, we cannot instantiate an `Operation`, so in this function, it will analyze the input string and generate a `InsertOperation` or `DeleteOperation`, or throw an exception if the string cannot be parsed.

In conclusion, the UML diagram of the operation class is in Figure 2.



projet.Operation
int uid
int ver
int pos
String execute(String text)
String executeMask(String mask_text)
Operation trans(Operation op2)
void check(String text)
Operation strToExp(String exp)
Operation ViewToModel(Operation op, String mask)
projet.DeleteOperation
int len
DeleteOperation(int uid, int ver, int pos, int len)
String execute(String text)
String executeMask(String mask_text)
String toString()
Operation trans(Operation op2)
void check(String text)
0..* <<Inst
projet.UpdateOperation
char c
UpdateOperation(int uid, int ver, int pos, char c)
String execute(String text)
String executeMask(String mask_text)
String toString()
Operation trans(Operation op2)
void check(String text)
0..* <<I
projet.InsertOperation
int len
String str
InsertOperation(int uid, int ver, String str, int pos)
String execute(String text)
String executeMask(String mask_text)
String toString()
Operation trans(Operation op2)
void check(String text)
0..* <<Instantiation>>

Figure 2: UML Diagram for Operation Classes

4.3 Implementation of Server and Client

We declared a `Server` class to represent the server side of the system. As for client side, we declared a `Client` class.

4.3.1 The Client Class

The `Client` class is inherited from `Thread`, so that different clients can execute operations separately. The UML diagram of `Client` class is in Figure 3.

A client will have it's own id, it's local text and mask string, as well as the version of this local text string.

The graphic user's interface is implemented by a `ClientJFrame` class, which is a `JFrame` (Swing). Users can input their operations and see the related informations by interacting with this JFrame (Figure 4).

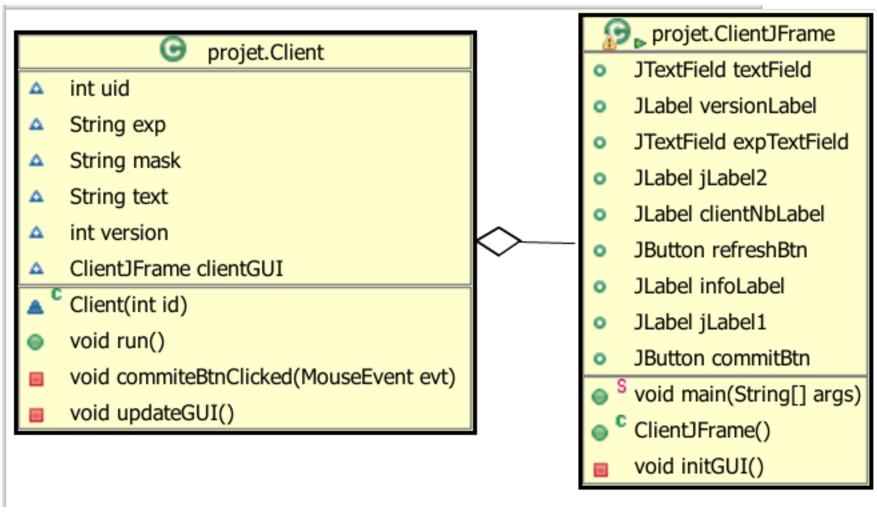


Figure 3: UML Diagram for Client Class

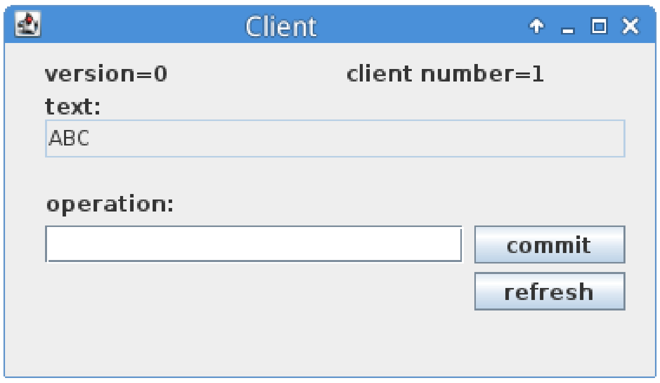


Figure 4: Client's Window

`Client` overrides the `run()` method, in this method, it will just initialize the GUI, and assign the correct handler functions associated with each button.

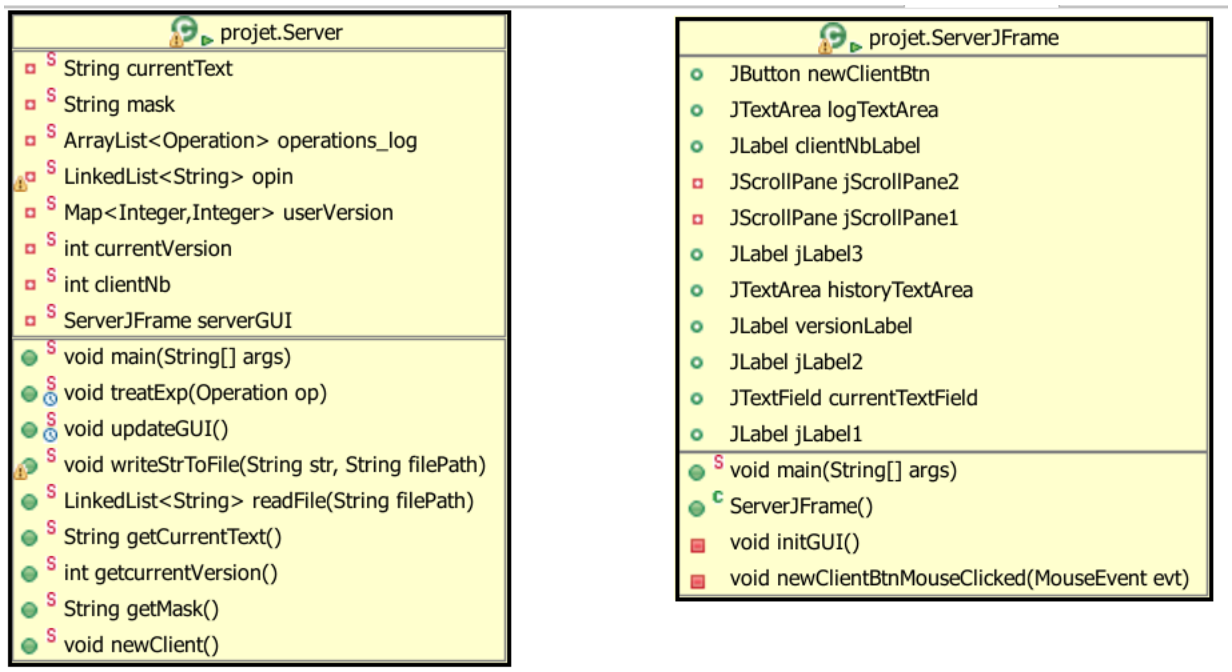
In the handler function for "commit" button (that is, when user clicked on the "commit" button), the program will get the inputted string, parse the string into an operation using the static function `Operation.strToExp()`, make the ViewToModel Transformation using the static function `Operation.ViewToModel()`, and then pass the operation to Server (using `Server.treatExp()`, discussed later). If any exception raised, we will show the exception message at the bottom of the JFrame.

4.3.2 The Server Class

In the `Server` we defined some static fields and static functions, and the entry point (`public static void main(String[] args)`) of the whole project is in `Server`. We choose to declare all the fields and functions of `Server` as static because we will have only one server running.

The UML diagram of `Server` is in Figure 5.

Apart from the common fields as `text`, `mask` and `currentVersion`, we used an `ArrayList<Operation> operations_log` to stock history of all operations performed, this will be useful when we make operational transformations. And we use `Map<Integer, Integer> userVersion` to keep a record of all clients' local versions.


projet.Server
String currentText
String mask
ArrayList<Operation> operations_log
LinkedList<String> opin
Map<Integer,Integer> userVersion
int currentVersion
int clientNb
ServerJFrame serverGUI
void main(String[] args)
void treatExp(Operation op)
void updateGUI()
void writeStrToFile(String str, String filePath)
LinkedList<String> readFile(String filePath)
String getCurrentText()
int getcurrentVersion()
String getMask()
void newClient()

projet.ServerJFrame
JButton newClientBtn
JTextArea logTextArea
JLabel clientNbLabel
JScrollPane jScrollPane2
JScrollPane jScrollPane1
JLabel jLabel3
JTextArea historyTextArea
JLabel versionLabel
JLabel jLabel2
JTextField currentTextField
JLabel jLabel1
void main(String[] args)
ServerJFrame()
void initGUI()
void newClientBtnMouseClicked(MouseEvent evt)

Figure 5: UML Diagram for Server Class

Also, a `ServerJFrame serverGUI` is declared, so as to interact with user (Figure 6).

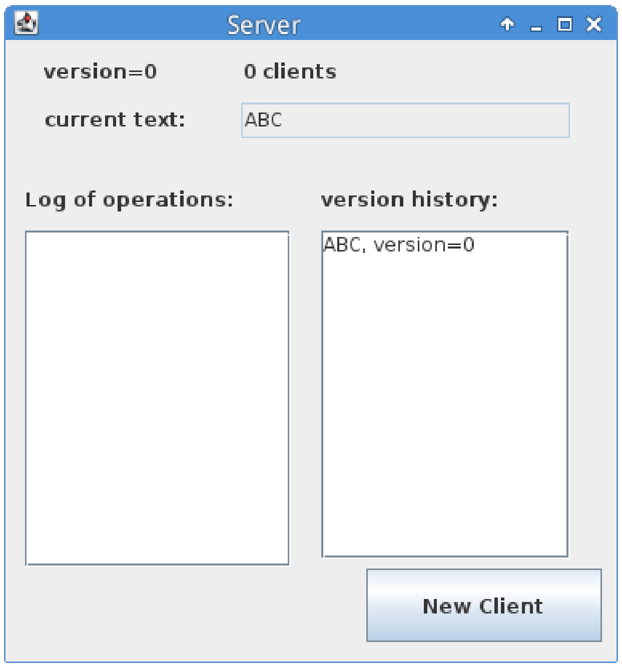

Server
version=0 0 clients
current text: ABC
Log of operations:
version history:
ABC, version=0

Figure 6: Server's Window

We defined the function `treatExp()` to treat the operations transmitted by client:

```
public static synchronized void treatExp(Operation op) throws Exception {
    //...
}
```

So when a client has prepared an operation to commit, he will call `Server.treatExp()`. In this function, the server will make a series of Operational Transformations according to user's local version and history of operations in `operations_log`. After the series of operational transformation, we get the correct operation to perform on `text`(and `mask`). As serveral client might commit to Server at the same time, we added the keyword `synchronized` to avoid potential concurrent problems.

Another function to mention in `Server` is the function `newClient()`, it is also the handler function for the "new client" button in `serverGUI`. In this function, server create a new `Client`, and let it run on a new thread by calling `newclient.start()`.

5 Extensions

5.1 Extension 1: Operation on Strings

To extend the insert and delete operations to work over strings, we made several modifications.

Firstly, we add a field `len`to the `InsertOperation` and `DeleteOperation` classes. We also changed the `char` field in `InsertOperation` to `str` which is a string. Some methods were also modified accordingly to adapt to the `String` type.

Secondly, the transformation functions were changed into:

```
T(insert(u1,v,s1,p1),insert(u2,v,p2,l2)) =
    if p1 < p2 then insert(u1,v+1,s1,p1)
    else if p1 = p2 and u1 < u2 then insert(u1,v+1,s1,p1)
        else insert(u1,v+1,s1,p1+l2)

T(insert(u1,v,c1,p1),delete(u2,v,p2,l2)) =
    insert(u1,v+1,c1,p1)

T(delete(u1,v,p1,l1),insert(u2,v,s2,p2)) =
    if p1 < p2 then delete(u1,v+1,p1,l1)
    else delete(u1,v+1,p1+l2,l1)

T(delete(u1,v,p1,l1),delete(u2,v,p2,l2)) =
    delete(u1,v+1,p1,l1)
```

We noticed that when a string is to be deleted from the document, we should actually delete segmented strings from the document. Unlike the inserting operation which can be realized by a simple insertion of a string, the string in deleting operation has to be divided into a sequence of characters. Positions of each character are transformed from the `View` to the `Model`. Then the program can call successively the classic transformation functions to perform the transformation.

5.2 Extension2: "Update" Operation

To add an update operation (which modify a character in-place), we declared the class `UpdateOperation`, inheriting from `Operation`. And we have to implement the 4 functions declared in `Operation`.

The point to notice is the `trans()` function, we have to add new transformation functions for `UpdateOperation`, and also in `InsertOperation`, `DeleteOperation`, add the case when op2 is a `UpdateOperation`.

The whole transformation table is in Table 1, as we have 3 different operations, we get $3 \times 3 = 9$ cases.

op1.trans(op2)	**insert(p2,str2)**	**delete(p2,len2)**	**update(p2,char2)**

insert(p1,str1)	if $p1 \leq p2$:
	insert(p1,str1)
	else:
	insert(p1+len2,str1)
		if $p1 < p2$:
		delete(p1,len1)
		else:
		delete(p1+len2,len1)
			if $p1 > p2$:
			update(p1+len2,char1)
			else:
			update(p1,char1)

delete(p1,len1)	insert(p1,str1)	delete(p1,len1)	update(p1,char1)

update(p1,char1)	insert(p1,str1)	delete(p1,len1)	if $p1 \neq p2$ or $p1 = p2, u1 < u2$:
			update(p1,char1)
			else:
			update(p1,char2)

Table 1: Whole Transformation Table

When two update operations are meant to modify the same character (3rd row, 3rd column in the table), we chose to subject to the user who's id is smaller. We resolve the *conflict* cases in this way, so as to ensure that committing operations by different orders will give the same final text.

For the same reason, when the intended character to update is deleted by previous operation, the new character will not appear, because `UpdateOperation` will not modify the current mask string.

6 Results & Possible Improvements

6.1 Results

We generated several clients, and used these clients to test a series of test cases (for example, several clients, possibly having different local versions of the text, doing different operations at the same position of the text), in our test cases, the system can correctly handle clients' concurrent operation requests.

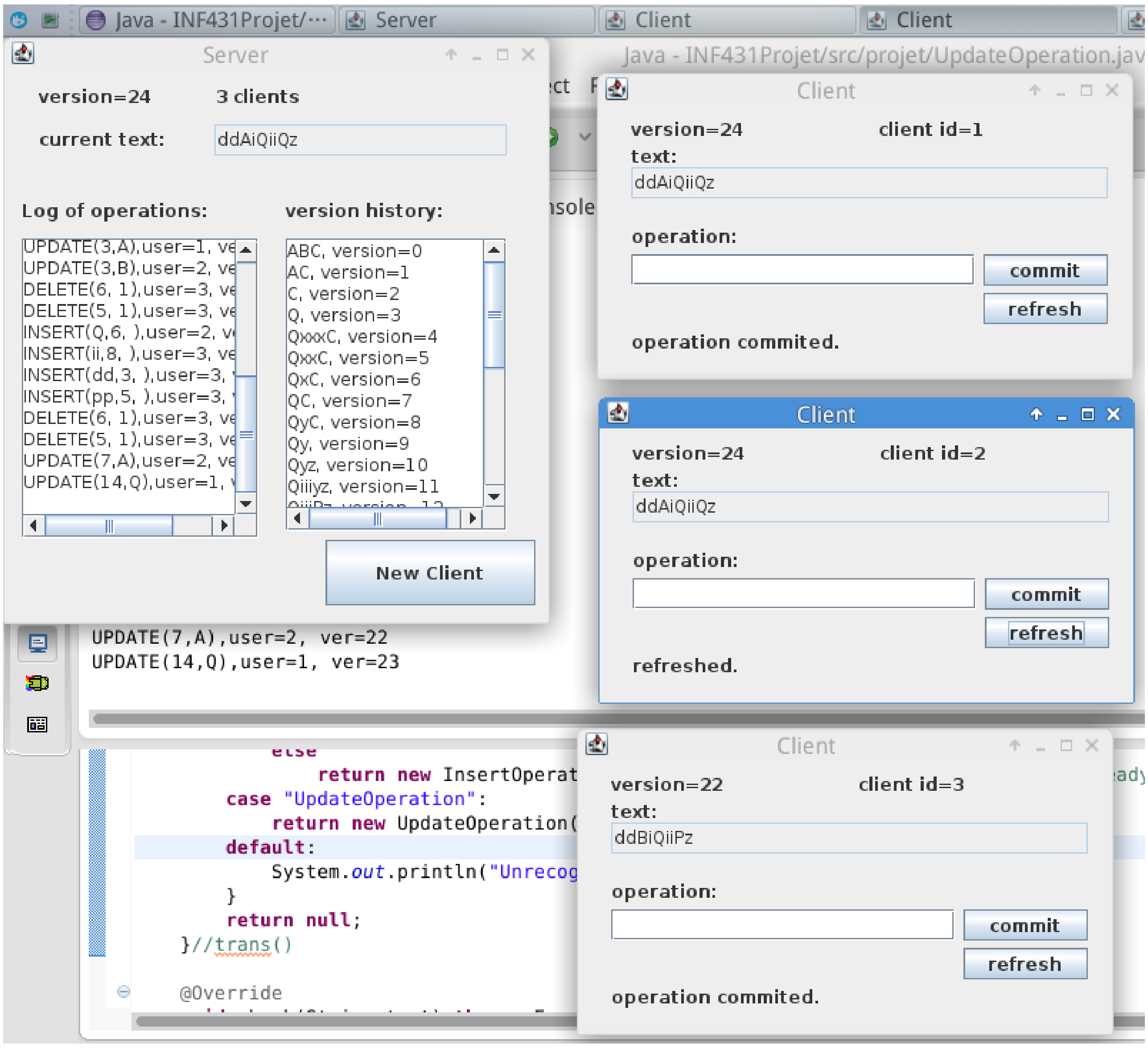
Figure 7 is a screen-shot of the test case.

6.2 Possible Improvements

We used a string as the mask, and the concatenation of strings are usually time-consuming, so we could use binary bits (like a `long` number if string length is less than 64), thus the concatenation operation can be realized by shifting the number.

As the version accumulates, we can delete some masked strings in *model text* and in *mask text* so as to save space, the test condition for deleting should be: when all clients have a version greater than **v**, then we can delete the mask chars corresponding to versions less than **v**, etc… And in the meanwhile we have to update clients' local masks.

Also, in our system, the client communicates with the server by giving certain parameters to Server's static methods, in real use, we should realize the communication by using for example a `socket` mechanism, so that the clients can run on different machines.



Server

version=24 3 clients

current text: ddAiQiiQz

Log of operations:

```
UPDATE(3,A),user=1, ve
UPDATE(3,B),user=2, ve
DELETE(6, 1),user=3, ve
DELETE(5, 1),user=3, ve
INSERT(Q,6, ),user=2, v
INSERT(ii,8, ),user=3, ve
INSERT(dd,3, ),user=3,
INSERT(pp,5, ),user=3,
DELETE(6, 1),user=3, ve
DELETE(5, 1),user=3, ve
UPDATE(7,A),user=2, ve
UPDATE(14,Q),user=1,
```

version history:

```
ABC, version=0
AC, version=1
C, version=2
Q, version=3
QxxxC, version=4
QxxC, version=5
QxC, version=6
QC, version=7
QyC, version=8
Qy, version=9
Qyz, version=10
Qiiiyz, version=11
```

```
UPDATE(7,A),user=2, ver=22
UPDATE(14,Q),user=1, ver=23
```

```
                    return new InsertOperat
            case "UpdateOperation":
                return new UpdateOperation(
            default:
                System.out.println("Unrecog
        }
        return null;
    }//trans()

    @Override
```

Client

version=24 client id=1

text: ddAiQiiQz

operation:

operation commited.

Client

version=24 client id=2

text: ddAiQiiQz

operation:

refreshed.

Client

version=22 client id=3

text: ddBiQiiPz

operation:

operation commited.

Figure 7: Test Case

7 User Guide

To use our program, run the `main` method in `Server.java`, when the server window appears, click the "new client" button to generate some clients, then, at each client's window, input operations you want to perform, then press "commit" button to commit and get the latest version of text from server.

or you can just press "refresh" button to retrieve the latest version of text from server.

The input operations are in three possible forms:

1. `insert(str,pos)`, where `str` is a string and `pos` is a number indicating the position to insert;
2. `delete(pos,len)`, where `pos` indicates the position from which to delete, and `len` indicates the number of characters to delete;
3. `update(pos,c)`, where `pos` is the position of the character to update, `c` is the new character to replace.

If input string is not correct, a message will appear at the bottom, and you can input again.