1 Reference

Sun, C., Jia, X., Zhang, Y., Yang, Y., & Chen, D. (1998). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Transactions on Computer-Human Interaction, 5(1), 63–108. https://doi.org/10.1145/274444.274447

2 Implementation

2.1 Algorithms

2.1.1 Text Editing

To solve problems concerning document state divergence, a somewhat modified version of the Generic Operation Transformation (GOT) control algorithm described in the document in 1 Reference will be implemented. The modifications are due to the document describing a peer-to-peer model, while SD will be client-server. The reasoning for this decision is discussed in section 3.1 Peer-To-Peer vs. Client-Server.

The client-server version of the GOT algorithm is based on the assumption, that all users receive all changes in the same order, as users receive changes from the server only and the server sends the changes to all users at the same time (in the same order).

Due to all users having the same ordering of incoming changes, causality-violation described in the document in 1 Reference will never occur. There is also no need for the state vector. The state vector as described in the document contains one number for each user and is used to correctly describe what other changes the change with the state vector relies on. Because the ordering of changes is the same for all users, it is sufficient to send what change the user received last. The other users receiving this change can then apply it by rolling back their document state to the stated last change, apply the change, and applying all the reverted changes back.

It should be noted that this approach only deals with situations where the sent change relies on an earlier document state than that present at other users, meaning that it is assumed that all users receiving the change already have all other changes the received change relies on applied. This assumption is correct, as all users receive changes in the same order, meaning all the required changes will arrive to users prior to the relying change.

2.2 Server Structure

There will be two servers managing all requests sent by users. The first one will manage users connecting to SD. This server will send the user the initial webpage using server-side rendering. The second server will manage all other transactions, except user authentication, which is managed by an external server.

2.3 Client-Server communication

2.3.1 Communication Protocol

Client-server communication will be realized using WebSocket. After the user opens a folder or document, a WebSocket connection will be established, allowing the user to send and receive changes made.

2.3.2 Transactions

After a client requests a document to be loaded (assuming he has the right to view the document), the server sends him the appropriate content.

Changes made to the document by users will be buffered and the buffer will be sent to the server in set intervals (for now assumed an interval of half a second).

The server will immediately send any change received to all users (it might be a good idea to buffer them, this will require further testing). The clients will filter out their own changes (sending their own changes back to the client is necessary to prevent document state divergence, for now, might be a target for change later) and apply the modified GOT algorithm to update their copy of the document. The server will apply the changes to his own working copy (using the modified GOT algorithm).

The server's working copy of the document is necessary for newly joined users to receive a current copy of the document from the server.

2.3.3 Transaction contents

The data sent by the user to the server will only contain data necessary for the modified GOT algorithm and the changes themselves. The modified GOT algorithm requires two pairs of numbers, one pair being the authoring user's ID and the changes serial number, the other pair is the ID and the serial number of the last change the user received.

The server will then broadcast the same data to all users.

2.3.4 Transaction Size and Bandwidth

As stated above, one transaction contains four numbers and the changes themselves. The changes will contain instructions to skip certain content, delete some amount of content, or add content. Content skipping will be denoted by positive integers, Content removal by negative integers, and content insertion by the inserted text.

Assuming the changes are sent to the server every half second and the average typing speed of professionals is about 400 characters a minute, an average user should send less than 5 characters. Including skipping and deletion instructions, the final transaction including WebSocket overhead should not exceed 128 bytes.

Regarding pasting content to the document, it shall be assumed that the average user pastes up to 1000 characters (200 words) per second and not all users paste all the time. Assuming UTF-8 encoding, 1000 characters translate up to 4000 bytes, rounded up to 4 KiB. The size of instructions other than inserting and WebSocket overhead should be negligible, as it is assumed that every UTF-8 character will take up 4 bytes, which will most likely not be the case. The size of such a transaction should therefore be around 4 KiB. As the assumption states 1000 characters per second, the client should send around 4 KiB per second.

Assuming there are 100 active users and 10 users paste 1000 characters every second, the server would have to have at least $10 \cdot 4096 + 90 \cdot 128 \cdot 2 = 64000$ B/s = 62.5 KiB/s $\doteq$ 0.5 Mibit/s download speed.

All this data has to be sent back to all 1000 users, meaning the server has to have at least $\sim$0.5 Gibit/s upload speed.

It should be noted, that this works on the assumption, that there are over 10000 characters added each second. If there was no possibility of pasting, the resulting required server download speed would be $\sim$0.2 Mibit/s and upload speed would be $\sim$0.2 Gibit/s.

2.4 Text editor design

The text editor would be a set of React elements representing individual rows. The benefit of having rows as individual components instead of the whole text being in a single element is that only some rows will need to be rerendered, not the whole element containing all content. The fact that there is no requirement for maximal row width means that inserting characters at the start of the document will not directly interfere with the rows below, not forcing them to rerender.

3 Design decisions

3.1 Peer-To-Peer vs. Client-Server

3.1.1 Peer-To-Peer

The benefit of having a peer-to-peer design is reducing the required server upload speed. The server download speed will still have to remain relatively similar to a client-server implementation, as the server has to have a current working copy, that it can send to newly connected users.

The drawback is that the users will have to communicate among themselves, meaning the full GOT algorithm has to be implemented, including state vectors. State vectors have a size equal to connected users, of which there might be 1000. If the vector contained uint16 values, the vector would take up ~2 KiB. This vector has to be sent along with every change, which has to arrive to every user. If a user were to send a change to all other users every half second, that would result in a required user upload speed of $2 \cdot 2 \cdot 1000 \doteq 4$ MiB/s = 32 Mibit/s. This requirement only counts the size of the state vector, not the change itself.

This upload speed cannot be required from users, as it is too high. A solution to this would be to send the change to only a fraction of the users, who in turn would send it to the rest of the users.

Let's assume there were only 931 users, 100 of which active. A user sending a change would send it to only 30 other users, who in turn would send it to 30 different users, resulting in all 931 users receiving the change. This would mean that sending a change would require an upload speed of $2 \cdot 2 \cdot 30 = 120$ KiB/s $\doteq 1$ Mibit/s.

However, this requires a total of 31 Mibit/s in combined user upload speed per user making changes. As there are 100 active users, that would result in a required combined user upload speed of ~3 Gibit/s, which divided among the 931 users would result in an average upload speed requirement of ~3 Mibit/s (assuming there was a way to distribute changes effectively, without any bandwidth loss).

3 Mibit/s might be too much for a text editing app, considering online streaming services including audio and video transmission usually require around 1-2 Mibit/s. The 3 Mibit/s do not include the actual changes sent between users (only state vectors), which might increase this number considerably. There is also the problem, that the required combined user upload speed increases quadratically with the number of users, meaning SD would not be easily upscalable for higher quantities of users. The design of sending the changes to only a fraction of the users, who then send it to the rest, also creates the problem of combined latency, as most users will receive the change from an intermediary, combining the author's and intermediary's latency.

3.2 Client-Server

The benefits of client-server design are low user upload speed requirements. As calculated in 2.3.4, a user who would paste 1000 characters per second would only require an upload speed of 4 KiB/s = 32 Kibit/s, which is substantially less than the peer-to-peer design. Another implementation benefit is that only a part of the full GOT algorithm has to be implemented, as the problem of causality violation never occurs.

The drawback of this design is a relatively high required server upload speed, calculated to be 0.2-0.5 Gibit/s. This should, however, not be much of a drawback for relatively big institutions.

3.3 Conclusion

Using the client-server approach is believed to be the better choice, as it puts minimal constraints on the users while requiring reasonable bandwidth from the service provider.