CRDT Canvas

Jack Morrison
Hugo Mayo
Serene Chongtrakul
Hashan Punchihewa
Kapilan Manu Neethi Cholan

January 2020

Contents

Executive Summary

Remote collaboration has been increasing in popularity over the last couple of years, within professional working environments, academia as well as other fields. The benefits are obvious: companies that allow their employees to work from home attract excellent candidates that may have previously been put off by having young families; large companies with regional offices all around the world can easily communicate and collaborate with teams from various locations; people living in less urban or less developed parts of the world have a larger number of economic opportunities. Today we use tools such as Slack and Skype for group communication or Google Docs and Office 365 for shared document editing. However, tools for more visual collaboration, such as through drawing, are behind. The whiteboard, a mainstay in every meeting room and classroom across the world, remains rather analogue, with many digital offerings suffering from lag, dropped connections or devices not syncing together correctly. This could be, in part, due to the reliance of these applications on steady internet connections and central servers, which cannot always be found if the applications are used in more rural areas or on mobile networks. Our group has created a resilient collaborative drawing app for iOS using Conflict-free Replicated Data Types (CRDTs) that aims to prevent failures as listed previously from becoming an issue.

Our app, which is optimised for iPads, is called Canvas. Canvas allows up to 1000 users to connect and draw at the same time. Changes are propagated in real time between all devices. But what's different? Well, if a user becomes temporarily disconnected, then on reconnection, the changes that they made during this period will sync up, thus supporting offline editing. Canvas has all the features of your typical drawing app, with some added extras. For example, Canvas comes with some shape recognition features, so when a user draws something that looks close to a straight line or rectangle, the drawing will be corrected, which is useful for things such as UML diagrams. It also has Bluetooth network optimisation, which means transmission speeds are much faster when people are working in close proximity to each other.

The novel aspect of Canvas is that it use CRDTs, as the underlying data structure. These are a class of data structures that allow for strong eventual consistency, without a central server i.e. it does not rely on one of the nodes to always be connected. So, if a single device disconnects, the rest of the devices can continue to use Canvas as normal; there is no reliance on one device. Furthermore, this reduces latency found in applications that rely on the cloud for synchronisation. This is what sets our app apart from others currently on the market, such as Microsoft Whiteboard, which uses cloud connections for synchronisation, or Inko, which, although it is also a collaborative iPad drawing app, only supports connections through Bluetooth.

1 Introduction

In North America, 85% of office workers expect their employers to provide technology that allows them to work from wherever they choose and 83% of employees use technology to collaborate in real time with people in different locations, however, 78% of them say they experience frequent technical difficulties that impact the collaborative experience [1]. It's clear that there is a huge demand for collaborative applications with tools like Google Docs or Office 365 allowing geo-distributed teams to work together on documents or spreadsheets. Unfortunately, visual collaboration tools like Microsoft Whiteboard are far from satisfactory, suffering from dropped connections, lag, and devices going out of sync, especially in areas with unreliable internet connection. Therefore, there is a need for a collaborative drawing application that solves these issues.

Real-time collaborative applications encounter a problem known as the CAP theorem [2]. This states that it is impossible for a distributed system to simultaneously provide more than two out of three of: consistency, all nodes see the same version of the data; availability, the system always responds within fixed upper limits of time; and partition tolerance, the system always gives correct responses even when messages are lost or network failure occurs. This has led to the introduction of the notion of eventual consistency, which means that eventually all nodes will see the same version of the data. This model of consistency is suitable for Canvas since nodes can disconnect from each other, but when they reconnect, they need to eventually become consistent so that what every user sees lines up.

Traditionally collaborative applications have been implemented using operational transformations (OTs) [3]. A notable example is Google Docs. However, operational transformation algorithms suffer from a number of drawbacks.They are complicated to design and implement. Formal proofs of correctness for OT algorithms are often difficult to formulate [4]. They also require a central server to synchronise changes. In contrast, conflict-free replicated data types (CRDTs), first defined in [5], are much simpler to implement, still guarantee eventual consistency and do not require a central server. While they haven't become as widely used as OTs [6], they have seen industrial use e.g. in Riak [7].

A key question that we explored in this project was the viability of using CRDTs in a software-engineering context, with the goal of delivering a product to end users. We looked into how this affected factors such as the repertoire of features we could implement, and the performance of the application. This is interesting as industrial use and research of CRDTs has been focused on a narrow range of applications, particularly collaborative text editors. We also considered the best way to implement CRDTs. We evaluated whether two different CRDT libraries are feasible options for non-specialists. These libraries were designed to allow arbitrary document structures to be synchronised.

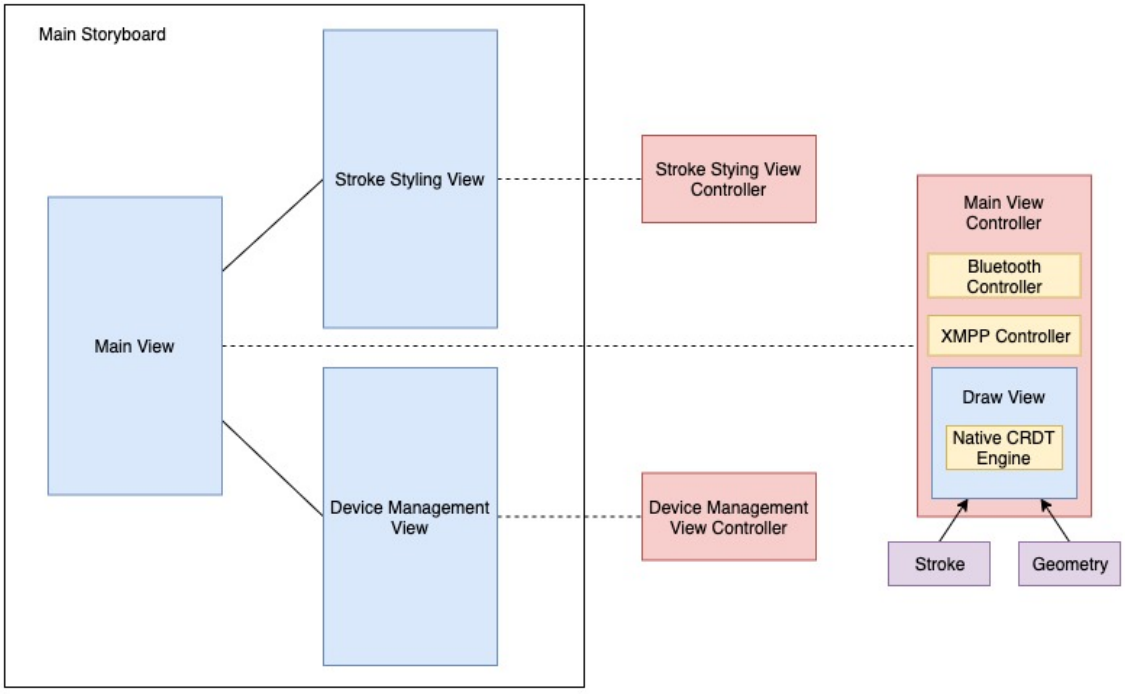
The overarching objective of the project is to develop a collaborative drawing app for iOS devices using CRDTs to answer these questions. Multiple users should be able to work together on the same drawing, supporting the automatic and real-time synchronisation of changes when users are online. Similarly, users should be able to edit offline, but their changes should be automatically synchronised upon reconnection. The app should be decentralised, setting it apart from other solutions such as Microsoft Whiteboard.

Our end product is Canvas, a peer-to-peer collaborative drawing app on iOS. The app comes with the core features of a drawing app along with the tools to connect users together. Users can connect to each other using either XMPP or Bluetooth, or a combination of both, depending on their physical location. Edits sent across the network between Canvas users comply with a protocol that other drawing apps can implement in order for them to be able to communicate with Canvas. We have tested this with another CRDT powered drawing webapp which implements this protocol. We followed a human-centred design methodology, which led to the development of features such as curve smoothing, and shape recognition for lines and rectangles, with a particular use case being UML diagrams.

2 Design and Implementation

2.1 System Design and Implementation

One important consideration in the design of the app was modularity. The core business logic behind our app is written to be agnostic to the underlying CRDT implementation (the CRDT engine), or the network stack. Our app was written for iOS. We considered the tablet the ideal form factor for our device for a number of reasons. Firstly, their screen size is big enough to be useful for collaborative work. Secondly, tablets support touch input and typically stylus input. Thirdly, they are widely available. There were 136.8m tablets shipped in 2019 [8]. Furthermore, 61% of tablet vendor market share in November 2019 was Apple [9]. Writing a native app meant we could provide a better user experience than a web app. It also meant we can take advantage of more features, such as Apple's `MultipeerConnectivity` framework.



Above is a system diagram, which gives an overview of the structure of our application. The app consists of a Main Storyboard, which contains the Main View. This is the main screen where the navigation bar and drawing area are. There are then two more views: one for stroke styling, which contains a colour palette and thickness selector, and one for connection management, such as which room the user is in and who else is in it. Each of these views has a view controller, which programatically controls what happens in the view. The view controllers for the stroke styling and device management are quite simple, just implementing the required code to complete the functions of that view. They don't use any other files, but do have references back to the main view controller. The main view controller contains an instance of a Bluetooth Controller and an XMPP Controller. These control Bluetooth and XMPP connections respectively, meaning they are modular and can be swapped out for different implementations, e.g. using a different XMPP server. The main view controller also creates the Draw View, which is essentially the canvas which the user draws on. Inside of this is where the CRDT is, since this has to directly interface with the strokes produced on the canvas. The draw view also relies on two external files. One of these is Stroke, which contains the data structure for a stroke, and functions to be called when there is a new stroke (or deletion) which occurs. The other file it relies on is Geometry, which performs geometric calculations and includes the logic for shape recognition.

2.2 UI Design and Implementation

We went through many iterations of the user interface of our project, taking into account responses from our client and test users, in order to create a functional human-centered design.

One of the key aspects of our design was the navigation bar (navbar). This contains every aspect of control for our application, including brush colour, deletion, shape recognition, sharing and device connection management. We went through a few iterations of the tool icons, experimenting with different numbers and styles, eventually settling on the most appropriate default Apple icons, since these are universally recognised by iOS users. The placement of our icons along the navbar was also guided by user feedback, with the undo/redo button to the left, drawing instruments in the middle, and connection management and sharing on the right, offering an intuitive design to our users.

One important piece of feedback we got regarding the navbar was that it was hard to see which colour was selected, or if another instrument, such as the eraser, was being used. We fixed this by changing the colour of the active button to red, and the colour of the paintbrush (which is the colour selection button) to the colour which is currently active. We also made the navbar follow iOS guidelines by changing between black and light grey for dark and light mode respectively.

One thing users said they would like to do was to share their drawings outside of the app, so we used the native apple share sheet in order to allow users to do this. Since it uses the native share sheet, it has all of their saved preferences for apps to share to, and since it is implemented as a popover, it does not take the user away to a separate page.

The next important UI feature was the colour palette for choosing stroke colour. We initially tried having preset colours, but users said they would prefer a wider choice. We

therefore use a framework called `FlexColourPicker`. This gave the base for a colour picker, however, we overlaid this as another view in the project to mean it could appear on top of the current view, so the user does not need to leave the page to change colour. We also improved the existing framework by adding the ability to change stroke thickness into the same view.

The next user feedback regarded shape recognition. Initially, we had it active all of the time, as was requested by our client, however, we soon realised this meant that it would pick up lines which should not be picked up, therefore, we created a toggle which lets this feature be turned on or off.

Another big challenge was device support. We wanted to prioritise support for all sizes of iPads, but we also wanted our application to support iPhones if possible. We initially tried adding scrolling and panning, however this meant that the application had to re-render the paths every time the user zoomed, and it changed things like axis scaling, which was very CPU intensive. We therefore decided to only include scrolling, so if a user has a smaller-sized screen, they can scroll across to see the rest of the canvas, where the default size is the size of the largest iPad (12.9").

The final main UI feature was the capability for device management (seeing who else is in the local room) and room management. We had this as another UI overlay, allowing users to type in the name of the room they want to join, and displaying which other Bluetooth users are in the room. We did this so that users can know which devices their device is connected to in a peer-to-peer format, since this was a privacy issue raised by some of our testers. It also was needed for us to implement session management, and before this we had separate overlays for session management and seeing who else is in the room, but users told us they would prefer it together, since it is more concise.

2.3 CRDT Design and Implementation

The novel part of the project was to use conflict-free replicated data types to sync changes between different nodes without the involvement of a central server. A mathematical introduction to CRDTs is provided in Appendix A. CRDTs for primitive data types such as integers and sets have been explored in academic literature, however, the issue of complex data structures formed by nesting various data types is one that has only arisen recently.

2.3.1 Automerge Implementation

We initially used a JavaScript CRDT library called Automerge, however, there were a number of disadvantages with using a JavaScript library. In order to incorporate it into our app, we used JavaScriptCore. First, since JavaScript is interpreted, it runs significantly slower than Swift. Secondly, there is the cost of interoperability, which involves serialising the Swift structures to a JSON string which is deserialised in JavaScript. This imposes an additional performance cost. These performance costs were observed in profiling and benchmarking. Therefore, we switched to a native Swift CRDT library.

In order to evaluate the performance of Automerge, we identified three metrics that could be used to compare Automerge with other CRDT implementations; the size of the the CRDT documents; the time taken for the JavaScript to run when another point is added to a stroke; and the time taken for the Swift to run when another point is added

to a stroke. Using the data collected for the last two metrics, we can also determine how great the cost is of the additional overhead caused by switching between Swift and JavaScript.

In order to measure the size of the documents, a benchmark runs, creating a single stroke to which points are individually added until the stroke contains 1000 points. After a point has been added to the CRDT, the length of a JSON serialisation of the document is recorded. The results of this are seen in Figure 1, in which we see as expected, a linear relationship between the length of the document, and the number of points that have been added to the stroke. On average, 17.8 extra chars are added per point. Without any form of compression or caching in the CRDT, the document size will quickly grow with multiple users drawing, causing operations on the CRDT (such as adding a new stroke or a point) to take longer, increasing the amount of observable lag that users in the session experience.

Because we know that the Automerge document size seems to influence the amount of lag a user experiences, we can hypothesise that the majority of processing time happens during the JavaScript callback in Swift, as this involves the serialisation and deserialisation of JSON to interoperate between JavaScript and Swift. To check this claim, we will use the same benchmark that adds 1000 points to a stroke but this time measuring both the time taken for the JS to run (effectively measuring the speed of the Automerge library operations) and the time taken for Swift to run. Figure 2 plots the time spent in JS runtime alongside the time spent in Swift runtime. On average 47 ms is spent in Swift runtime per each added point which is a lot greater than the average of 3.7 ms for each added point within JS runtime, lining up with the prediction that the JavaScript callback in Swift takes a long time, and that the interoperation is the performance bottleneck.

In Figure 3, we see a cumulative version of Figure 2 which shows that the JS processing time scales linearly, similarly to the size of the Automerge document which reinforces the point that Automerge itself is quite efficient and is not the bottleneck that causes lag. Instead, it's the quadratic growth of the processing time within Swift that is the bottleneck, supporting the hypothesis that the JS callback in Swift is what's leading to a great amount of lag.

2.3.2 YJS Implementation

As an experiment, we wanted to try another CRDT library to see if there are better alternatives to Automerge. Unfortunately, although we wanted to test a CRDT library in Swift or C++ to remove the bottleneck of the JavaScript callback, there aren't any suitable CRDT libraries that allow the composition of data structures, which is necessary to support the needs of Canvas and so we settled on testing another JavaScript CRDT library, YJS. We chose YJS as an alternative to Automerge as the API seemed relatively straightforward and its capability as a well-written, functional CRDT library had been proved through a number of demo projects on their website that use it in their backend.

To test YJS's performance, we wrote an adapter, allowing for YJS to be slotted in for Automerge. We then ran the same benchmarks on YJS, obtaining the following results for the size of the YJS documents and the time taken to add points to a stroke in JS and Swift. In Figure 4 we plot the length of the CRDT documents for Automerge and YJS against each other and observe that both have a linear increase in size for their respective documents, however, the rate for YJS is greater than Automerge's. So, it is

reasonable to assume that it will be slower than using Automerge, which is confirmed by Figure 5, which plots the cumulative JS and Swift times for both YJS and Automerge against each other. It's worth noting that the benchmarks for YJS took too long to run, with average runtimes for JS and Swift at 7.2 ms and 195 ms respectively, and so only the results for adding the first 100 points to a stroke can be shown. Perhaps with further optimisations to our usage of YJS, we would be able to make it more efficient and bring it in line with Automerge, however, YJS is still a good alternative to use as far as a general CRDT library goes.

2.3.3 Native Swift CRDT Implementation

The theoretical model for our CRDT implementation is defined as follows. We implement a CvRDT. A document is a G-Set of strokes. A stroke of length p consists of three main things: a unique identifier, a list of points and a subset of the interval $[0, p]$, representing the part of the strokes that has been shown. For instance, if the stroke has 5 points, and the subset shown is $[0, 5]$ then the entire stroke will be shown. If the subset shown is $[0, 1] \cup [2, 5]$, the part of the stroke between points 1 and 2 will be hidden. If the subset shown is $[0, 0.5] \cup [1, 5]$, part of the stroke, starting from halfway between points 0 and 1, and ending at point 1 will be hidden.

There are three operations: stroke addition, point addition and stroke partial removal. Stroke addition consists of a unique identifier and a start point. Point addition consists of a unique identifier and a list of points to add. Stroke partial removal consists of an identifier, and an interval $[a, b]$ to remove from the subset of points to be shown. To show its correctness we must show that the application of two operations are commutative. Note that since we are using a CvRDT (as opposed to a CmRDT), we only have to show that two operations from different nodes are commutative, since messages from the same node in CvRDTs arrive once an in-order.

Let us consider the point addition case. This appends the points in the change message, to the list of points for the stroke with the given identifier in the document. It is easy to see how point addition commutes with stroke partial removal. It is easy to see how point addition commutes with point addition and stroke addition, when they refer to different strokes. The tricky case is when they refer to the same stroke. In this case, you can always assume they originate from the same node. This is handled theoretically by only considering point addition operation valid, if the originating node is the same as the node that creates the stroke. Therefore, we do not need to consider these cases.

Let us consider the stroke addition case. This adds a new stroke with the given unique identifier and the list of points initialised to a singleton list with just the start point. It should be easy to see that stroke addition commutes with stroke addition, since they are given unique identifiers. We have seen that stroke addition also commutes with point addition. Now consider stroke addition with partial removal. It is possible that the partial removal arrives before the stroke addition. This is not difficult to handle, simply create a placeholder stroke with empty points, and assume the subset of points shown to be the entire real line, and apply the operation.

Let us first consider the partial removal case. If the stroke at the downstream node currently display subset A of points, and the change removes interval B, the new subset of points to be shown is $A \setminus B$. Note this operation is essential set difference. It should be easy to see partial removal commutes with partial removal. $A \setminus B \setminus C$ is the same as

$A \setminus C \setminus B$. We have already seen that partial removal commutes with stroke addition and point addition.

One difference between our CRDT implementation and Automerge's CRDT implementation is the way lists are handled. In the case of the list of points, Automerge assigns each element in the list, a unique identifier. This is necessary due to the semantics of JSON, which allows the insertion and deletion of elements at arbitrary points in the list. In the case of point addition, we knew that we would only have append operations. Moreover, we knew all these append operations would originate from the same node. Therefore, we did not to use identifiers. Not using identifiers led to a massive reduction in message size.

One interesting property of our CRDT implementation is that it does not take into account the timestamps when various actions took place, unlike other common CRDTs. For example Automerge and YJS use a last-write-wins approach to avoid conflicts. However, some features that we could have added might have necessitated the introduction of this approach. For instance, allowing users to change the properties of a stroke such as position, colour, thickness, would have required us to embed a Last-Write-Win Register in the stroke data structure, for each of these properties.

Another limitation of our CRDT implementation is that it is not possible to undo the remove of parts of a stroke (partial deletion). This is not a problem that could be solved with Last-Write-Win Registers since partial deletion is applied to a real-valued interval between 0 and the number of Bézier curves, not one of a finite collection of parts. One approach to this problem is to store the history of the partial deletion operations. This ties in with the idea of making the CRDT document simply the history of changes to the document which is explored in [10].

This may seem inefficient in terms of storage, but this is remedied by removing operations that are rendered irrelevant due to future operations. This idea could be extended to cover this case, by modifying operations that cancel each other. For instance, if interval $[0, 20]$ is deleted but $[10, 30]$ is made visible at a later timestamp, then the previous operation could be changed to $[0, 10]$. Once a partial delete operation is only effective over a zero-length interval, it can be removed. Furthermore, adjacent removed partial removal operations could be coalesced, e.g. the removal of $[0, 20]$ and the removal of $[20, 40]$ could be unified to $[0, 40]$.

After removing Automerge and using our Swift CRDT to handle syncing users we noticed a much smoother drawing experience with the lag being a lot less noticeable even after drawing a lot on the Canvas. Benchmarking this using the same tests we used for Automerge and YJS gave us the amount of time it took for adding 1000 points to a line, with the results displayed in Figure 6. From this we can see that the run time for adding points is linear, compared with the quadratic growth that both Automerge and YJS experienced. When looking at the total time taken to add 1000 points, the run times are 65.7 ms and 48631 ms for the Swift CRDT and Automerge respectively, and for YJS it took 17756 ms to add just 100 points. The Swift CRDT avoids the bottleneck that Automerge experiences, resulting in the benchmarks performing at an order of magnitude better using the Swift CRDT which can be clearly seen in Figure 7 which plots the Swift CRDT and Automerge data on the same graph.

2.4 Stroke Design and Implementation

Strokes are implemented as a series of Bézier curves. We did this because generating lines from the points would produce a jagged curve, since the number of points would be too few for the curve to look smooth, and having a smooth curve is an important visual feature for Canvas. One challenge was to generate a Bézier curve from the list of points given. This involved some experimentation to find an algorithm that produces visually appealing results, as we noted variation in the quality of different algorithms.

Another difficulty was implementing partial deletion. The eraser was a circle, so the challenge was to find the intersection between a Bézier curve and a circle. Algebraically, this produces a sextic equation, which is not solvable in radicals, as per the Abel–Ruffini theorem. A number of approaches were considered to find the intersection. The first approach we tried was Newton's method. The computer algebraic system (SymPy) was used to derive the relevant equations. However, this approach led to problems with performance and was overly complex as sextic equations can have up to six distinct roots. The approach we went with in the end was a variant of De Casteljau. This recursively split the curve into smaller curves, which are then treated as straight lines. This reduces the problem of finding the intersection of a circle and a Bézier curve to the much simpler problem of finding the intersection of a circle and a line.

2.5 Shape Recognition Implementation

One of the main features developed with the user in mind was the shape recognition feature. After talking to many possible end users about the potential uses of Canvas, one of the things that came up a lot was the ability to draw shapes so that they could be used for things such as UML diagrams. We decided that straight lines and rectangles would be most useful for those cases and most straightforward to implement. As mentioned earlier, the hand drawn shapes will only be corrected when the shape recognition mode is on.

For straight lines, after a user has finished drawing something by hand, we detect the start and end point of the stroke drawn. Using this start and end point, we then formulate a straight line equation using the following formula:

$$\frac{(y - y_1)}{(y_2 - y_1)} = \frac{(x - x_1)}{(x_2 - x_1)}$$

We then loop through all of the points that the user drew and see if the points either lie on the line formulated or lie within 20 pixels plus or minus of the line. If yes, then we decide that what the user had drawn bares a close enough resemblance to a straight line and delete the stroke the user just drew. We then replace it with a new stroke that is a straight line with start and end points the same as what the user hand drew. It should be noted that the 20 pixels was chosen fairly arbitrarily, with us fine-tuning this number through trial and error to ensure we correctly recognise a hand drawn straight line in almost all cases.

Recognising a rectangle proved harder. Our first design had a fairly poor success rate, with the main concept being attempting to detect corners from what the user drew. We decided to define a corner as when a user has drawn a stroke that has a sharp turning of over approximately 22 degrees. Similarly to the way 20 pixels was chosen for the

straight line, this 22 degrees was chosen because we found that if the value was any lower, then corners were often detected when the user was drawing a curve, and if the value was any higher, then we often missed corners, leading to pentagons being detected as rectangles. We loop through approximately every 10th point that the user draws and compare it to the 10th point away. We calculate the angle between them using atan2(x, y) which gives in radians, the angle between the positive x axis and the line drawn from (0,0) to the point (x,y) != (0,0). If the angle between these 2 points is larger than 22 degrees, then we add this corner to a list of detected corners. We then check if we get the expected number of corners (3 in this case because a user is only going to draw 3 to form a quadrilateral, the 4th corner would just be the meeting of the start point and the end point) and if yes, we continue on to check if each 'side' is a straight line. If not every side is a straight line or we do not get exactly 3 corners, then we also do not classify it as a rectangle. Then similarly to straight line correction, we delete what the user has drawn and replace it with a rectangle, formed with corners as detected previously. This design had a success rate of 60%.

Our second design for rectangle recognition proved to be much more successful. This new design works by looking for the minimum x and y and maximum x and y values in the hand drawn shape that the user drew. Using these values, we generate the 'perfect' rectangle and note down the corners for this rectangle. We then loop through the points in the shape the user drew and look for the points that are closest to each of the 4 corners and call these the corners for what the user drew. Using the same function as earlier when checking if we have a straight line, we check that each of the 4 lines between each of the corners that we have found are indeed straight lines. If yes, then we delete what the user has drawn and replace it with the 'perfect' rectangle that we formulated earlier. This design had a success rate of 88.5%

2.6 Network Design and Implementation

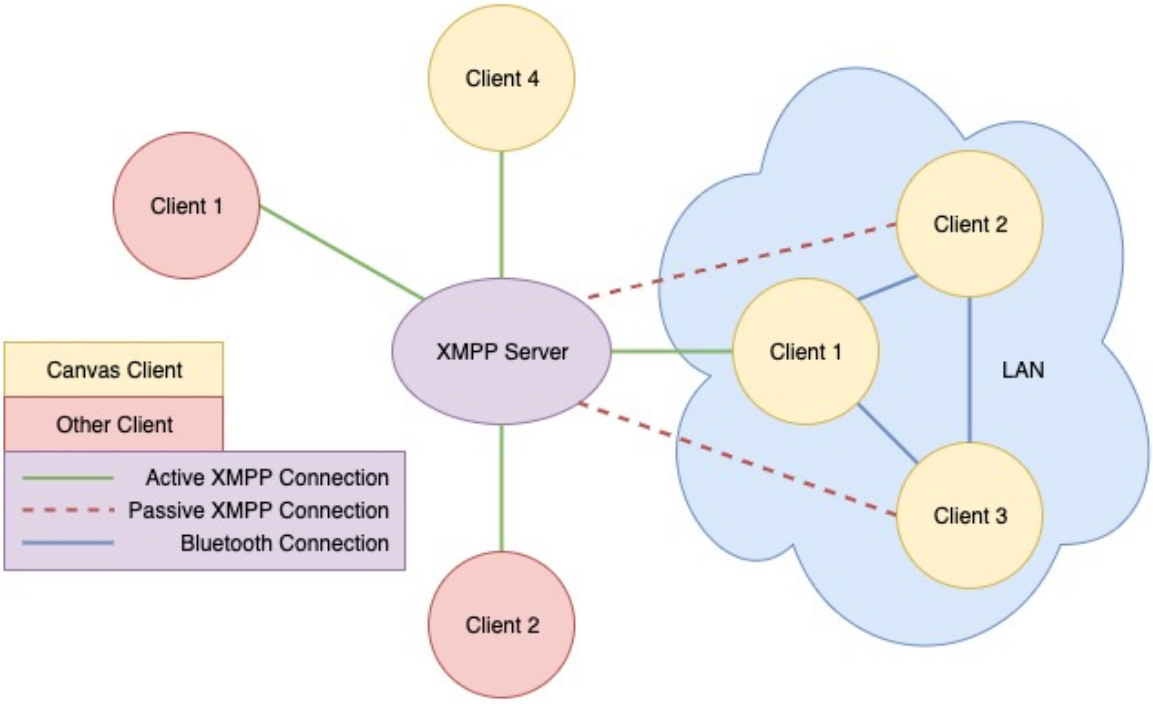
We decided to use two main technologies for networking. These are XMPP and Bluetooth.

XMPP is an open standard for messaging, which means it is a protocol which would easily allow expansion of our application. It is not hard to set up an XMPP server, and it would allow users to potentially use their own server which fits their own needs (e.g. number of users). This is our primary method of networking. We currently have one centralised XMPP server, however, this could easily be distributed to more, based on the number of users we have.

We also use Apple's framework for creating a LAN, called `MultipeerConnectivity`. This uses a combination of Bluetooth and WiFi in order to create a LAN of up to 8 iOS devices. It is all done behind the scenes in order to optimise the user's experience, and is used in combination with XMPP.

Below is a diagram showing an example situation which may occur, and how the application would respond.

1. The first Canvas user joins a room (Client 1). They are the only person in the room. They have an active XMPP connection to the server.



2. Client 1 looks to see if there are any devices around it which it could create a LAN with. This would happen if they are in the same room. There is not.
3. Canvas Client 2 joins. They create an XMPP connection, and then look for any devices around it to create a LAN with.
4. It sees Client 1, so creates a LAN with this. There is now a LAN with Client 1 and Client 2 in.
5. Client 1 is the "Master User", so they are the only one who send and receive data from the XMPP server. Client 2 sends its data over the LAN connection to Client 1, who propagates it to the rest of the network. However, Client 2 retains a passive XMPP connection in case the connection to Client 1 drops.
6. Canvas Client 3 joins the network the same way as Client 2, sending data over the LAN.
7. Canvas Client 4 joins. It is out of range of the existing LAN, and therefore uses its XMPP connection to communicate with the other devices.
8. Clients 1 and 2 from a different application also join the network. They also use the XMPP connection to communicate.

We chose to use two different mechanisms in order to reduce latency. This is because Bluetooth has a much lower latency than using XMPP over a network. We chose this because in our most common use case, all of the users would be in a geographically similar location, e.g. in a lecture theatre, and so it is beneficial to use Bluetooth to provide a better user experience, since latency is something our user wanted to reduce.

We also chose XMPP since it is a well-established protocol. There are many different servers and clients, meaning that we can take advantage of all of this existing technology. This also means that if an organisation wanted to have their own instance of Canvas, they could use their own XMPP server which has appropriate access rights for their needs. We designed our application to be modular, therefore, it is not tied to our XMPP server, so this can be switched out for any other one. We do this by using a framework called `XMPPFramework`. We create a class called `XMPPController`, which uses this framework. An instance of this class represents a connection to an XMPP server.

2.7 Interoperability Design and Implementation

As part of our project requirements, we had to have our application interoperate with that of Group 15. They were also creating a CRDT-powered whiteboard, however they were doing theirs as a web application. In order to do this, we had to define a standard protocol, which our messages conform to. This was difficult as we initially had different ideas, about how the data model of our applications would look. The common protocol consists of three types of messages: Add, Append, Delete. The full details of the protocol can be found in Appendix C. Interoperability represents a classic software engineering trade-off. The advantage of conforming to a protocol, is that it allows other applications to communicate with our application. The disadvantage is that it constrains the possible functionality of our application.

3 Evaluation

3.1 CRDT Evaluation

At the beginning of the project, a major consideration was whether CRDTs are practical in a general software engineering context. Our results were mixed. We found both the JavaScript libraries promising, however there were drawbacks. Using Automerge allowed to have a functioning app based on CRDTs very quickly, without having to worry about the theoretical details. This is espeically true, since it uses the JSON data structure, which is prevalent in industry today.

One drawback we faced was JavaScript, and this situation was worsened by the fact that Automerge and YJS and the only two serious general CRDT libraries today, both of which were written in JavaScript. Serialising and deserialising JSON to interoperate with JavaScript proved to be a major performance burden. However, there were other issues with generic JSON CRDT libraries. These libraries have a complete ignorance of the document structure, which means they cannot take advantage of optimisations e.g. of knowing which keys in a JSON dictionary will be used. A second issue is that sometimes the semantics of JSON may not be the best way to describe your document structure. For instance, JSON has no set data structure. Instead, one must use ordered sequences, but as explained in general this requires each element to have a unique identifier.

It is unfortunate that choosing to use CRDTs offers a stark dichotomy between using one of these generic CRDT libraries that completely abstracts away control, or implementing a CRDT from scratch. One might imagine a middle ground approach as a library that allows you to design a bespoke CRDT library, that is optimised for your application.

Our own experience with designing a CRDT implementation was positive. However, our CRDT was relatively simple with some clear limitations, and did not support things that other more sophisticated whiteboarding apps support. As outlined earlier, supporting them would necessitate a more complex CRDT implementation. One interesting avenue would be to use formal methods to verify the correctness of a particular CRDT implementation. For instance, a proof assistant such as Coq or TLA could be used.

3.2 Scalability Evaluation

As mentioned in the 'Design and Implementation' section under our Networks section, Canvas uses Apple's own framework, `MultipeerConnectivity` for Bluetooth connections between devices. As this is a product that we did not code up ourselves, we are limited by Apple's design which allows maximum 8 devices to be connected together in this way. It could be argued that if we had used an alternative framework or written our own, our Bluetooth clusters would not be prohibited to 8 devices per cluster.

From our tests, we can see that data sent via Bluetooth is transmitted much faster than data sent via XMPP. Because of the way our network is designed, stroke information that is sent between 2 Bluetooth clusters have to go via XMPP. Therefore, if we had been able to increase the number of devices that could be connected within the same Bluetooth cluster, thus increasing the amount of information transmitted via Bluetooth rather than XMPP, then it is possible that this could reduce some lag within the application. However, because `MultipeerConnectivity` is purpose built by Apple and we are developing for Apple, it makes sense for us to use a framework that has a lot of support available already (with `MultipeerConnectivity` being used for AirDrop amongst other things). It also means that `MultipeerConnectivity` can be integrated easily into our existing code.

To gauge the limits of Canvas, we performed some tests to see how many points it would take before Canvas crashes, and to see the maximum number of users that could use Canvas at any one time. We determined that the average number of points that a single user would send per second is 15, by asking random users to draw a generic diagram and recording how many points were stored and how long they took. Then, based on results from benchmarking the performance of our CRDT, we know that we can add 15220 points within a second. $15220/15 \approx 1000$, and so roughly 1000 users can be in a session writing simultaneously, connected via our mesh topology. This, however, does not consider limits with rendering. At the moment, after 20,000 points have been added rendering slows down noticeably. Caching is a possible solution to this, where we would only rerender paths that have changed.

3.3 Shape Recognition Evaluation

Rectangle and line recognition received praise from end users as a useful tool to help with creating diagrams. The tool has been fine tuned throughout its creation to ensure that it recognises rectangles and lines correctly as much as possible and redraws them accurately. When we ran tests on multiple end users, we found that Canvas correctly identified 98% of the straight lines that were drawn, which shows an almost perfect recognition.

The old design for rectangle recognition had a fairly low success rate of 60% and we found that the most common reasons for rectangles not being correctly recognised were the following:

- Drawing the rectangle too slowly. This makes it harder for the tool to recognise rectangles because when you draw more slowly on Canvas, the number of points in the stroke is much higher and therefore, when we run the tests to check if either the points on each edge are on a straight line or if there are a correct number of corners, then there is a larger margin for error.
- When the last edge of the rectangle doesn't meet the starting point.

On the other hand, the new design for rectangle recognition proved to be much more successful, correctly identifying 88.5% of the rectangles drawn. We believe that this much more reliable rectangle recognition provides a much better experience for our users and will lead to this feature being used much more often.

Since the intended use case for the shape recognition feature is for the users to be able to collaboratively draw diagrams, one thing that could be included if we had more time is the ability to type text. Handwriting at a small enough size to fit into boxes as labels is quite hard, especially if a user has a smaller device and therefore, if we were able to include typed text, this could improve the experience for users who primarily want to use Canvas for diagrams. Further to this, a much more complex thing which would be very interesting to try and implement would be handwriting recognition. Since Canvas would be used mostly on the iPad, of which most users would have an Apple Pencil, this would provide a significant amount of information such as pen stroke, pressure and speed of writing. There has also been a vast amount of research into handwriting recognition through neural networks and deep learning, enabling recognition of text to reach 99.73% accuracy [11]. At this point, there are in fact already Swift frameworks that offer some form of handwriting support: DBPathRecogniser [12] and Scribe [13]. Therefore, this is definitely a viable future task that we could consider for Canvas.

Further feedback received suggested that we could include other shapes to be recognised such as circles, triangles or stars. This could have been something that we included had we had more time, however, as previously mentioned, the main use case for the shape recognition feature is for diagrams, such as UML diagrams, and shapes such as stars would not be particularly useful for that.

3.4 User Evaluation

Generally, Canvas received positive feedback in our end user tests, with users saying it contained most of the features that a drawing app would need, with the bonus of it being a good collaborative tool. The UI was easy to understand and navigate, with tool icons being appropriately chosen for their functionality and the setting up of collaborative sessions being a very quick and intuitive process. Users found collaborative sessions to be seamless within Canvas as the app would ensure that users had the best connection to the session, whether it's via XMPP or Bluetooth and the users wouldn't suffer if they experienced poor connection as Canvas would transition them to an offline editing mode, automatically syncing up their changes with other users in the session when they reconnect.

In terms of the actual drawing experience, users reported that the lines they draw are quite smooth, however, as the number of lines on the screen increases, the lines begin to lose their smoothness and users experience a slight but noticeable input lag, which gets worse in a collaborative session when multiple users are all drawing at the same time. Users found that the complete and partial line removal tools worked as intended, however, in cases where there was a huge amount of strokes, there could be a slight lag. Overall, users liked Canvas and all the features it brought with its collaborative nature.

3.5 Reflection

One of the biggest challenges that we faced as we started this project was striking a balance between building a product for end-users that works for them and experimenting and understanding if CRDTs were a viable technology for a collaborative drawing app. We faced a number of technical risks - as a group - we were relatively unfamiliar with iOS development and mobile development, and there was a steep learning curve. However, we picked it up rather quickly, and by the end of the project, we were able to write our own application-specific CRDT implementation in Swift. We also had a lack of knowledge about XMPP, how it worked and how to best set it up for our use. But, by the end of the project users could connect to others nearby via Bluetooth and globally with XMPP.

There were a number of other features we would have liked to add to the app throughout the course of the project, although other features ended up taking priority. However if we had more time, we would like to add more drawing tools, such as different brushes, a fill tool or the ability to import pictures that users can then annotate. For collaboration features, we would like to give users a comprehensive history log of all changes that have been made in that session, with the option for them to revert back to a previous state. Furthermore, we would like users to have the option of more control over the automatic syncing of CRDT documents, to ensure they get exactly what they want when users are resyncing.

4 Ethical Considerations

One major ethical issue we had to think about was privacy. One way this came into play was in the design of our network and how users interacted with others. We made the decision that the standard version of the application, which we would deploy to users, would run on our central server. The server has no restrictions on who can join which room, and therefore anyone can work together with ease. We decided on this because the point of our application was mainly to focus on efficiency and research of CRDTs, and the use cases would most likely be in the workplace or in education, where the average user is more likely to exploit the application.

However, thinking ahead, we also made it possible for groups of people, for example a company, to use their own XMPP server with the application. This means that they could use their own login and security for users, and would be able to set permissions on who can access each room etc. Since us and our client did not see this as a priority, we left this as an extension, but structured the code in a way such that it easy to swap out

which XMPP server is being used, and since permission logic is left to that server, would allow for an app with privacy setting customisable by the organisation using Canvas.

The next ethical consideration we had was how to handle the interoperability between Bluetooth and the XMPP server. Since we wanted to abstract this away from the user, the model we used, as stated in the Design and Implementation section, required devices to connect to each other via Bluetooth without having to explicitly ask the user. Initially we were worried about this, since we were unsure if some users would object to this happening without their knowledge. However, since it drastically improved the efficiency of our app, we decided it was worth the possible objections. Therefore, one thing we did was to display to the user which devices it is connected to via Bluetooth. This means that if they want, they can disable Bluetooth and exclusively use the XMPP server, sacrificing some performance optimisations. We also made it so that devices only connect to other devices which are in the same room. This means redundant connections are prevented, and so reduces the potential for exploitation of the Bluetooth connections.

Another ethical consideration we had was the monitoring and moderation of content. This would be the detection and prevention of offensive content being spread using our application. We decided that since this is more of a research-based project, we would not focus on actively implementing features to prevent this. However, we do keep track of all data sent on out XMPP server, so it would be possible to see which user sent a message if needed. One way we could work on this problem if we developed this application commercially would to use image and text recognition. We already do something similar to this with our line and shape detection features, and therefore it would be simple to expand this to use an API which can detect offensive images or text. Since we manage the XMPP server, we would be able to block certain users from using the app, and if organisations managed their own XMPP servers, they would have the ability to do this, as well as more finely tuning access rights based on their needs.

References

[1] Collaboration Unleashed: Softchoice Research Study. https://www.softchoice.com/research-studio/enable-end-users/employee-collaboration-research/. Accessed: 2020-01-04.

[2] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[3] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 399–407, New York, NY, USA, 1989. Association for Computing Machinery.

[4] Du Li and Rui Li. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work (CSCW)*, 19(1):1–43, Feb 2010.

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, July 2011.

[6] Chengzheng Sun, David Sun, Agustina, and Weiwei Cai. Real differences between OT and CRDT for co-editors. *CoRR*, abs/1810.02137, 2018.

[7] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Tablets - Statistics & Facts. https://www.statista.com/topics/841/tablets/. Accessed: 2020-01-04.

[9] Tablet shipments market share by vendor worldwide. https://www.statista.com/statistics/276635/market-share-held-by-tablet-vendors/. Accessed: 2020-01-04.

[10] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017.

[11] Have we solved the problem of handwriting recognition? https://towardsdatascience.com/https-medium-com-rachelwiles-have-we-solved-the-problem-of-handwriting-recognition-712e279f373b. Accessed: 2020-01-04.

[12] DBPathRecogniser. https://github.com/didierbrun/DBPathRecognizer. Accessed: 2020-01-04.

[13] Scribe: A Handwriting Recognition Component for iOS. https://dzone.com/articles/scribe-a-handwriting-recognition-component-for-ios. Accessed: 2020-01-04.

[14] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

A Order Theory and CRDTs

This section provides a primer to order theory and conflict-free replicated data types (CRDTs).

Definition 1 *A partially ordered set, or poset, $\langle S, \leq \rangle$ is a set S coupled with a relation $\leq$, with the following properties:*

- *Reflexivity,* $\forall x \in S \ (x \leq x)$
- *Anti-symmetry,* $\forall x, y \in S \ (x \leq y \wedge y \leq x) \rightarrow x = y$
- *Transitivity:* $\forall x, y, z \in S \ (x \leq y \wedge y \leq z) \rightarrow x \leq z$

A simple example of a poset is $\langle \mathbb{Z}, \leq \rangle$, i.e. the integers equipped with the normal less than or equal relation.

Definition 2 *A join-semilattice is a poset $\langle S, \leq \rangle$, with a binary operation, join, that is:*

- *Associative:* $\forall a, b, c \in S \ ((a \,\mathrm{join}\, b) \,\mathrm{join}\, c = a \,\mathrm{join}(b \,\mathrm{join}\, c))$
- *Commutative:* $\forall a, b \in S \ (a \,\mathrm{join}\, b = b \,\mathrm{join}\, a)$
- *Idempotent:* $\forall a \in S \ (a \,\mathrm{join}\, a) = a$

CRDTs are introduced in [5] in two different forms: state-based CvRDTs and op-based CmRDTs. State-based CvRDTs are where upon changes different nodes send their entire document to other nodes, and the entire document structure is merged. CvRDTs do not assume that messages from the same agents will arrive in a causal order, nor do they assume that messages will arrive once. CvRDTs work by having documents which are members of a join-semilattice. The operation to merge two CRDTs is the join (least-upper bound) operation of a join-semilattice. There is one additional requirement, that the join operation is monotonically non-decreasing. In other words:

$$a \,\mathrm{join}\, b \leq a$$

In contrast, in CmRDTs the operations that describe changes to be made are exchanged between nodes. In contrast to CvRDTs, it is assumed that there is a reliable communication channel that ensures that messages are delivered in a causal order and only once. Unlike CvRDTs there is no merge stage, instead there is an update stage which takes a change and modifies the document structure. There is a split into two phases the *prepare-update* phase and the *effect-update* phase. The prepare-update phase at the source node originating the change. It takes the original operation and the source node's document structure to produce a new update. Note this is a side-effect free function, so the document structure isn't modified. This new update is sent to all other nodes,

and used to update the document structure, in the effect-update phase. The reason for the prepare-update phase, is to allow the update to incorporate information about the source node's document structure. It can also be used for adding timestamps to updates, which is a strategy sometimes used for handling conflicts. The only condition is that updates are commutative. In other words, it doesn't matter which order you apply updates, the resulting document structure will be the same.

Note that there exists a special class of op-based CmRDTs explored in [10], where the prepare-update phase simply returns the original update, and doesn't do any modification. This is called a pure op-based CvRDT. The CRDT that is used in our application is a pure op-based CvRDT.

I shall proceed to outline some common CRDTs found in literature, that helped inform the design of our own CRDT. [14] is an excellent survey of these data structures:

- A G-Counter is a a counter that can only be incremented. To implement as a CvRDT, it is represented as a 'vector clock', this is a list of integers, with each index corresponding to a node. A node only updates the value at its index. The least upper bound operations take a component-wise maximum. The value of the counter is the sum over all the components. For instance, $[0, 1, 1]$ and $[0, 2, 0]$ would be merged to produce $[0, 2, 1]$.

- A PN-Counter is a counter that can be incremented an decremented. It can be constructed as a CvRDT from two G-Counters, one used to store increments and another for decrements.

- A G-Set is a set to which items can only be added. it can be as a CvRDT with the merge operation being set union.

- A 2P-Set is a set allowing the addition and removal of items. It is made from two G-Sets, one for added items A and one for removed items R. The value of the set is $A \setminus R$. In other words, it has the 'remove-wins' property. Once an element has been removed, it cannot be readded.

- A LWW-Register consists of a datum and a timestamp. It can be implemented a CmRDT. The prepare-update phase adds a timestamp to the update. The effect-update phase only replaces the value of the datum, if the timestamp of the update is newer than the timestamp stored. This data structure uses the 'last-write-wins' strategy.

- A LWW-Set is a set allowing the addition and removal of items, including the addition of previously removed items. It is made from an add and remove set, where values are equipped with timestamps. An item is only considered removed if the timestamp associated with the item in the remove set is greater than the timestamp associated with the item in the add set. This allows a removed item to be readded. This data structure also uses the 'last-write-wins' strategy.

- An OR-Set is a set also allowing the addition and removal of items, including the addition of previously removed items. Rather than using timestamps, values are associated with a set of tags (unique identifiers) in the add and remove sets. When

an item is removed from a set, it is removed only with the tags visible at the source node. An item is only considered removed, if all the tags in the add set, are also in the remove set.

B Performance Evaluation Graphs

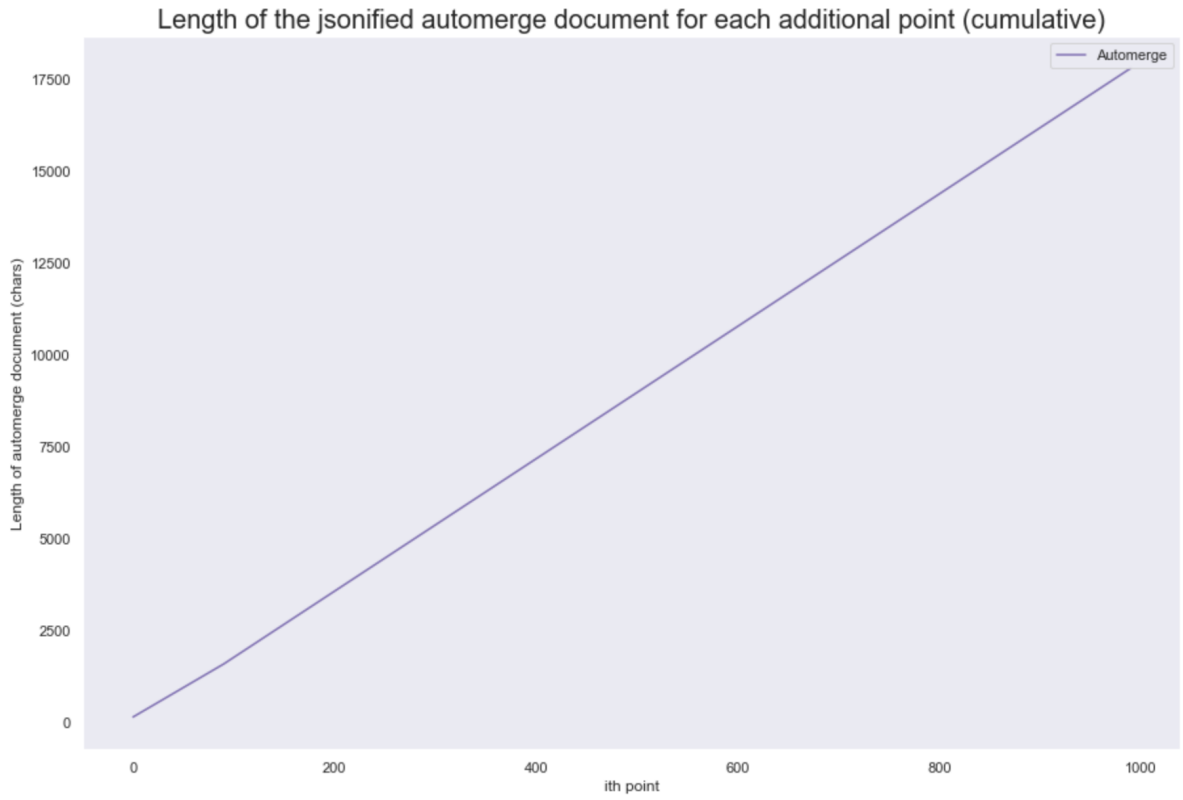


Figure 1: The size of an Automerge document grows linearly with the number of points added to a stroke. The average number of chars added to the document per point is 17.8.

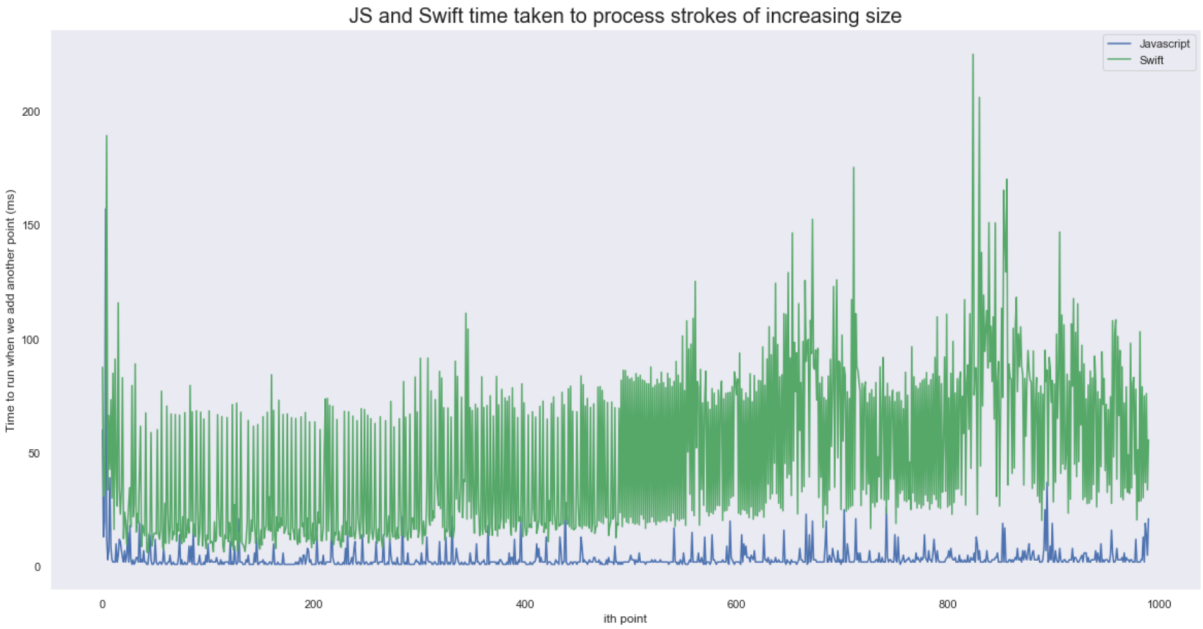


Figure 2: Time spent in JS and Swift runtime adding 1000 points to a stroke. This shows that whilst Automerge itself isn't a bottleneck and is quite quick at processing changes in the CRDT, however, the JS callback in Swift runtime adds on average an extra 49ms, as opposed to 3.7ms within JS runtime.

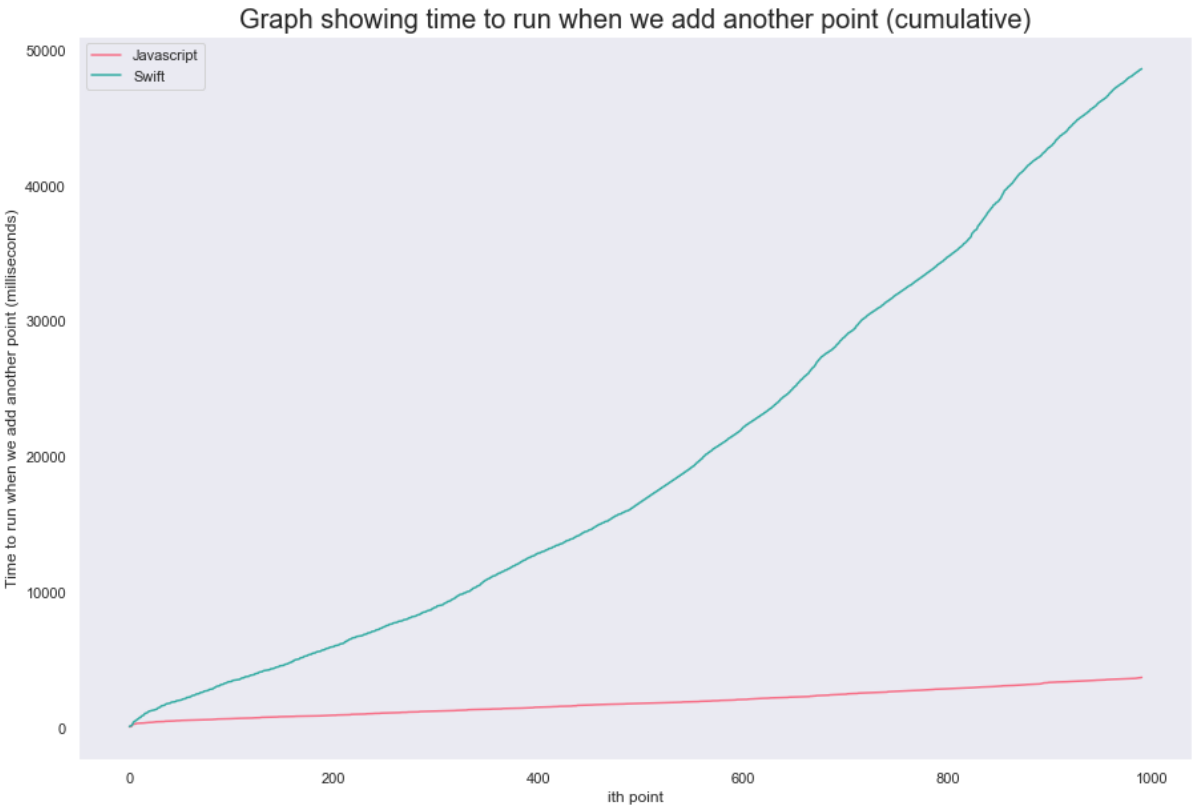


Figure 3: For each point added to a stroke, the majority of time is spent in Swift runtime as opposed to JS runtime and this exhibits quadratic growth which will eventually cause a bottleneck when enough points are added.

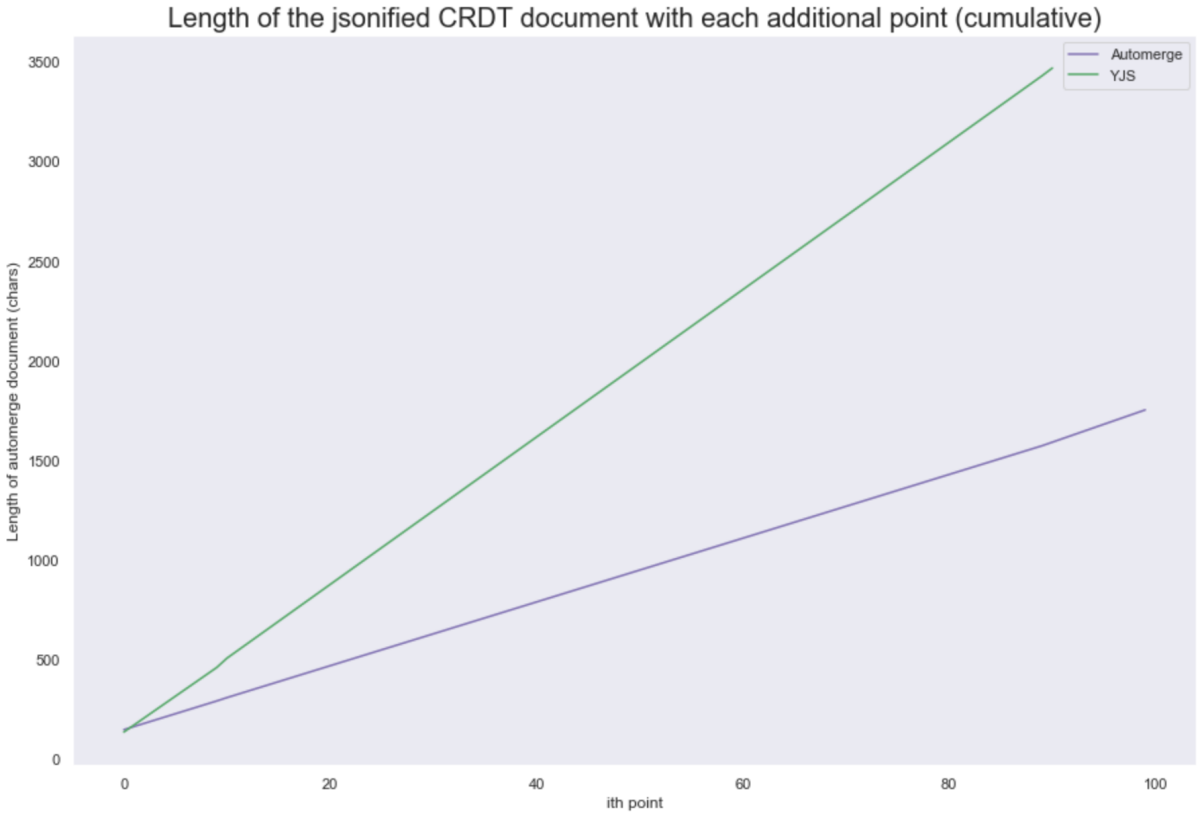


Figure 4: Both documents experience a linear increase in their document sizes when more points are added to a stroke, however, YJS documents increase at a greater rate, which suggests that more time will therefore be spent processing these documents in both the JS and Swift runtimes.

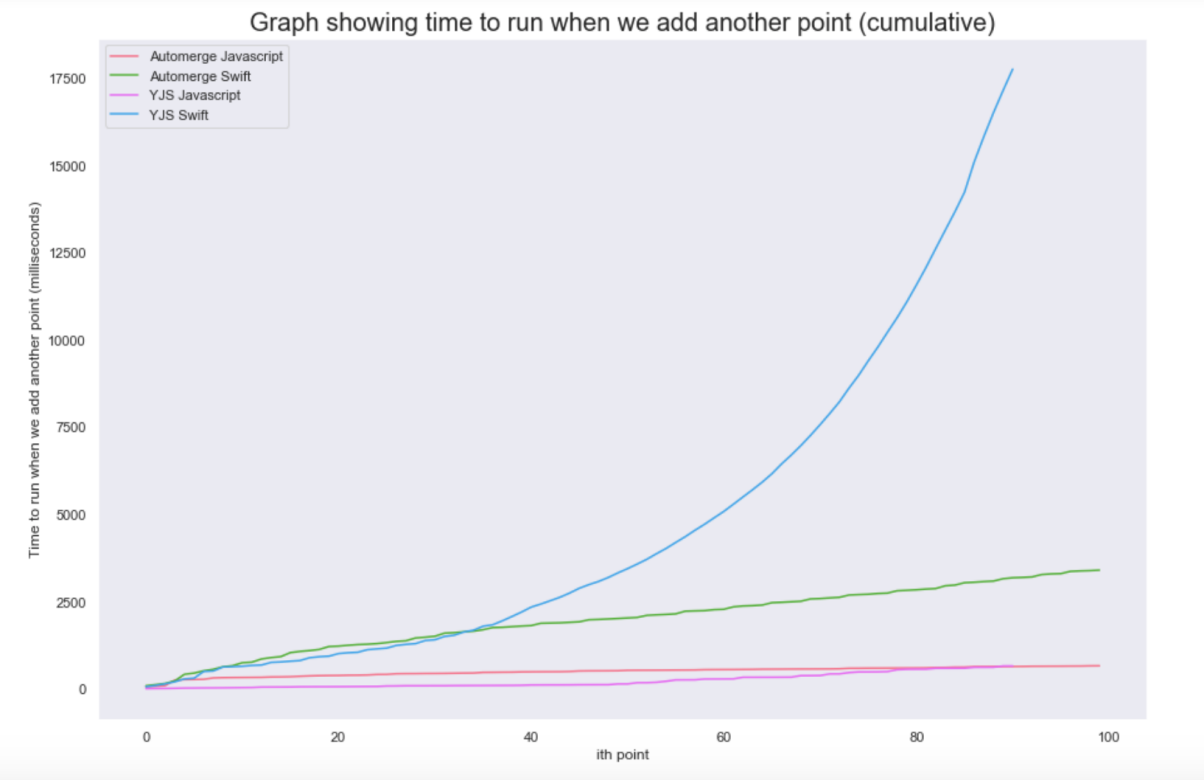


Figure 5: Over 100 points, YJS spends on average 7.2 ms in JS runtime and 195 ms in Swift runtime, showing that Automerge greatly outperforms YJS. It's worth noting that testing YJS took too long and so we couldn't obtain results for adding 1000 points.

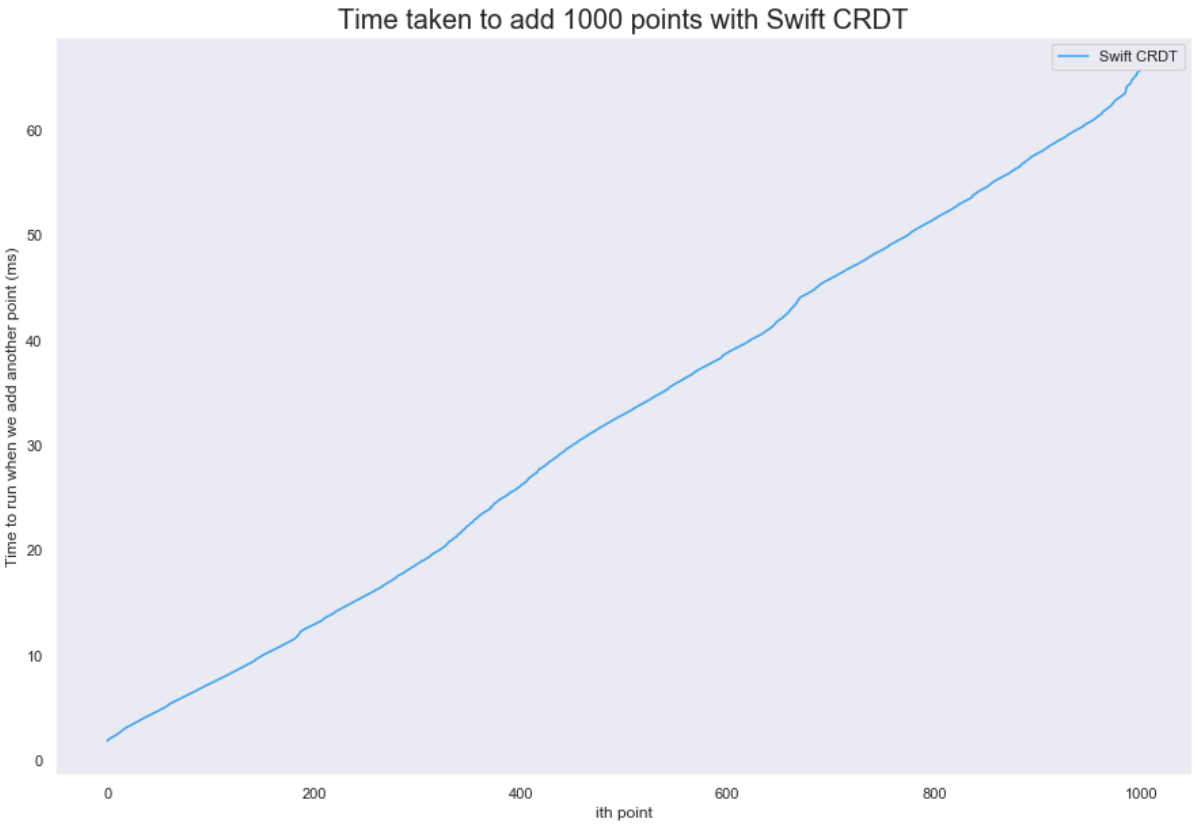


Figure 6: The Swift CRDT is app-specific and so we can add various optimisations to it that we couldn't for a general-purpose CRDT library. Because of this, The Swift CRDT has a linear increase in processing time for each point as opposed to the quadratic growth that both YJS and Automerge had. The total runtime for this Swift CRDT is 65.7 ms.

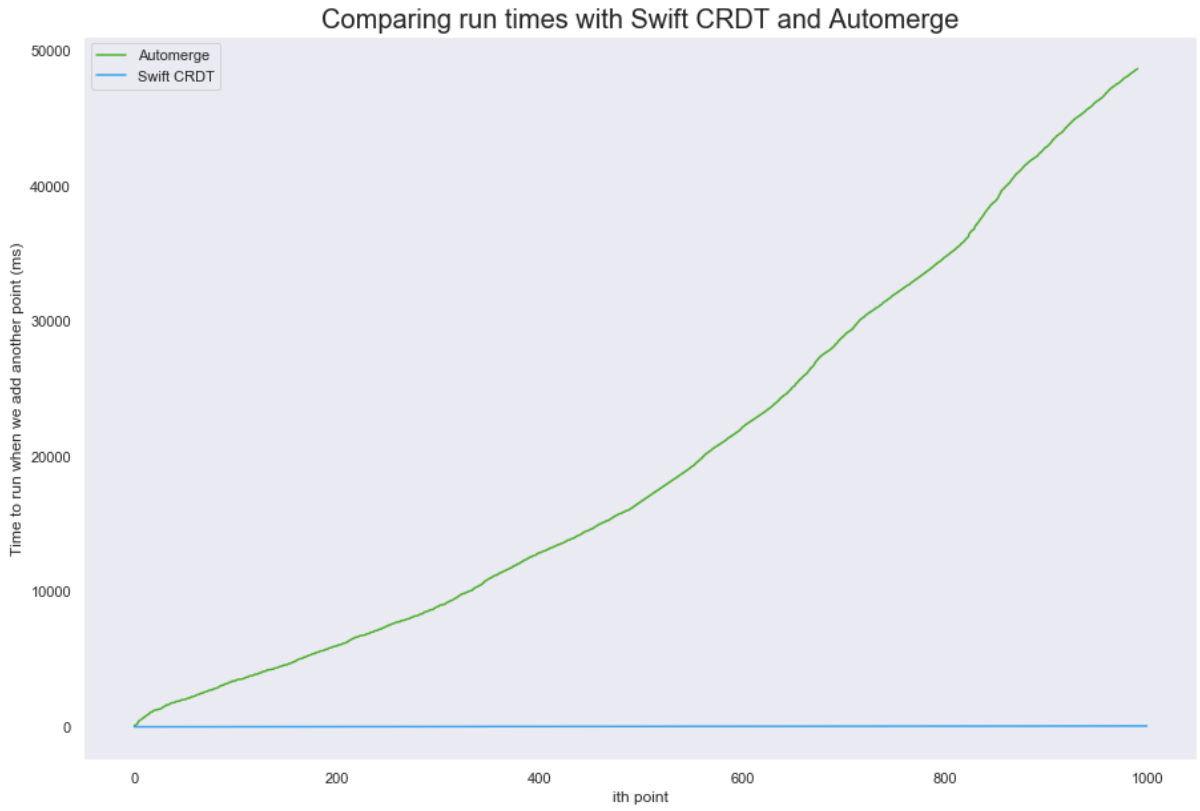


Figure 7: There is clearly a huge benefit by having the CRDT running in Swift rather than JS in our iOS app, and by doing this the bottleneck of switching between JS and Swift is eliminated and the app experiences much less lag.

C Protocol

Below are examples of messages that conform to the common protocol agreed upon by our group and group 15. There are three types of messages `ADD`, `APPEND`, `DELETE`. Each message is a JSON object, with a property `type`, that contains one of these values.

Note that throughout our protocol, points are represented as two-element arrays of floating-point numbers. The first element is the x component. The second element is they y component. Also colours are represented as three-element arrays of integers between 0 and 255 representing the red, green and blue components. Also the unique identifiers of strokes are encoded as strings.

```
{
    "type": "ADD",
    "identifier": "ABC_123",
    "weight": 1,
    "colour": [50, 0, 0],
    "start": [0, 0]
}
```

The `ADD` has four other properties: `identifier`, the unique identifier of the stroke;

`weight` a floating point number that is the weight; `colour` that represents the colour of the stroke; `point`, the starting point.

```
{
    "type": "APPEND",
    "identifier": "ABC_123",
    "points": [[127, 45], [35, 78]]
}
```

The `APPEND` has two other properties: `identifier`, the unique identifier of the stroke; `points`, a list of points.

```
{
  "type": "DELETE",
  "identifier": "ABC_123",
  "start_offset": 15,
  "end_offset": 67
}
```

The `DELETE` has three other properties: the `identifier`, the unique identifier of the stroke; `start_offset` a floating point number that is the start of the interval to remove; `end_offset` a floating point number that is the end of the interval to remove. Note that if `start_offset` is equal to or greater than `end_offset` this is interpreted as an empty list.

D Screenshots

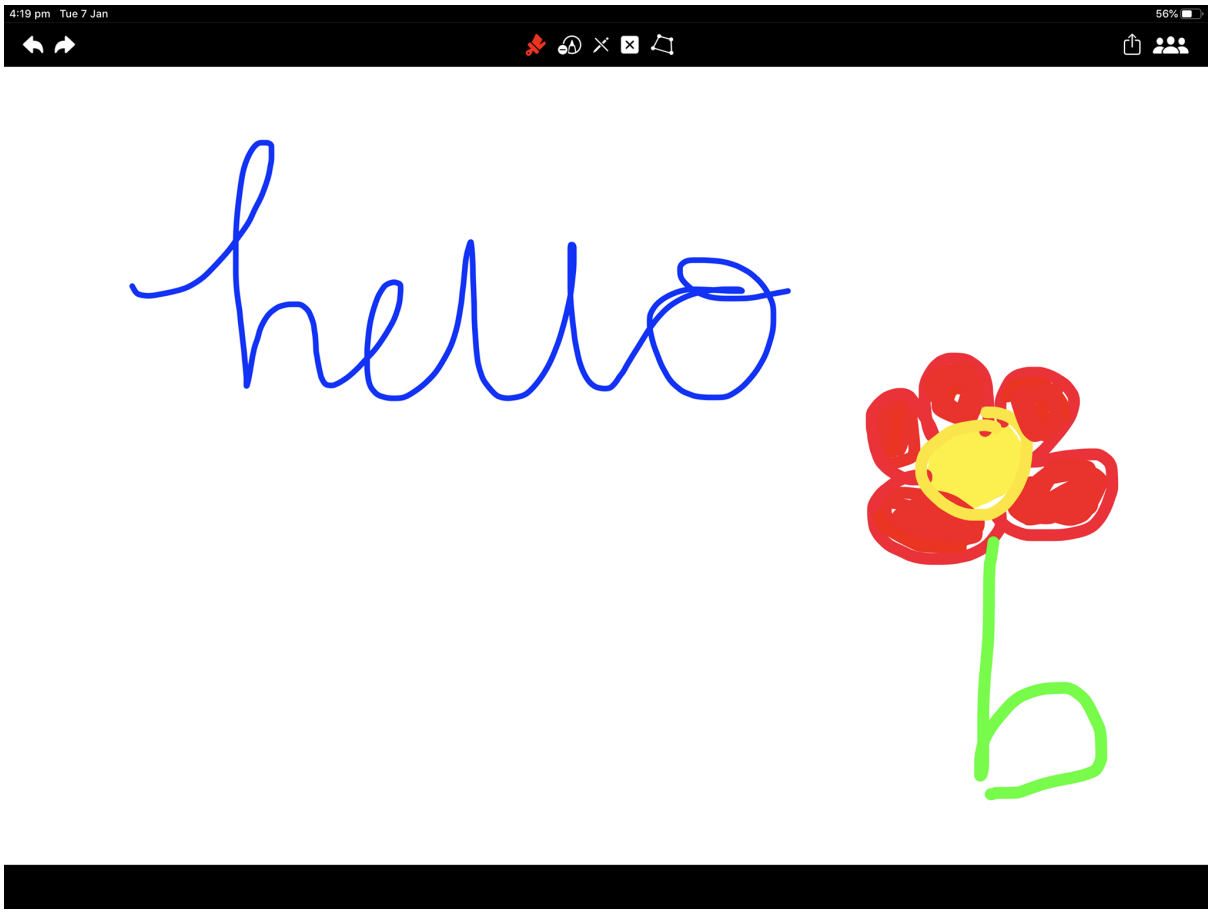


Figure 8: This is the base UI for Canvas, allowing users to select different drawing tools, undo and redo their drawings and collaborate with other users.

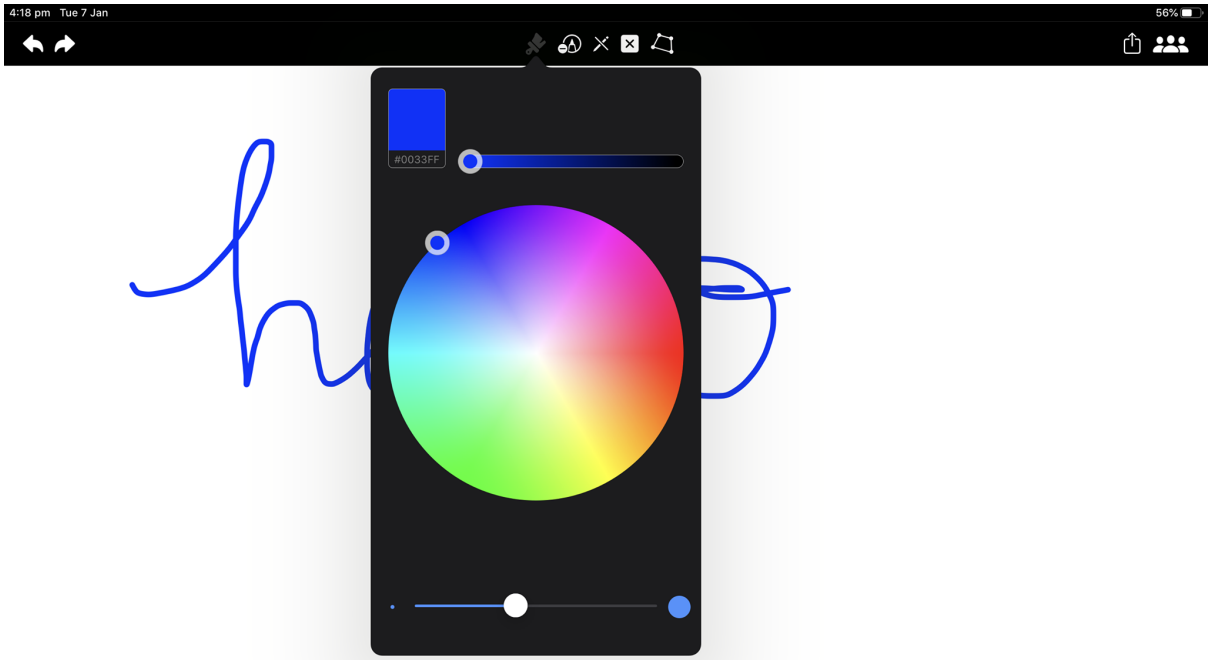


Figure 9: This shows the colour picker tool, overlaid on the main canvas, allowing users to pick a stroke colour from the palette, change its brightness and change the stroke's thickness.

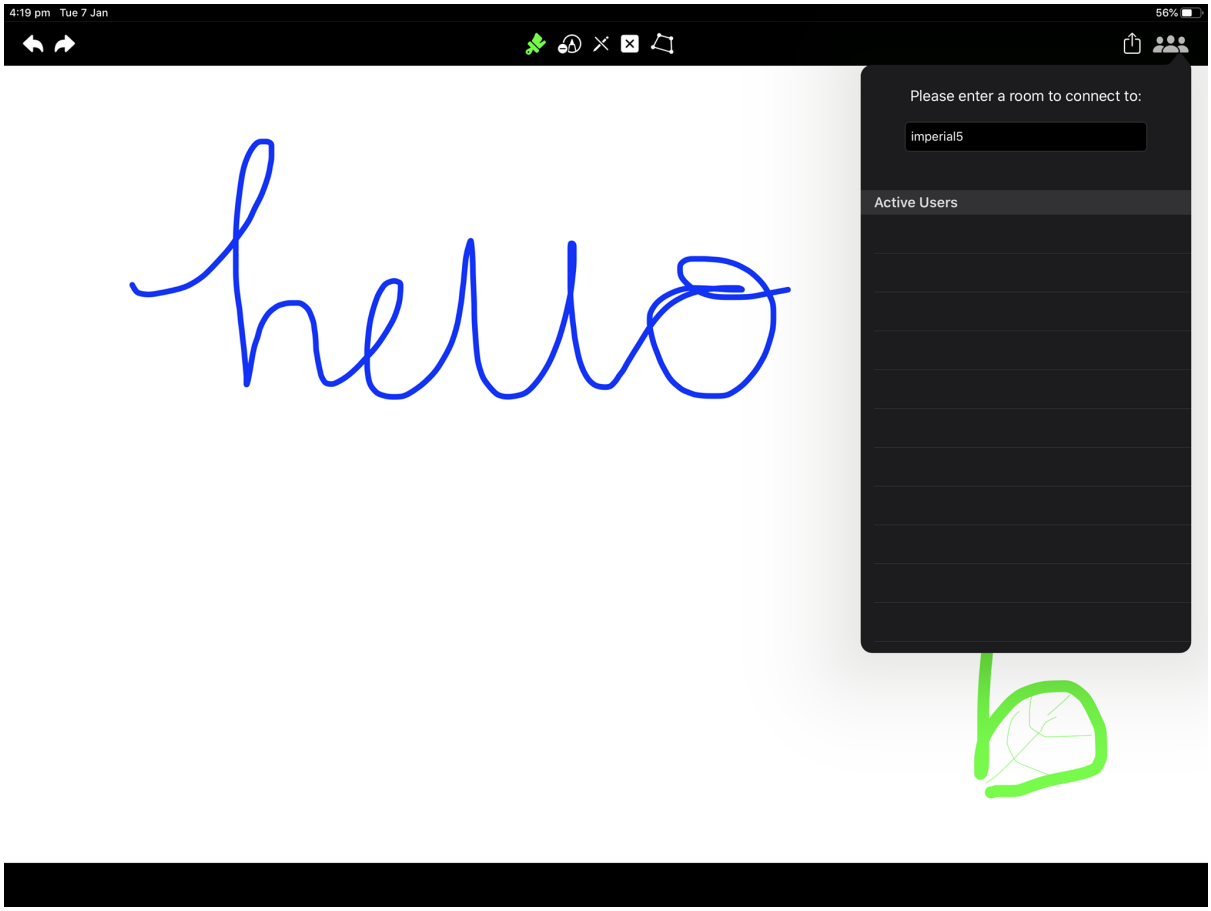


Figure 10: This shows the session management tool, overlaid on the main canvas, allowing users to pick a room to connect to and see which users are in the session.