Real-time Collaboration in PlaySketch

BENG TECK WEN ANDREW
School of Information Systems
Singapore Management University

Abstract - PlaySketch is an informal web-based animation tool that targets video game designers and extends the capabilities of K-Sketch. One of PlaySketch's key features is real-time collaboration, allowing teams of users to collaborate on the same animation, or a single user to work across multiple devices or windows. A key technical challenge with collaborative tools like PlaySketch is ensuring client-side interface responsiveness while maintaining consistency across all sites. To assess a suitable software architecture for PlaySketch, this research reviews Operational Transformation (OT), the underlying technology that enables real-time collaboration in web-based productivity tools like Google Docs and EtherPad. This research also highlights technical requirements for PlaySketch's HTML5 canvas system to be compatible with OT. The result of this research is a prototype sketching application built on top of open source OT and HTML5 canvas libraries that demonstrates a potential event-driven software architecture for PlaySketch.

Terms: Animation, real-time, collaboration, Operational Transformation, HTML5 Canvas, sketching

1. INTRODUCTION

Informal animation tools allow users without prior knowledge of computer aided animation to create simple animations. Previous work and research on the K-Sketch project highlighted significant gaps in existing animation tools that prevented novice animators expressing their ideas with sufficient speed, flexibility and simplicity [1]. In addressing these limitations, the intuitive pen-based interface of the K-Sketch animation tool allows a stylus not only to be used for sketching, but also to manipulate sketched objects on the rendering surface. In animation mode, K-Sketch implements animation through demonstration by recording these manipulation gestures and playing back the recorded sequences as animations. These intuitive sketching and animation techniques allow a wide range of novice animators, including primary school students and teachers, to create animations easily and rapidly without compromising on some of the more complex features found in formal tools like Adobe Flash.

While K-Sketch has validated more intuitive animation techniques for individual novice animators, PlaySketch will be a new web-based tool that extends existing K-Sketch features to allow teams of users to collaboratively design interactive video game prototypes [2]. The animation by demonstration functionality of K-Sketch will be extended to enable programming of game logic by demonstration. For example, a sketched object on the rendering surface could represent a game character. Through touch gesture manipulations, the PlaySketch system will be able to learn environment influenced behaviours like the effects of gravity or physical obstacles, or expected behaviours when certain game control buttons are activated. The programming by demonstration feature in PlaySketch therefore attempts to bridge the gap between games designers and developers, when interactive game dynamics are not easily inferred from traditional static wire-frames and storyboards.

The goal of this research is to identify suitable technologies and an architecture that will enable efficient and maintainable real-time collaboration features in PlaySketch. Besides the ability of teams of designers and developers to collaborate on shared game prototypes, real-time collaboration technology will also allow individual users to extend their workspace by working on the same prototype across multiple devices or multiple browser windows. In the extended workspace mode, users could use a tablet device for sketching and character manipulation, with the combined use of a desktop machine to assign keyboard stroke controls to specific character behaviours. This paper first explores Operational Transformation, an underlying real-time technology for concurrency control, and subsequently presents a concurrent sketching web application that serves as an early prototype for PlaySketch. The open source technologies and software architecture that were used to build this prototype will also be discussed.

2. BACKGROUND - OPERATIONAL TRANSFORMATION

Originally pioneered by Ellis and Gibbs in 1989, Operational Transformation (OT) is currently the most extensively used concurrency control technology used in modern web-based collaborative tools. Some OT enabled editors include Google Docs and EtherPad. Early implementations of OT algorithms focused on solving concurrency and consistency issues in text editing groupware systems [3]. An example of such an issue is shown in Figure 1, where two users concurrently make changes to a string of text. Each change a user makes to the shared text string is an *operation*, which is either an insertion or, as shown in this example, a deletion. The concurrency issue here occurs when the *operation* “Del 6” is propagated from Person 1 to Person 2's text editor. Because Person 2 has already removed the third character from the string in *operation* "Del 3", "Del 6" cannot be achieved as there are now only five characters left.

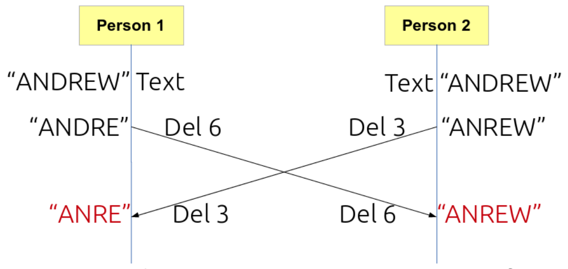


Figure 1: Propagation and execution of operations "Del 3" and "Del 6" in the text editors of Person 1 and 2 respectively.

OT algorithms introduce the concept of *transforming* any concurrent incoming operation from other sites against the *operation* that has been executed locally. In the revised example shown in Figure 2, the *transform* function implemented by OT in Person 2's editor takes both "Del 3" and "Del 6" *operations* and computes a new *operation*, "Del 6" transformed by "Del 3", which is understood by the editor as "Del 5". The same process occurs in Person 1's editor, where "Del 3" *transformed* by "Del 6" still results in "Del 3". Both *transformations* eventually result in the documents on both editors converging to the same *state*, i.e. having the same string characters. Furthermore, OT algorithms must be able to handle incoming concurrent *operations* from any other user, e.g. Person 3. Figure 3 shows Person 3's local *operation* "Ins 7, T" propagating to the editors used by Person 1 and 2. The two editors have to ensure that *transforming* "Ins 7, T" by their preceding *transformed operations* ("Del 3" and "Del 5") both result in syntactically equal *operations* on both sites, i.e. "Ins 5, T". The two examples of OT shown here outline the two important Convergence Properties (CP1 and CP2) governing consistency-related correctness requirements in OT [4].

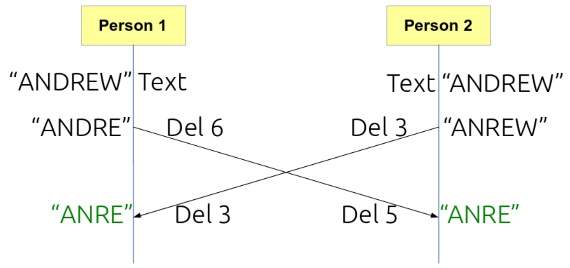


Figure 2: Operation propagation with transforms. Person 2's OT enabled editor produces the operation "Del 5" as result from transforming "Del 6" against the locally executed "Del 3" operation.

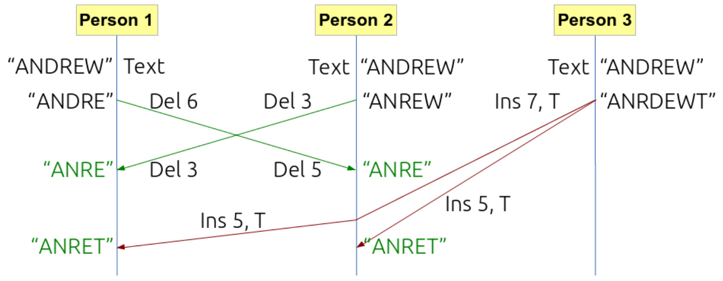


Figure 3: Operation "Ins 7, T" from by Person 3 is transformed against the previous operations on both receiving editors. The transformed operations are syntactically equivalent, and demonstrates CP2 (red arrows). CP1 is seen in the previous example (green arrows).

Unlike the more traditional solution of obtaining locks from a central server to edit certain resources [3], OT allows any remotely connected user to make changes on their local copy of the document first, before propagating their respective modification to the other user's editor (Figure 4). This technique is known as optimistic concurrency, a key feature of OT that allows concurrent editing without the trade off in responsiveness that is experienced in lock requesting. Further work on the Jupiter collaborative system, and more recently on the Google Wave protocol describe how optimistic concurrency with OT can be implemented in client-server protocols that support collaboration between any number of clients [5][6]. Following the previous example using a client-server model in Figure 5, Person 1's text editor serves as the client (from Person 1's perspective) that receives *operations* from all other clients (e.g. Person 2 and 3) via a server. Figure 5 also shows how a consistent converged document state is achieved through transformations of client and server *operations*. All possible document states resulting from different progressions of client and server *operations* can be represented in a *state* space matrix shown in Figure 6.

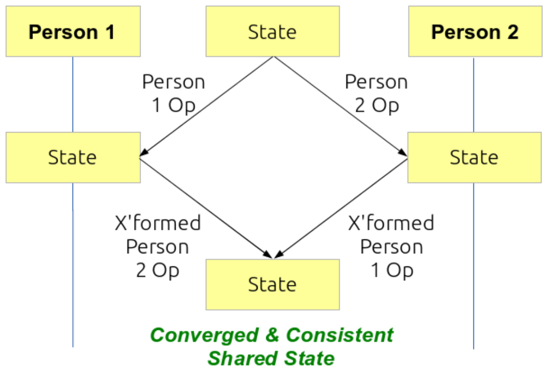


Figure 4: Transformed operations across multiple users (Person 1 and 2).

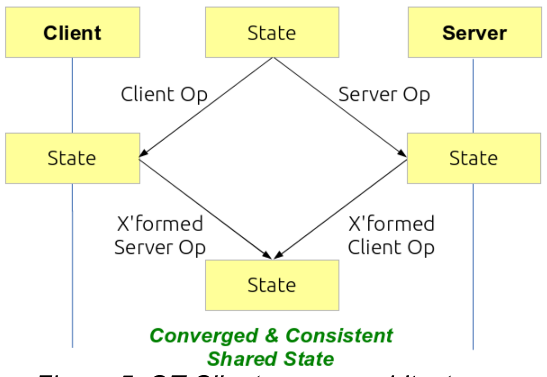


Figure 5: OT Client-server architecture

3. OT LIBRARIES & DOCUMENT DATA FORMATS

The two OT libraries that were assessed in this research are the Google Drive Realtime API (Realtime API) and ShareJS. Besides enabling developers to implement real-time applications without having to implement their own OT algorithms, libraries like the Realtime API and ShareJS can handle documents with data formats other than text. Two common data formats that allow more complex hierarchical shared data models via OT are XML and JSON. While Google's discontinued Wave Protocol featured support for XML-like documents [6], both the Realtime API and ShareJS support the increasingly popular and versatile JSON format. The flexibility of JSON also presents a suitable data model format for PlaySketch. This section presents a technical comparison of the two OT libraries.

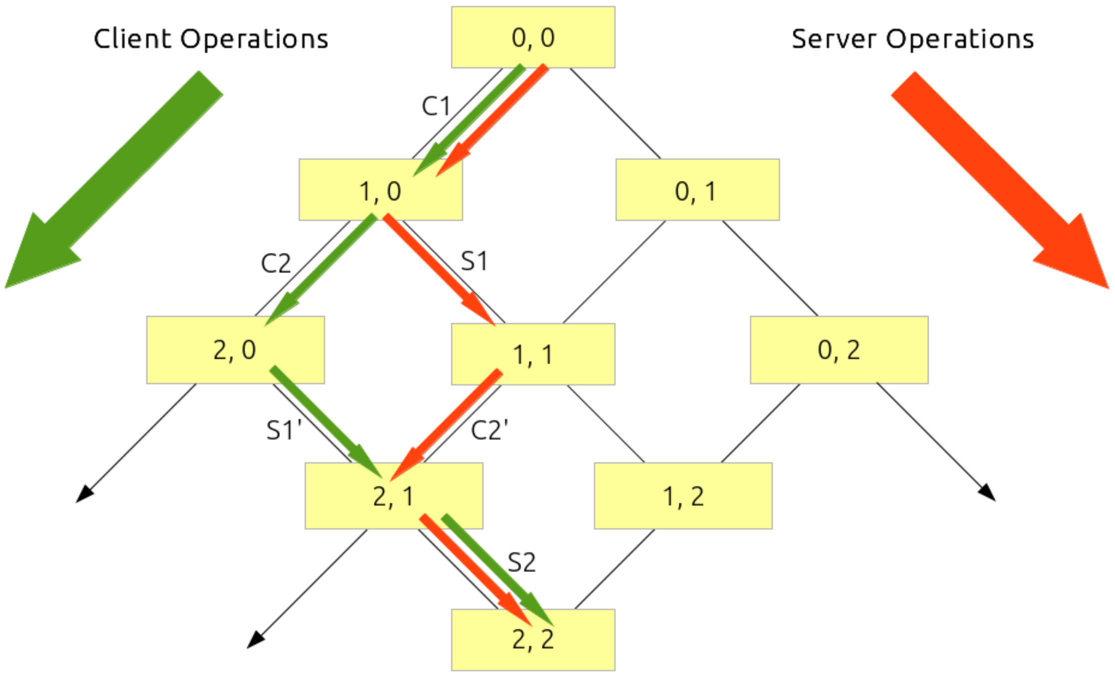


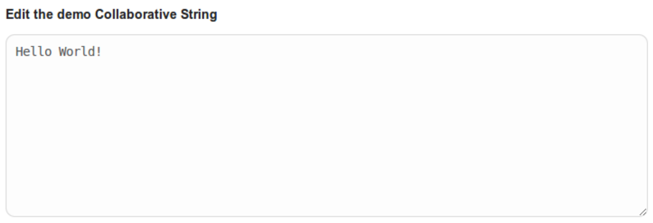
Figure 6: The state space diagram shows all possible traversals of client and server operations. The document state is represented by a tuple of numbered client (left integer) and server (right integer) operations. Each client will have its own unique state traversal, although the states across all clients will eventually be converged and consistent.

This centralised architecture makes OT very suitable for web applications. Contributions to OT libraries and APIs by technology firms like Google and the open source community now allow developers to build real-time collaborative applications without the struggle of implementing custom OT algorithms from scratch. The next section will discuss two such OT libraries that were considered to enable real-time collaborative features in PlaySketch.

3.1. Google Drive Realtime API

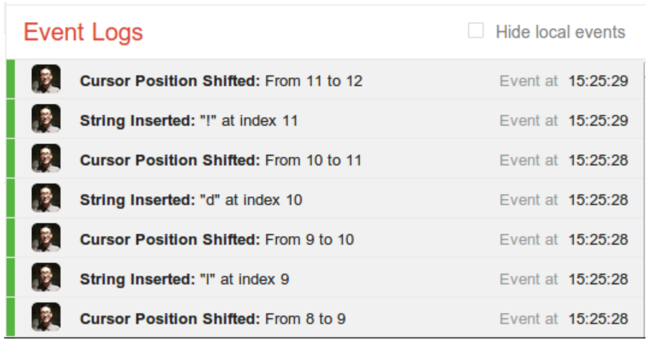
The recently launched Realtime API provides Google Doc-like collaboration as a service for files persisted on Google Drive, the technology firm's cloud based data storage platform [7]. The entire server-side OT component in the Realtime API is hosted on Google's servers. The web-based API exposes collaborative document data formats, events and methods as a client-side JavaSript API. Fundamental JavaScript data types like strings, numbers and lists, as well as any custom JavaScript objects can be shared collaboratively. Since the API relies on Google's server-side OT implementation, developers are only involved in integrating the Realtime API into the front-end code of collaborative web applications.

The software architecture pattern used in Realtime API enabled front-end applications is known as event-driven architecture. This architectural approach is found in most modern JavaScript libraries, including front-end frameworks like jQuery and AngularJS. The Realtime API relies on the front-end system's ability to trigger events asynchronously, and the developer's use of event listeners to appropriately handle different types of events. In event-driven systems like PlaySketch, an *event* is a significant change in *state* [8]. These changes trigger *event* messages to be emitted and consumed by *event listeners* which are able to filter the messages by *event* name. Figure 8 shows several OT specific text *events* that the Realtime API supports. *Event* specific application behaviours, e.g. updating the user interface (UI) after changes have been made to a document, are programmed as callback functions which are passed as parameters to *event listeners*. For text documents (e.g. Figure 7), callback functions may not be necessary as the Realtime API exposes a method to directly bind the shared string to a specific text box element.


Edit the demo Collaborative String

```
Hello World!
```

Figure 7: Text editing example with the Realtime API found in Google's Realtime API Playground.


Event Logs

Hide local events

Cursor Position Shifted: From 11 to 12	Event at 15:25:29
String Inserted: "!" at index 11	Event at 15:25:29
Cursor Position Shifted: From 10 to 11	Event at 15:25:28
String Inserted: "d" at index 10	Event at 15:25:28
Cursor Position Shifted: From 9 to 10	Event at 15:25:28
String Inserted: "l" at index 9	Event at 15:25:28
Cursor Position Shifted: From 8 to 9	Event at 15:25:28

Figure 8: Events triggered when edits are made to the shared text document.

Several limitations in the Realtime API posed considerable challenges. The first of these relates to the complete abstraction of Google's closed source server-side OT functions as a web-based API. While the Wave Protocol (from which the Realtime API evolved) has been open sourced, Google's server-side OT mechanics remains impossible to examine, extend or debug. Besides server-side OT limitations, another significant limitation of the Realtime API is the restriction of document data persistence on Google Drive. Similarly, the usage of the Realtime API is also limited exclusively to users who are logged in via Google's authentication service.

3.2. ShareJS

Unlike Google's Realtime API, ShareJS is a middleware OT library that gives developers full control of both client and server application codes. Built on top of the Node.JS platform, ShareJS is also an open source OT library project conceived and led by a former Google Wave engineer [9]. The client-server architecture of ShareJS, shown in Figure 9, reflects a centralised and symmetric OT approach. Communication between client and server, in particular flows of operations between connected clients, are made through HTTP polling or WebSocket methods.

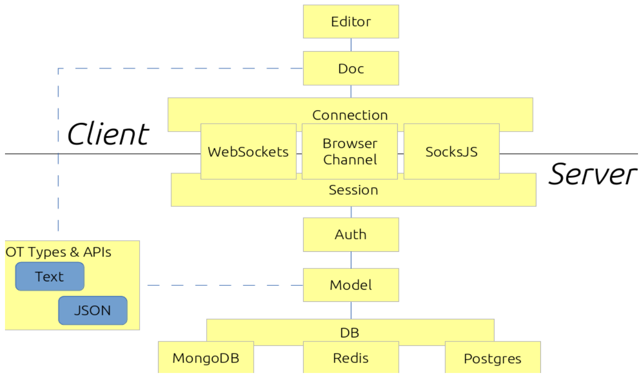


Figure 9: ShareJS client-server OT architecture.

The OT data formats supported in ShareJS are known as *doc types*, with text and JSON available by default. These *doc types* expose type specific properties that allow *operations*, *transforms* as well as serialisation and de-serialisation of shared data. *Operations* in ShareJS are constructed as JavaScript objects with properties describing the changes to be made. A wider range of *operations* is required for the JSON *doc type* to support both lists as well as nested objects that may be added to the shared data model. Due to potentially highly nested object properties in JSON documents, JSON *doc type operations* also feature the useful path property. This path property allows the location of a root level or nested properties to be identified and traversed to when operations are executed, as shown in Figure 10. Custom doc types can be implemented by developers with additional properties and operations should the need arise.

The client-side programming pattern in web-based systems using the ShareJS OT library follows an event-driven approach similar to that used in Realtime API enabled applications. UI frameworks in collaborative applications using complex JSON documents emit *events* triggered by user interactions (e.g. button click or touch) to be consumed by callback functions attached to *event listeners*. However, unlike the Realtime API where *operations* are inferred from changes made in the UI elements bound to the data model (e.g. text boxes), ShareJS client-side applications require *operations* to be explicitly constructed and submitted to the ShareJS OT component that runs on a Node.JS server-side application. Examples of JSON *operations* used in ShareJS are shown in Table 1.

```
// doc is the shared JSON document
doc = {“id”: 1, “words”: [“world”]}

// Path to the “words”
path = [“words”]
// Path to the word “world”
path = [“words”, 0]
```

Figure 10: JSON traversal using path lists.

Operation	**Description**
List Insert	Inserts an element to a list based on the path to the document list property.
List Delete	Deletes an element from a list based on the path to the list property.
Object Insert	Adds a new object property to the JSON document.
Object Delete	Removes an object property from the JSON document.

Table 1: ShareJS JSON doc type operations.

Besides the additional server-side code maintenance and the use of explicitly constructed OT *operations*, another limitation of using ShareJS is its lack of out-of-the-box group undo functionality in the current stable version of ShareJS. However, workarounds discussed in the features and challenges section can be implemented to achieve this feature. The use of Node.JS as the server-side platform is also a potential benefit to the development process as the platform allows back-end developers to use the same programming language (JavaScript) and a similar event-driven pattern as front-end developers. Developers using ShareJS are able to integrate other open source modules (libraries) available from the Node.JS Package Manager. Similarly, as the source code of the OT implementation is openly available for modification, developers working on OT driven projects constantly contribute to improvements and recommendations for future releases of the ShareJS.

3.3. Limitations & Alternatives to OT

The main limitations of OT relate to semantic consistency issues [10], and in particular those relating to the preservation of user intentions [11]. Unlike *operational* consistency preservation which is achieved in OT systems adhering to CP1 and CP2, semantic consistency is concerned with whether *operations* made on shared documents result in meaningful *states*. An example of a semantically inconsistent *state* resulting from concurrent OT operations is shown in Figure 11. Solutions around this issue include integrating locking mechanisms to prevent specific regions in the document from receiving concurrent updates resulting in a semantically incorrect *state* [10].

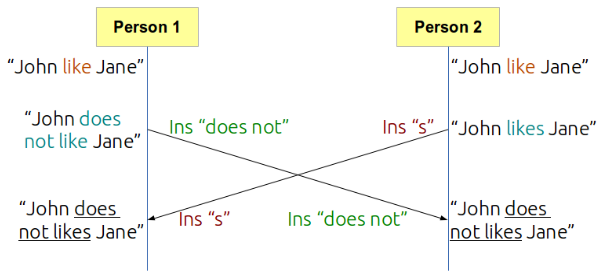


Figure 11: Semantic inconsistency (English grammatical errors) resulting from syntactically correct consistent OT operations.

The literature studied identified Differential Synchronisation (DS) as the only documented alternative to OT [11]. Unlike OT which focuses on the propagation and convergence of multiple concurrent client and server *operations*, DS is based on a constant cycle of *diff and patch* updates based on changes made by users. Another significant difference of DS is the need to maintain additional reference copies of the document on both client and server applications [11]. Although this approach is relatively inexpensive for text based documents, the additional overhead of the reference copy may pose memory issues for applications like PlaySketch which require more complex document data models.

3.4. Choice of OT Library

While both OT libraries are new and somewhat bleeding edge technologies with the potential for both major and disruptive improvements, the lack of OT implementation source code, server-side programmability and the restriction of using Google's storage and authentication services were limitations of the Realtime API that outweighed those found in ShareJS. ShareJS was therefore chosen as the OT library to build a collaborative sketching application.

4. PROTOTYPE METHODOLOGY & IMPLEMENTATION

A suitable web-based rendering technology for sketching had to be identified for use in the prototype sketching application. Based on the preceding discussion of OT technologies, the sketching and rendering technology used should be able to support the serialisation of graphics objects into JSON documents for sharing across multiple sketching interfaces. The application's sketching interface should also feature an event-driven system that will be able to communicate with the front-end OT component in the ShareJS OT library. The integration of an appropriate sketching technology, serialised data model and real-time OT collaboration in this application will serve as a model of reference for future PlaySketch designs and prototypes.

4.1. HTML5 Canvas & SVG

The two web-based technologies available for graphics rendering in browsers, without relying on additional plug-ins, are SVG and the HTML5 Canvas element [12]. While both technologies can produce very similar embedded graphics, key conceptual and technical differences exist. The older SVG technology uses an XML-like structure to append graphics components to the Document Object Model (DOM) of a HTML page. This is in contrast to the HTML5 Canvas element where no additional elements are added to the DOM. Instead, the HTML5 Canvas element allows scripting of 2D and 3D graphics on the Canvas' rendering context through a low-level JavaScript API [13]. Due to the absence of additional graphics DOM elements, web pages featuring graphics generated by the HTML5 Canvas element are more lightweight compared to those using SVG. The rendering performance benefits this brings is more significantly felt as the complexity of the graphics increases [14]. Since the rendering surface of PlaySketch will have to cope with potentially complex sketches of game environments and characters, HTML5 Canvas was chosen for the prototype application.

4.2. Canvas Interactivity, Data Serialisation & Graphics Libraries

The extremely low-level HTML5 Canvas API presents a challenge for developers in terms of graphics interactivity and serialisation of graphics data for persistence or OT driven collaboration. Unlike SVG where graphics data in the form of XML-like elements are available for manipulation and persistence from the DOM via event listeners, the HTML5 Canvas, by default, has no memory of any graphics object previously added. Implementing interactivity using the HTML5 Canvas API is challenging and impractical. Some workaround solutions require up to hundreds of lines of code to enable simple mouse selections and moving of objects. Similarly. serialisation of Canvas data requires scripted graphics instructions to be captured and encapsulated as objects.

To bridge these impracticalities, web graphics developers have implemented JavaScript libraries that add interactive and more classically object-oriented layers on top of the low-level Canvas APIs [15]. The range of libraries and variety of extended Canvas features have grown significantly as the HTML5 Canvas element matures into the web-based graphics technology of choice. Such HTML5 Canvas libraries include Paper.js, EaselJS and KineticJS, and feature graphics capabilities similar to that seen in Adobe Flash with ActionScript. However, the Fabric.js Canvas library stood out as a candidate for the prototype application. Factors influencing this decision include the quality and organisation of documentation and samples, emphasis on an event-driven pattern, as well as an efficient class hierarchy approach that can be adopted in PlaySketch.

4.3. Frabric.js Canvas Library

The event-driven architectural pattern is strongly used in the Fabric.js Canvas library [15]. For example, an *event* is triggered to every time a user clicks or touches (on a touch screen) a shape. This *event* informs the system that an object has been selected. Based on the object data attached to *event* messages, other developer defined behaviours can be programmed. The *event* framework available in Fabric.js therefore provides a suitable approach to integrate OT related functions into *event listener* callbacks. Some of the Canvas *events* available through the Fabric.js API is shown in Table 2. These *events* were extensively used in building the client-side prototype sketching application.

Event Name	Description
path:created	Triggered when Path objects are created on the Canvas.
object:moving	Triggered while any object is moved on the Canvas.
object:scaling	Triggered while any object is scaled on the Canvas.
object:rotating	Triggered while any object is rotated on the Canvas.
object:selected	Triggered when an object is selected on the Canvas.
selection:created	Triggered when a group objects is selected.
object:modified	Triggered after Canvas objects have been manipulated.

Table 2: Fabric.js events used in the prototype.

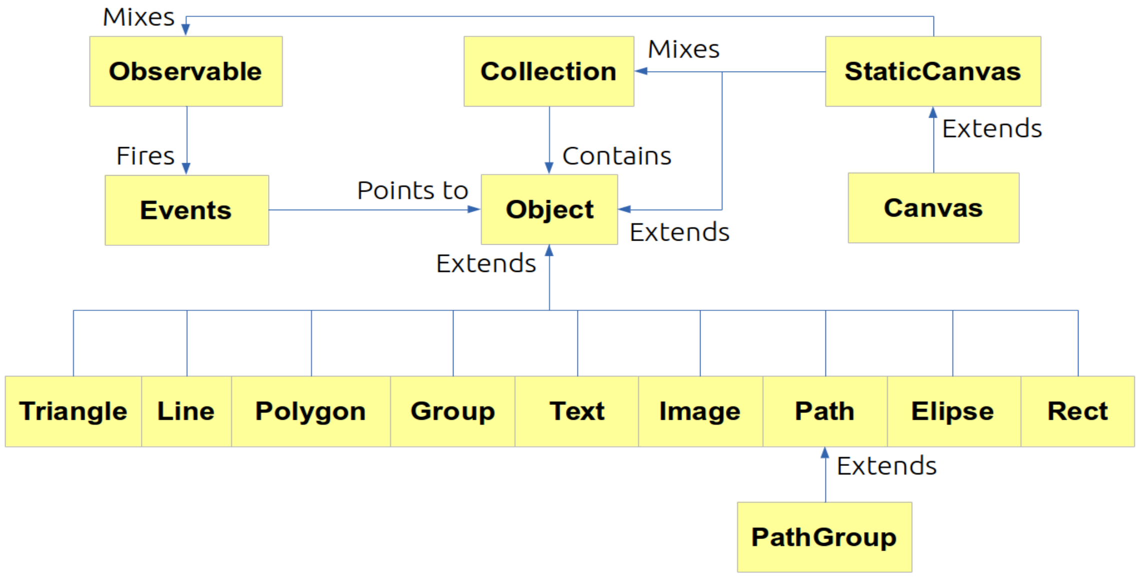


Figure 12: The Fabric.js class hierarchy. Properties in the Canvas, StaticCanvas and other graphics primitives classes are inherited from Object, while Collection and Observable are modules (class property containers) that mix-in object containment and interactive functionality.

Complementing the strong Canvas *event* framework is an object-oriented class based hierarchical organisation of the entire Fabric.js library. The pseudo classical approach in organising the different elements of the library also makes serialisation and de-serialisation a straightforward task. The hierarchical organisation of Fabric.js components can be represented by a directed graph shown in Figure 12. In Fabric.js applications, Canvas objects are the highest level objects that are bound to the HTML5 Canvas element. The graph representation of Fabric.js highlights the relationship Canvas class objects share with other graphics components, as well as the interactive and event layers.

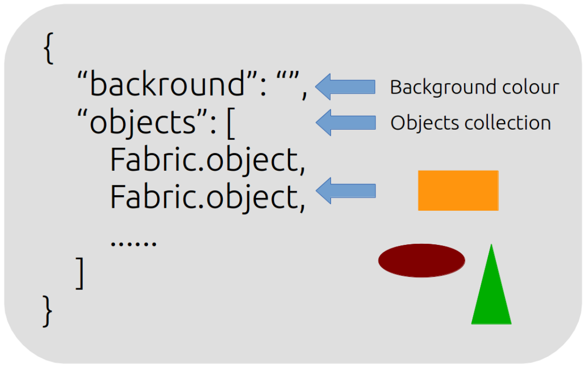

```
{
  "backround": "",
  "objects": [
    Fabric.object,
    Fabric.object,
    ......
  ]
}
```

Figure 13: The JSON serialised representation of the Fabric.js Canvas object also serves as the data model for the collaborative document.

Any objects inheriting from the Object class will feature the "toJSON" and "loadFromJSON" methods. These methods allow serialisation and de-serialisation (creating class instances from plain JavaScript objects). Figure 13 shows a graphical representation of a serialised Canvas object that also serves as the data model for the shared sketching document used in the prototype application. In this sketching data model, root level properties represent Canvas properties like background colour or images. All other graphics primitives contained in the Collection component Canvas objects are stored in the "objects" array.

4.4. Integration of Sketching & OT Features

When integrated with ShareJS in the prototype application, newly created sketches first serialise a blank Canvas object and set the JSON object returned as the shared data model. Other clients connect to this newly created sketch via a unique URL fragment identifier ("#" followed by a randomly generated string). Upon connection, any client opening a previously created sketch has to de-serialise the ShareJS document, allowing the Fabric.js sketch to be rendered on the 2D context.

The prototype's sketching interface allows the user to create free hand "pencil" sketches. These sketches are essentially a collection of Fabric.js Path objects, a graphics primitive object that can be modified by fills, strokes and other methods. The Fabric.js event framework is used to to detect changes on the Canvas, and ShareJS *operations* are constructed and submitted to the application server which propagates these *operations* to all other connected clients.

4.5. Prototype Architecture

Figure 14 shows the client-server architecture used in the prototype sketching application. The entire web application is hosted on an Amazon EC2 server that runs the ShareJS server on top of a Node.JS runtime. The client-side sketching application featuring the HTML based sketching interface is also served from the EC2 instance. All client-server communications are made through HTTP method calls.

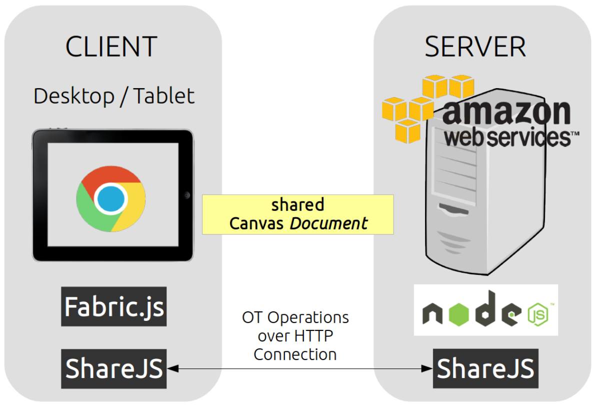


Figure 14: Prototype client-server architecture.

5. PROTOTYPE FEATURES & CHALLENGES

The final prototype application (main interface shown in Figure 16) implements the following features described in Table 3. Detailed implementation challenges and solutions of the various features are also discussed in this section.

Feature Description	**OT Compatible?**
1. Creating free hand sketches	Yes
2. Move canvas objects	Yes
3. Rotate objects	Yes
4. Scale objects	Yes
5. Grouped operations	Yes
6. Erase	Yes
7. Undo & redo	**No**

Table 3: Features List of application prototype.

5.1. Creating Sketches

The implementation of this feature was achieved by utilising the "path:created" Fabric.js *event*. The *event listener* callback function serialises the Path object contained in the *event* message into a plain JavaScript object, and a list insertion *operation* pointing to the "objects" document property is constructed and submitted. No significant challenges were faced in implementing this feature.

5.2. Moving, Rotating & Scaling

All of the subsequent prototype features, except for undo and redo, involve user manipulations of previously created graphics objects through mouse or touch gestures. The interactive Canvas layer Fabric.js provides all the functionality to select a single object, or a group of objects from the rendering surface, as well as to perform manipulations on the selected object or group. A simple manipulation would be to move a selected object by dragging an object from its original position to a new position.

While ShareJS *operations* representing the creation of Path objects are relatively straightforward, object manipulation *operations* require more effort in ensuring client-side efficiency and responsiveness. Since it is not important for other clients to be able to view the actual object movement gestures made by a user, movement *operations* should be constructed after the "object:modified" *event* is triggered, and not while the "object:moving" *event* is being fired.

A simple but inefficient solution to manipulation *operations* is to replace the entire currently selected Path object in the shared document's "objects" collection with a newly serialised copy of the updated Path object. Keeping track of the currently selected object is made possible by comparing the object pointed to by the "object:selected" event with those in the "objects" collection and returning the object's index in the collection if there is a match. However, this solution is not a scalable one due to the large *operation* size when submitting a list updates which are composed as a two-part delete and insert *operation*.

Alternatively, graphics object manipulations can be seen as updates made to specific Path object properties. For example, object movements are essentially changes made to either the "left" (horizontal movements) or "top" (vertical movements) properties, or a combination of both. Likewise, object rotation changes only the "angle" Path object property, while scaling involves changes to combinations of "scaleX", "scaleY", "left" and "top" properties. With this approach, smaller object update operations on the shared document can be made.

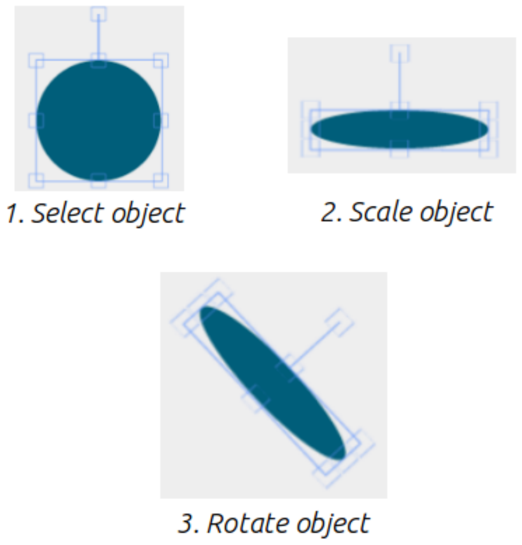
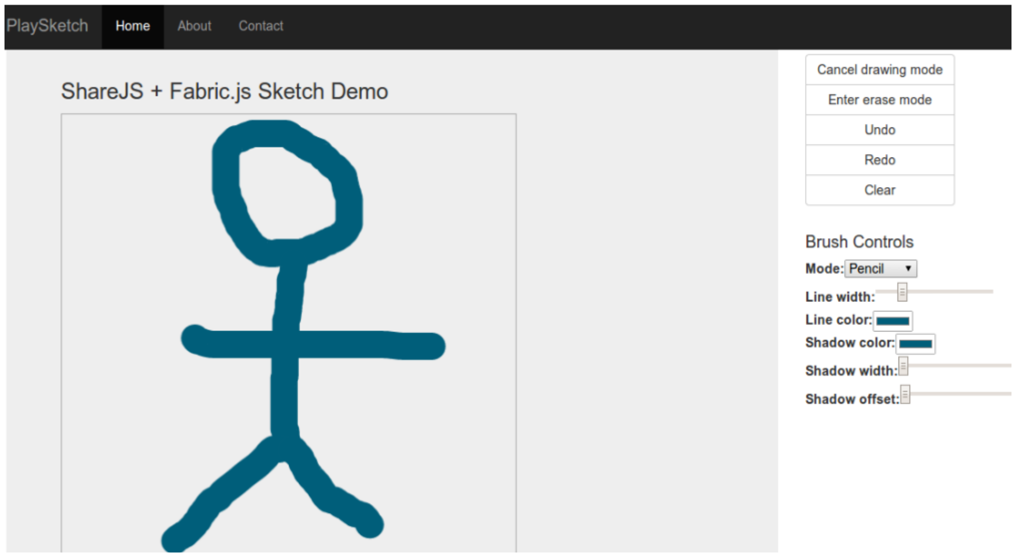


Figure 15: Canvas object selection and manipulation in the prototype application.

In order to implement this approach, a significant workaround was made to allow Path object changes to be detected, as the "object:modified" *event* does not contain information about how objects have been modified. The solution involved borrowing the from the concept of *diff and patch* used in Differential Synchronisation. In the workaround, the *diff* concept is utilised to compare a reference object (a copy of the currently selected object) against the final object after the modifications have been made. The *diff* functionality, provided by the JSONDiffPatch library, is used to generate *delta* objects containing the properties and values. These *delta* objects are then used in the "object:modified" callback to construct the corresponding *operations*. The code snippet in Figure 17 shows how a delta (`objectModDelta`) is created using a diff (`delta`) when objects are moved.

5.2. Grouped Operations & Erase

Grouped object manipulations are achieved in a similar manner to individually selected objects. Instead of tracking a single object reference index, an array of indexes is generated within the "selection:created" *event listener* callback. This *event* points to the group of objects selected by using a mouse or touch to grab an area of the Canvas containing Path objects. Modification *diffs* for each Path object in the group selection are created and used to construct individual OT *operations* which are then submitted to the ShareJS server as a list of *operations*.



Mode: Pencil

Figure 16: Main interface of prototype sketching application. Canvas modes (drawing, erase, undo, redo and clear) and brush controls (line width, colour etc.) are available from control panel. The interface is also compatible with mobile and tablet devices supporting the Chrome browser. Accessible at http://bit.ly/is470_playsketch.

```
canvas.on('object:moving', function(e) {

    // The currently selected object serves
as the reference object
    selectedObject =
canvasObjects[originIndex];

    // The event points to the object that
is being moved
    var movingObject = e.target.toObject()

    // Diff delta object is created using
the diff method that compares the selected
and moving objects
    var delta =
jsondiffpatch.diff(selectedObject,
movingObject);

    // Movements are changes to "top" and
"left" Path properties
    objectModDelta = {
        top: delta.top,
        left: delta.left
    };
});
```

Figure 17: The callback function added to the "object:moving" event listener.

The erase feature is achieved by reconfiguring the "object:selected" *event listener* callback to remove the selected object from the Canvas "objects" collection immediately after the selection has been made. Erase *operations* do not require *diffs* to be used as they are formed by constructing simple list deletions.

5.3. Undo & Redo

The undo and redo features in the prototype application were implemented using undo and redo stacks on the client-side application. When *operations* are submitted to the ShareJS server, they are also pushed to the undo stack. Clicking the undo button from the control panel causes the most recently submitted *operation* to be popped from the stack, and an inverse *operation* (e.g. inserts are inverted to deletes, and vice versa) generated from the popped *operation* is executed locally and submitted to the server. Due to the inverted *operations* not being *transformed* against any server *operation* that may have arrived after the latest local client *operation*, this approach does not adhere to the correctness requirements for collaborative group undo and redo *operations* in OT [16].

The correct approach to OT group undo is shown in Figure 18. In this example, the user wishes to undo the client *operation* C1, and server *operations* S1and S2 are also received concurrently. The complete undo *operation* is constructed by first inverting C1, creating C2. C2 then has to be *transformed* first against S1, and finally against S2. The result of this series of *operation* inversion and *transforms* mean that incoming server *operations* S1 and S2 are executed as if *operation* C1 had never occurred [16]. While conceptually simple, this process presents several implementation challenges, as lists of client and server *operations* have to be maintained by the client-side application.

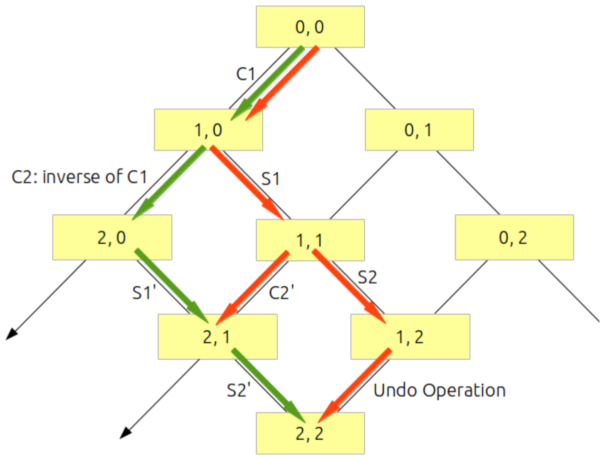


Figure 18: The undo operation as a series of inversion and transforms.

6. CONCLUSION & FUTURE WORK

The prototype sketching application created for this research validates the integration of Operational Transformation techniques with modern web-based graphics libraries that extend the functionality of the HTML5 Canvas element. The application also serves as a reference for future development of the PlaySketch tool. The event-driven architectural pattern was also identified as a suitable programming approach for both client and server-side components of PlaySketch.

This research also highlights several implementation challenges of forming OT *operations* efficiently in web-based graphics tools like PlaySketch. To reduce network overheads of large object replacement operations, a workaround was devised based on the concept of object *diffs*. While this approach significantly decreases the amount of data sent over the network, it increases the memory footprint on the client-side application. Future work on PlaySketch will investigate other workarounds to this issue, including extending the *event* framework in Fabric.js to allow tracking of object property modifications.

Future work on the PlaySketch tool will also focus on solving the OT correctness issues in the undo and redo feature. The solution derived for this feature could also be contributed back to the ShareJS project as open source patches to the existing version of ShareJS, or as recommendations for future versions.

Finally, as PlaySketch evolves into a more complex tool with additional features like animation and game mechanics, the underlying data model serving as the shared collaborative document will be reviewed to ensure that OT *operations* remain efficient and sufficiently light-weight. Particular attention will be paid to how instructions for animations and game logic are serialised into the JSON data model. The object-oriented and pseudo classical approach used in Fabric.js provides PlaySketch a useful reference in implementing rich JavaScript features with hierarchical code organisation. The prototype application is available at http://bit.ly/is470_playsketch.

ACKNOWLEDGMENTS

This research project acknowledges the support of Dr. Richard C. Davis for his continuing support, and guidance over the K-Sketch and PlaySketch projects.

REFERENCES

[1] Davis R.C., Colwell B, and Landay JA. K-Sketch: a “kinetic” sketch pad for novice animators. In. the SIGCHI Conference on Human Factors in Computing Systems 2008, ACM (2008), 413-422.

[2] Davis R.C. Prototyping Video Games with Animation. In. the Games and Innovation Research Seminar, Game Research Lab (2011), 49-52.

[3] Ellis, C.A. and Gibbs, S.J. Concurrency control in groupware systems. In. the SIGMOD international conference on Management of data, ACM (1989), 399-407.

[4] Sun C., Jia X., Zhang Y., Yang Y., Chen D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM 5, 1(Mar. 1998)

[5] Nichols D.A. , Curtis P., Dixon M. and Lumping J. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In. the 8th annual ACM symposium on User interface and software technology, ACM (1995), 111-120

[6] Wang D., Mah A. and Lassen S. Google Wave Operational Transformation. (July 2010). Retrieved August 21, 2013 from http://www.waveprotocol.org/whitepapers/operational-transform.

[7] Google Inc. (October 2013). Google Drive Realtime API Overview. Retrieved October 20, 2013 from https://developers.google.com/drive/realtime/overview.

[8] Chandy K.M. Event-Driven Applications: Costs, Benefits and Design Approaches. In. Gartner Application Integration and Web Services Summit, Gartner (2006).

[9] Gentle J. ShareJS – Live concurrent editing in your app. (November 2011). Retrieved 21 August, 2013 from http://sharejs.org.

[10] Sun C. and Ellis C.A. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In. ACM Conference on Computer Supported Cooperative Work, ACM (1998),59 – 68.

[11] Randolph A., Boucheneb H., Imine A. and Quintero A. On Consistency of Operational Transformation Approach. EPTCS 107 (2013), 45-59.

[12] Johnson D.W. and Jankun-Kelly T. J. A Scalability Study of Web-Native Information Visualization. In. GI '08 graphics interface, GI (2008), 163-168.

[13] Opera Software. SVG or Canvas? Choosing between the two. (February 2010). Retrieved 25 August, 2013 from http://dev.opera.com/articles/view/svg-or-canvas-choosing-between-the-two

[14] Kamel Boulos M.N., Warren J., Gong J. and Yue P. Web GIS in practice VIII: HTML5 and the canvas element for interactive online mapping. International Journal of Health Geographics 9, 14 (2010)

[15] Zaytsev Y. and Kienzle S. Introduction to Fabric.js, Parts 1-4. Retrieved 5 September, 2013 from http://fabricjs.com/articles/

[16] Sun C. Operational Transformation Frequently Asked Questions and Answers. Retrieved 15 October, 2013 from http://www3.ntu.edu.sg/home/czsun/projects/otfaq/