Collins Sirmah
Justin Lee
Dec 4 2015

A Collaborative Blockly Editor

Introduction

In this project we aimed to create a collaborative Blockly editor with real time chat. Blockly is a client side javascript application where users can drag blocks to create programming logic. The blocks can then be "compiled" to Javascript, PHP, Python or Dart. Blockly is an open source project from Google, and can be customized to create educational tools, custom language creator and games. Our platform aims to create a unified

High Level Explanation.

Our initial design comprised of sharing the document state between different clients. For example, we would capture the state of a block and then reflect it to the other clients via a client server model. However we realized that this model would not work because Blockly is a "sandboxed" javascript application. Thus the browser would never capture the events occurring within a Blockly client, and as such we would never have a way of getting the different document states. The other approach was to take the control of the cursor and reflect it across the different clients. The "owner" of the cursor would have been the first client to take control of it. However, we argue that this kind of an application is not collaborative.

We realized that Blockly exposes a toXml function which generates an xml of the Blocks. This informed our second design iteration, which involved locking a file as it is edited by a particular client, and then when the client is done with it, release the lock and give another client the lock access. The main challenge posed by this was that as the number of clients increases, it becomes hard to maintain and synchronize locks. Furthermore the network latency affects the synchronization of the locks.

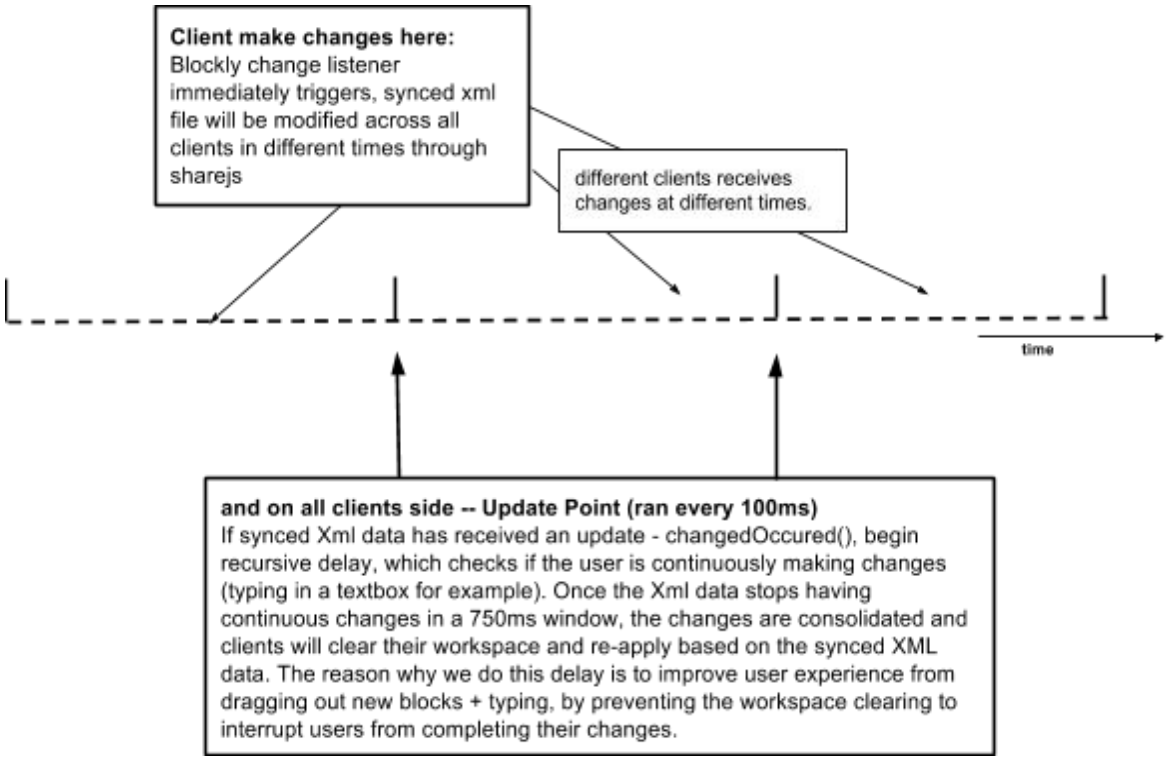
Our final solution involved use of Operational Transformation on xml files generated by Blockly. Operational transformation is a process where small changes(e.g edits) from a client are resolved in a manner that everyone will end up with the same document. Thus for example if we are to edit ABCD and client A sends an *insert[a, 3]*, while client B sends a *delete[c, 3]* at the same time, then the two of them might end up with a different document depending on the winner of the race condition. In operational transformation, operations are defined e.g *insert[elem, pos]* and *delete[elem, pos].*

We also had to define a two-way communication channel. In its current format , HTTP supports only client-initiated communication. In this application, used websockets, a TCP-based communication channel, which maintains a bidirectional persistent connection(Initiates a handshake with the server and the server in return initiate another handshake) and uses HTTP headers(uses the additional upgrade header field).

Current Implementation

Our current implementation uses sharejs to maintain a convergent XML document state. This is used as the single point of reliable transform when a client is updated with the most recent workspace "snapshot".

The only interface we could use from Blockly is workspace.addChangeListener which detects any change that occurs on the Blockly workspace. When this occurs, we generates a textArea xml which is then sent to the server. However the defined transport protocol also expects key input for for it to expect any change, and as such we initiate a fake key press programmatically every time there is a change. This comes with the consequence that the application cannot be supported by Internet Explorer, and that the the server does not have the knowledge of current position of a cursor.



Cases we handle

Blockly

Our implementation works for basic real-time updating on all concurrent clients. Node.js servers, the backbone in which sharejs runs on, is extremely efficient and scalable. When client a drags a block and places it down, different clients sees the change after a relatively short delay. This syncing method via sharejs is versatile as it handles all block types included in the core blockly module. Theoretically, if one were to include other custom block types built using the custom factory, it would work just the same. (https://developers.google.com/blockly/custom-blocks/block-factory?hl=en)

Chat

Our chat system is implemented using socket.io. Unlike sharejs, it doesn't guarantee order and state syncing. Chat messages might be in different order to different clients. Chat rooms are confined to the subdirectory of the root URL, meaning that each collaborative

workspace has its own chat room for communication. This is done via socket.io's namespace/rooms system which allows for some great abstraction for handling different type of messages (connection, disconnection, room joining and chatting). A lot of other workspace events can also be emitted via socket.io, making it a great module to include overall.

Cases we currently cannot handle

If two users are making a change in the same time; for example, client A is dragging an existing block and client B is pulling a new block from the menu, whoever releases (and update the XML) first will in turn emit a change event in the other client's text area, forcing a change (workspace clear and redraw using XML) on their end and cancelling the change/insertion they were originally working on. Although this is not a scenario that completely breaks the usability of our program, it is a big limitation. It is not collaborative if it is simple control switching. Imagine a scenario where Google docs offers only real time client side viewing but not bug-free concurrent editing.

Why?

The reason why this occurs right now is because each change is not a modification of the XML file, but a complete overwrite of it. ShareJS cannot parse through an entire block of text and figure out what has changed since we do not have the knowledge of textarea cursor position and Blockly change listeners exposes nothing about which block is being changed/added/attached. This makes it difficult for us to leverage the operation transform that shareJS offers us.

Furthermore for us to reflect the new change on Blockly, we have to clear the current workspace before importing the new xml. In case a user was dragging a block when this happens, then the block they were dragging would return to the start position. However all this happens in the order of milliseconds and thus a user might never notice.

Improvements

One of the ways which we could have created a better editor would have been by modifying Blockly's source. However Google's Blockly supports a large number of applications and our modifications would have possibly made our application incompatible with them. Indeed when designing the application, one of the constraints we set ourselves was not to break the abstractions of the interfaces we use.

However we have identified how we could further improve the application within the constraints that we set.

1) Using ShareJS JSON object operation transformation package instead of text. There would be overhead of xml to json(and vice versa) conversions. In large Blockly programs, this would create a lag that would be felt by the users.
2) Determining changes by parsing the XML data by recursively traversing the tree based data structure to determine what is being changed by the client. This would be

a lot easier when Google[1] implements better event listeners (MUCH less computationally expensive than to just brute force search through a tree to determine one singular type of change on any particular node/branch (move, insert, remove)). This would also entail implementing operational transformation on XML as shown below.

3) Now instead of overwriting the textbox of XML, we can call shareJS's methods on applying operational transformation on an XML tree.
    a) methods like insert(Block's XML data, {path}), where path is an array of tree nodes one have to traverse through to reach the point of action.

** NOTE: This modified scheme still require a way for Blockly to update the workspace without the need to (clear+reapply), which cancels the client's current control. In other words, the workspace needs to be able to dynamically change without affecting the client's freedom of actions.

Another Problem

Another problem right now is that Blockly does not assign unique ID to individual blocks, meaning shareJS cannot reliably know where exactly to traverse in the {path} variable, if there are two of the same type of blocks under the same parent node.

Example:

```
if(){
        some_subtree-a;
}
if(){
        some_subtree-b;
}
{path} = [ if_block, some_node, ….]
```

Our initial operations which we defined are listed below based on the Blockly's block. Insert(block, {path}), delete(block, {path}). This wouldn't work until Google complete the development of their UUID system, which will identify each instance of blocks with a unique identifier.

Conclusion

One of our worries at the start of this project was that we would not complete it as there was no clear solution and we had to define ours. Thus our goal was to make sure that we have at least a working implementation. The most work (and learning) was on how to build it, and once we figured out what to do most of the implementations became trivial.

While this application comprised of dragging blocks and reflecting the changes to different clients, it made us think of the notion of state in a networked application and how we can create a "collaborative"/concurrent application from the perspective of a user.

---

[1] Blocky developers released a detailed plan on a new version which was to implement collaborative editing. The plan can be found at and involves also the creation of a UUID for Blocks. https://docs.google.com/document/d/1KyF6X7yrc3phPqLGzpPdngLMMzZ0jvXOFK5W43IAwbY/edit

While most of the applications till now have focused on text editing, collaboration can also be extended to other kinds of applications. An example of such applications include design tools(e.g AutoCad), medical tools(diagnosis over distance) and even simple everyday tools such as trip planning/booking an uber ride.

However we argue that such applications might prompt a redesign in how we handle state in different computing systems. And the same way we include interfaces such a pthread and fseek in linux, we have to find a way of including interfaces such as Operational transformation in the linux kernel. This might enable us to create more applications that are not just limited to the “web”. Furthermore this might make us rethink the design of network systems e.g how to create communication channels that better handle concurrent/shared data. An example is while a CDN might take content closer to a user, that benefit might not be felt in the case of a collaborative system where two users are on two different parts of the world consequently negating the benefits of a CDN.