A Document Server for Collaborative Concurrent Editing

Subject proposed by Karthikeyan Bhargavan
Karthikeyan.Bhargavan@inria.fr
Difficulty: High (***)

January 24, 2014

Online document repositories such as Google Docs [1] enable multiple users to share and edit a single document. The document server merges the edits and provides a consistent view to all users. The aim of this project is to build a document server in Java that receives and executes concurrent editing operations from multiple clients and executes them in real-time.

When different users edit different parts of the document, merging these edits is easy. However, when they change the same sentence or word, the server must decide how to combine the desired changes. As a simple example, consider two users Alice and Bob concurrently editing a single word "abc". Alice inserts an extra 'x', say at position 2 between 'a' and 'b', to make the word into "axbc". We can represent this edit compactly as an *operation* insert (2,'x'). At the same time, Bob deletes the 'b' at position 2, to make the word into "ac"; the corresponding operation can be writen delete(2).

If the document server receives Alice's operation just before Bob's, and naively executes the two operations in sequence, it will first insert 'x', ("abc" →"axbc") and the delete the 2nd letter ("axbc" →"abc"). Hence, the 'x' gets deleted although Bob's intention was to delete the 'b'. Clearly, the document server would have to use a more sophisticated algorithm to merge the two edits, to obtain the final word "axc" intended by Alice and Bob.

This is just one example of the many problems that occur when concurrent accesses to shared data structures, such as documents, is allowed. There are two well-known approaches to solving such problems. In the *pessimistic* approach, Alice and Bob are not allowed to edit the same piece of text at the same time, each user must lock ("check out") the text they wish to work on and unlock ("check in") when done with their edits. In the more liberal *optimistic* approach typically used in real-time collaborative editors, Alice and Bob may edit anywhere in the text and the document server later merges their edits in a way that captures their intentions.

[1] `http://docs.google.com`

Our aim is to build a document server that implements an optimistic technique called *Operational Transformation* to merge concurrent edits to a document. This technique is commonly used in collaboration software, such as Google Wave[2], when a document may be modified in real-time by a large number of users without previously agreed-upon editing policies[3].

1 Operational Transformations

The key idea of operational transformation[4] is to modify users' concurrent operations so that they can be executed sequentially. Returning to our example, when the server gets Alice's edit insert (2,'x') it simply executes it as before, but when it gets Bob's edit delete(2), it first transforms this operation to delete(3) hence incorporating the effect of Alice's edit, and then executes the transformed operation, thus obtaining the intended result "axc".

To perform these transformations, the server maintains a master copy of the document along with additional information, such as a document *version* number and a *log* of all the operations executed on the document so far. Initially, the document version number at the server is 0 and the log is empty. After n operations, the version number is n and the log consists of n operations $O_1, \ldots, O_n$. Each user has a local, possibly out-of-date, copy of the document and edits it, sending new operations to the server. When the server receives a new operation O from user U who has document version i, the server transforms this operation against the operations $O_{i+1}, \ldots, O_n$ in the log to obtain $O_{n+1} = T(T(...(T(O, O_{i+1})), ...O_{n-1}), O_n)$. It then executes operation O_{n+1} on the current document, updates the version to $n+1$ and adds O_{n+1} to the log. Hence, for example, by transforming an operation O against two operations O_1, O_2, we would generate $O_3 = T(T(O, O_1), O_2)$.

Different collaborative editing systems use different transformation function T and different parameters for the operations O. In the simplest case, we only consider operations to insert and delete single characters, written with parameters as:

- 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 pos is the position from which a character is to be deleted.
- noop(uid,ver), an identity operation that does nothing to the document.

For such operations, a commonly used transformation function T (defined by Ressel *et. al.*) is as follows:

[2] `https://wave.google.com/wave/`
[3] See `http://www.youtube.com/watch?v=3ykZYKCK7AM`
[4] `http://en.wikipedia.org/wiki/Operational_transformation`

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)

In our example, the document at version 0 has content "abc". Suppose Alice has uid 1 and Bob has uid 2. Initially, both have version 0 of the document. So, Alice's operation is written **insert**(1,0,'x',2), and Bob's operation is written **insert**(2,0,2). The server executes Alice's operation as is, resulting in document "axbc', and transforms Bob's operation to T(**delete**(2,0,2),**insert**(1,0,'x',2)) = **delete**(2,1,3) (according to the third clause of T above). Hence, at the end of these two operations, the document version is 2, the log contains:
insert(1,0,'x',2), **delete**(2,1,3)
and the document content is "axc".

2 Programming Goal

Part I You are to write a Java program that takes as input a text file containing the initial document contents and inputs from the command-line a sequence of operations, e.g.,

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

Here, user 1 issues one operation, and user 2 issues two operations on version 0 of the document, then both receive the new version 3 of the document and continue sending edits. Your program must then execute each of these operations in order, transforming them as necessary. The output of the program is a text file containing the final document.

Well-formedness Your program should check that the sequence of operations is well-formed, so that when any operation cannot be executed on the current document, the server flags an error. For example, your program should check the following.

- The position parameter is each operation is valid. For example, for inserts, the position argument must be greater than zero and less than or equal to one plus the length of the document.
- For each user u, the sequence of operations uses strictly non-decreasing version numbers. That is, if **insert**(u,v1,c1,p1) is followed in the sequence by **insert**(u,v2,c2,p2), then v1 <= v2

Part II The transformation function defined above fails to reflect the users intentions in the following case. Assume Alice, Bob, and Charlie have uids 1,2, and 3, respectively. The initial document is "abc" as before. Alice proposes the edit **delete**(1,0,2). Bob proposes the edit **insert**(2,0,'x',3). Charlie proposes the edit **insert**(3,0,'y',2). So, the document at version 3 should be "ayxc" but according to the function T above, it becomes "axyc"

Fix the transformation function, and extend the document details kept by the server, if necessary, so that the server works for all sequences of well-formed operations sent by any number of clients. You may assume that the list of users is fixed in advance and the uids of all users is known to the server. For example, you may adapt some of the transformation functions described in Section 5 of *Proving correctness of transformation functions in collaborative editing systems*, by Oster *et. al.*[5]. You will be expected to evaluate your final solution against other possible choices, and show why your solution works better.

Part III To test the effectiveness of your server, use Java threads to implement a single document server that runs in parallel with an arbitrary number of concurrent clients. Each client will first refresh the document by retrieving the current version from the server, then accept a series of commands on behalf of a single user, send these commands, refresh the document again, then repeat. The precise UI of these clients is left for you to design. The key idea is that different client threads should be able to edit the document concurrently, and the resulting should be the "correct" one.

[5] `http://www.loria.fr/~urso/uploads/Main/ostertochi05.pdf`

Efficiency The efficiency of the document server typically depends on the number of transformations that need to be performed on each operations before executing it. This in turn depends on the length of the log. Your program should try to compress the log as much as possible by deleting old operations that are no longer needed. Moreover, an operation should be transformed only when necessary. However, be careful not to sacrifice correctness for efficiency. Your program's primary concern must always be to produce the correct document intended by all the users.

3 Evaluation

Your submission will be judged by three parameters.

1. The program will be tested against a wide variety of documents and operation sequences. In each case, it should produce the expected document at the end.

2. The underlying transformation function should be clearly documented in the code, so that a reader can see how it works and how to modify or extend it.

3. The program will be tested for efficiency by running it over large sequences of operations. You should be able to explain and reason about how much time the server is expected to take over each operation.

Project presentations may be either in French or in English, as you prefer. You will be given test suites and other supporting material in advance, and you are encouraged to ask for guidance on different aspects of the project.

4 Extensions for Extra Credit

- Extend the insert and delete operations to work over strings, such as "xyzw", rather than characters such as 'x'. Hence, insert operations now have the form **insert**(uid,ver,string,pos) and delete operations have the form **delete**(uid,ver,pos,length). You will need to extend and implement new transformation functions for these operations. (Difficulty: **)

- Extend the operation set with an update operation that enables a character to be modified in-place. For example, on the string "abcde", the operation update(1,0,3,'x'), changes the string in one step to "abxde". You will need to design and implement new transformation functions for this operation. Updates introduce the possibility of *conflict* where there is no single final document that both users will be happy with. In such cases, you are free to resolve the conflict by choosing any one of the documents intended. (Difficulty: ***)