A Collaborative Editor

Baoxiang Yang, Haotian Wu, Haoyu Zhang

April 21, 2017

1 Introduction

In this project, we plan to implement a collaborative website editor, which supports a group of users editing files at same time.

2 Structure

We used Django framework to build our application. To alleviate server load, consistency algorithm is running on the client side. And the servers are only in charge of the database. To enable fault tolerance, we used a backup server. And a shared MySQL database is used to store persistent data.

3 Operational Transformation

In our project, different users can edit a same file at same time. This is supported by a operational transformation algorithm. Our editor is based on the open source editor CodeMirror. (`https://codemirror.net`) With the help of CodeMirror's API, all changes made by users will be caught by a event listener, and is encapsulated in JSON format:

$$\{from : \{line : line\#, ch : character\#\}, to : \{line : line\#, ch : character\#\}, text : [str1, str2, ...]\}$$

When such a change is made by user, our algorithm will record it in a list called *notCommit*. Next, one of the following two things may happen:

1. A heartbeat timer expired, and trigger a heartbeat event. In this case, the client will try to send all the content of *notCommit* list to the server along with a version number which indicating the last synchronized version of the file on this client.

2. A operation list from other clients arrives. Then this client will first do a transform on new arrived operations, and then apply those transformed operations. The operational transformation function will also update the *notCommit* list, which we will discuss in the following article.

To illustrate how our operational transformation work, let's assume the original file is 'Hello world', and there are two users. The first user wants to insert 'Hi' at position 0, and the second user wants to insert '!' at the end of the string. In our example, assume the first user receives the operation list from the second user, which is:

$$\{from : \{line : 0, ch : 11\}, to : \{line : 0, ch : 11\}, text : ['!']\}$$

But on his screen, the string is 'HiHello world' now. Without the operation transformation, the output will be 'HiHello wor!ld'. So, what operational transformation does is adding an offset to the newly received operation, which result in:

$$\{from : \{line : 0, ch : 13\}, to : \{line : 0, ch : 13\}, text : ['!']\}$$

As shown above, an operational transformation will take two operations a and b and produces two new operations a' and b', where $b'(a(file))$ equals $a'(b(file))$. Since there are two kinds of operations, there are 4 possible cases:

1. insert and insert.
2. insert and delete
3. delete and insert
4. delete and delete

For those 4 kind of combinations, we have 4 corresponding transformation functions. And after we transform and apply new operations, the version number of the file will also be updated. As we discussed before the *notCommit* list is also updated and can be applied directly to the new version. So when the client send the refreshed operations to the server, the server can apply those operations directly.

4 Server Functions

Since operational transformation is done on the client side. All the work a server need to do is simply dispatch operations. When a client send a list of operations to the server, the server will broadcast this list along with the current file version number to all clients, and operational transformation algorithm on the client side will make sure the files are consistent. If a client missed several operations, it will send a wrong version number to server, and the server will notify it by sending the missing operations to it.

5 Fault tolerance

Our simple fault tolerance mechanism is inspired from the paper The Design of a Practical System for Fault-Tolerant Virtual Machines. We assume that there is no network partitioning between server and clients. If client cant receive response from server, it will be considered as server failure.

Primary/Backup Server:
When we startup the primary server, we run a backup server at the same time on another physical server, which is connected to the same mysql database server that the primary connects to. That means primary server and backup use the same database. When primary server is handling requests from clients, backup server will do nothing but just wait. Once if primary server fail, clients requests will also fail. Clients will give another several try and then will be directed to the backup server. When backup server receives the first request, it will load data from database and become to the primary server.

Shared Database:
Since collaborative editor has to respond as soon as possible, separate database is not a good implementation because it will slow the whole communication process. Primary server has to send the received log to backup server and wait the response from backup server. However, if we use shared database, we dont need communications between primary server and backup server anymore so that server will respond much faster.

6 Discuss

Our collaborative editor enables several users editing on a same file, and it can synchronize the file on each client in seconds. To get a better user experience, we also supported several editor mode such as vim, sublime and emacs.