2014-06-05

Application Sharing from Mobile Devices with a Collaborative Shared Display

Richard S. Shurtz
Brigham Young University - Provo

Application Sharing from Mobile Devices with a Collaborative Shared Display

Richard S. Shurtz

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

June 2014

ABSTRACT

Application Sharing from Mobile Devices
with a Collaborative Shared Display

Richard S. Shurtz
Department of Computer Science, BYU
Master of Science

With the increasing ubiquity of smartphones, tablets, and large pixel-rich displays, there are many exciting new possibilities for using these devices for collaborative work. While there already exist hardware and software that support communication and interaction between mobile devices and shared displays, application sharing in these scenarios is still limited and inflexible. We present a new method of application sharing which allows collaborators to download clips or snapshots of each other's applications. These snapshots can be used to re-launch and resume the shared application back to the state it was in when it was shared. We have built a system that supports sharing, annotating, organizing, and downloading these applications to and from a LearnSpace server. We have built an application framework which allows Android applications to be built for this system while only requiring minimal changes to the program. We also describe solutions for extending our solution to new types of collaborative displays and to other application platforms.

Keywords: mobile devices, Android, collaboration, shared display, application sharing

Table of Contents

Chapter 1 – Introduction

Smartphones and tablets are quickly becoming the most ubiquitous general purpose computers in the world. Similarly, large high-resolution displays are becoming less expensive and more pervasive. With the growth of such technologies, there are new opportunities and possibilities for performing co-located collaborative work in convenient, flexible, digital spaces. Collaborative workers need to be able to share content and applications from their mobile devices on large shared displays. They need to be able to organize, analyze, and critique each other's work. They also need to take shared content away with them on their own device, continue evaluating and critiquing, and then be able to bring it back together at a future meeting. To realize such situations, we need a collaborative environment designed to support these interactions on a large shared display and an application framework that allows mobile applications to be full participants in that environment.

Face-to-face collaboration is effective and even necessary in a variety of productive tasks: meetings, design work, team discussion, classroom learning, group studying, etc. Collaboration in these situations normally involves discussing each contributor's ideas, critiquing and exploring those and new ideas, and partitioning of future work [20]. For the most part, such activities still rely heavily on physical media: papers, pictures, sticky notes, whiteboards, etc.

However, because smartphones and tablets afford additional power, flexibility, and convenience, there are many benefits to moving collaborative work into such a digital realm. Collaborators will be able to perform group work at a moment's notice and in less formal situations. They will be able to take the entire discussion with them when it is over, and can continue to work on the project away from the group. Students, designers, professionals, and many others who need to work in face-to-face collaborative environments would benefit from a framework that allows for applications to be shared, annotated, and collected using a large shared display. Figure 1 shows an example of such a space and the flexibility such a solution could provide.

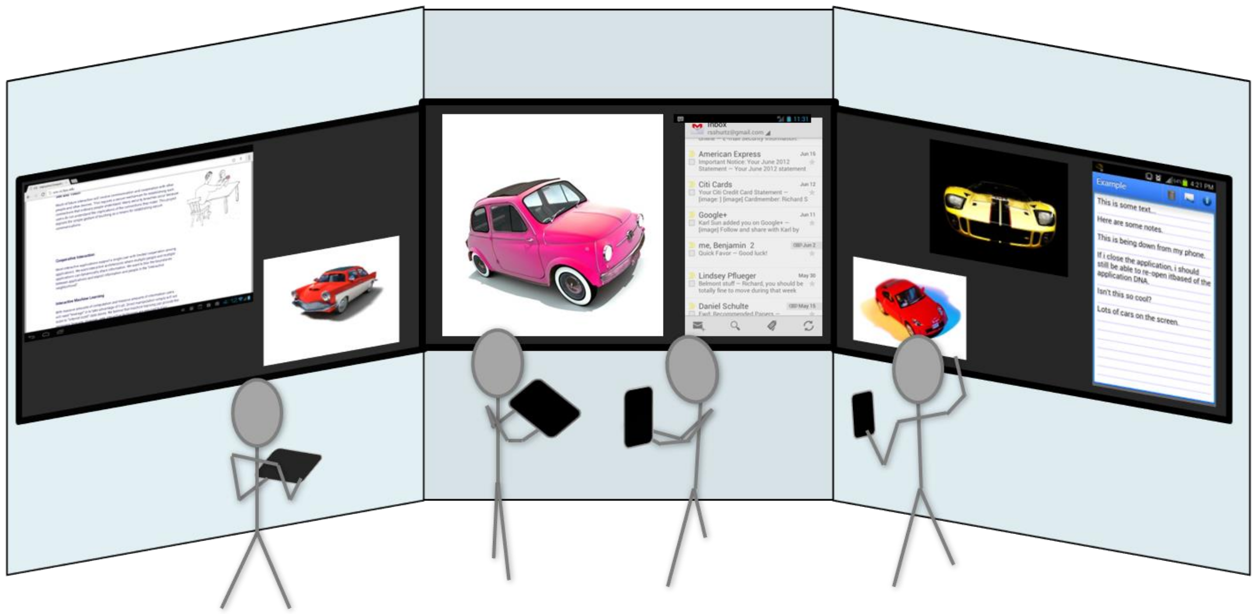


Figure 1 – An example of a small group meeting using handheld mobile devices and a large shared display to do collaborative work.

1.1 Example: Student Study Session

Consider the case of three students studying together for their final exam. They all bring their own devices – an Android tablet, an Apple iPad, and a Windows laptop – to a room equipped with two wall mounted 70” displays.

To start, Henry opens his email application and finds the study guide the teacher sent to the class. He shares the email application to the shared display. All three students look it over. As they do, each uses his own device to draw over and “highlight” topics on the study guide he would like to discuss with the group. After they have all finished highlighting, the group discusses each topic, starting with those with the most highlights.

Each student pulls open his notes or relevant materials on his device and shares them to the screen for all to see. Henry has a PDF of the teacher's slides, so he shares his PDF viewer application. Kristen wants to share her own notes on the subject, so she opens the note taking application and shares that to the display. There is no need to export to some common media. Also, the screen is able to support any number of shared applications, and the students are free to position or resize any shared item to fit the needs of the discussion. Figure 2 below shows how this discussion might look.

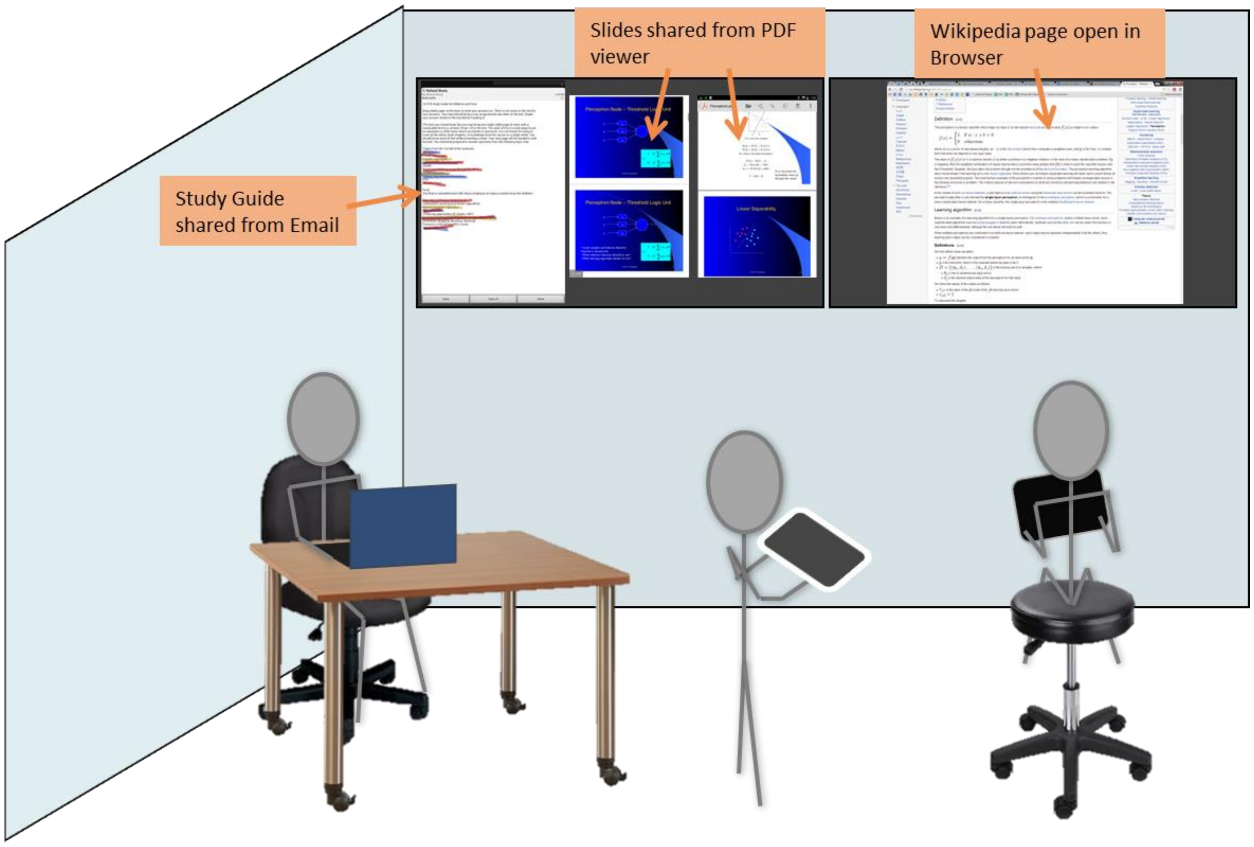

Slides shared from PDF viewer
Wikipedia page open in Browser
Study Guide shared from Email

Figure 2 – Students share a variety of applications to the shared displays as part of a group study session

For one topic, Frank has some questions, but Kristen and Henry have a good enough understanding and explain the concepts verbally to him. Because Frank is afraid he might forget what they've explained, he downloads a copy of Kristen's notes so that he can review them the night before the final. Frank can even capture an audio annotation of Henry explaining the concept.

For another topic, all three students feel they do not understand the topic very well. Henry opens up his browser on his device, shares it to the display, and navigates to a Wikipedia page on the topic (see Figure 2 above). With him using the browser, all three students read, discuss, and suggest answers together. At an informative point during their exploration, Kristen downloads a snapshot of the shared browser to her own device so that she can refer back to it at a later time.

On the night before the exam, Kristen reviews one more time by opening the collection of items she downloaded from the review session. She sees the snapshot of the Wikipedia webpage for the topic they had understood poorly. She uses the snapshot to automatically launch the browser on her own device to the same webpage and to the same scroll position on the page. She re-reads that section of the article and then follows a link to gain some further understanding.

Based on this example, the key features required for a successful system are as follows:

- Users are able to share arbitrary applications to the display
- Users can create ink and audio annotations on shared applications for use both on the shared display and also on personal devices
- Multiple users can all connect and share simultaneously
- Users can download any shared application to take away from the meeting with them
- Users can resize and reposition shared applications to meet the needs of the discussion
- Users can use the downloaded application "snapshot" to automatically resume the application from the state in which it was downloaded

1.2 Example: Professional Automotive Interior Design Team

As a second example, consider a team tasked to design the interior for next year's car model. In order to develop a coherent design, the team meets together frequently in relatively informal and sometimes impromptu meetings. Let us examine how one of these meetings might proceed early in the design brainstorming process, and then later during the design refinement process.

1.2.1 Initial Brainstorming

Before meeting together, each team member explores different areas of design (furniture, architecture, home interior design, etc.) and collects these ideas on his personal device. These could be pictures taken at the furniture store, snapshots of his browser, etc. The team meets together in a room with a single high-resolution projector to discuss and choose what elements they want to incorporate into their design. They each have a company-issued tablet.

One team member, Clara, starts by sharing the collection of all her images, web pages, etc. to the display. (Altogether, she has a dozen or so.) The whole team takes a few minutes to consider all the shared items and draw over any parts they find particularly interesting. Team members ask questions, and discuss in pairs or threesomes as they find ideas they would like to explore further. They also use their tablets to reposition and organize the many shared ideas to attempt to combine similar designs to help with the critiquing process.

During this time of relatively disorganized exploration, Bill finds that he really likes the shape of the back of a couch that Clara found online. He uses the shared browser snapshot to launch the browser on his own device to see if there are any other pieces of furniture that belong to the same set. The browser opens to the same web page viewing the same information. Bill then looks at some related items on the website and finds a recliner that he thinks he can use for

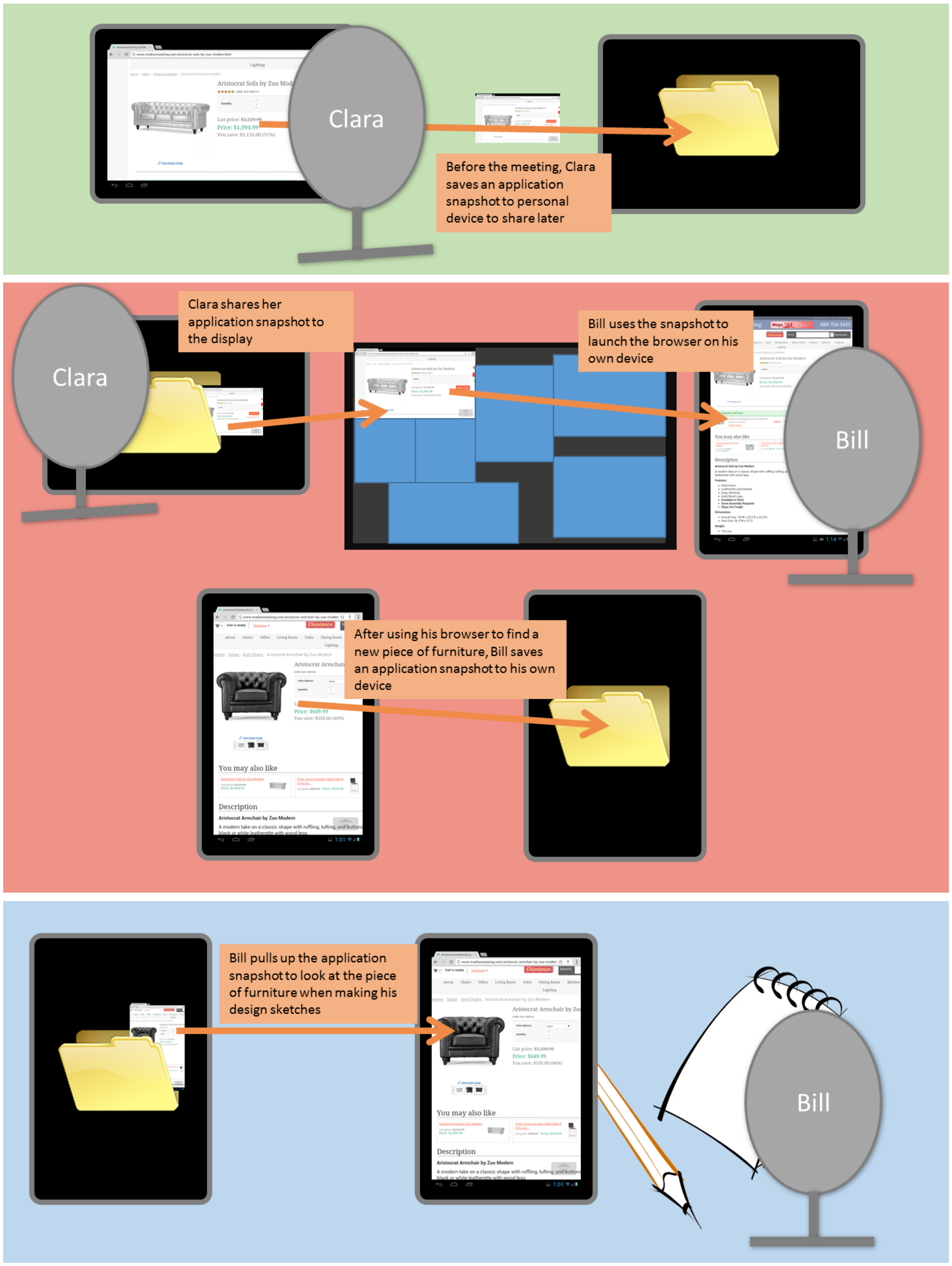

Clara
Before the meeting, Clara saves an application snapshot to personal device to share later
Clara
Clara shares her application snapshot to the display
Bill uses the snapshot to launch the browser on his own device
Bill
After using his browser to find a new piece of furniture, Bill saves an application snapshot to his own device
Bill pulls up the application snapshot to look at the piece of furniture when making his design sketches
Bill

Figure 3 – Summarized workflow of using and sharing a web browser: starting with Clara's pre-meeting work, to sharing during the meeting, and finally to Bill's work after the meeting.

inspiration in his design of the driver and passenger bucket seats. He saves that browser application screenshot to a folder on his own device so he can use it later as he sketches his own seat design. Figure 3 summarizes this workflow.

The meeting continues with each team-member sharing their collected ideas to the screen. The team considers the ideas, explores them further, decides which are useful, and then divides them up for individual team members to use in their future work.

1.2.2 Design Refinement

After the team has pieced together the general design of the interior, team members will start working with sketches and 3D computer models to refine their designs. The team decides to meet to have everyone report and show their progress. Peter shows two different designs for the car's console he is considering with a couple prototype 3D models. The team discusses them and decides that he should pursue the second of the two. Sarah is in charge of designing the car's door panel, and during the meeting start to wonder if she can design the door handle to match better with the knobs on the console in Peter's design. As Peter is using his application and showing the team, Sarah downloads a snapshot of the modeling application so she can work with it later (see Figure 4 below).

After the meeting, Sarah experiments with a few different designs and finds a combination of handle and knob that work very well together. This, however, requires that Peter change his design slightly. She meets with him to share her new designs. She shares her snapshot of his model to the display. He is able to open up his modeling application from the snapshot right at that point and immediately being making adjustments to the knobs.

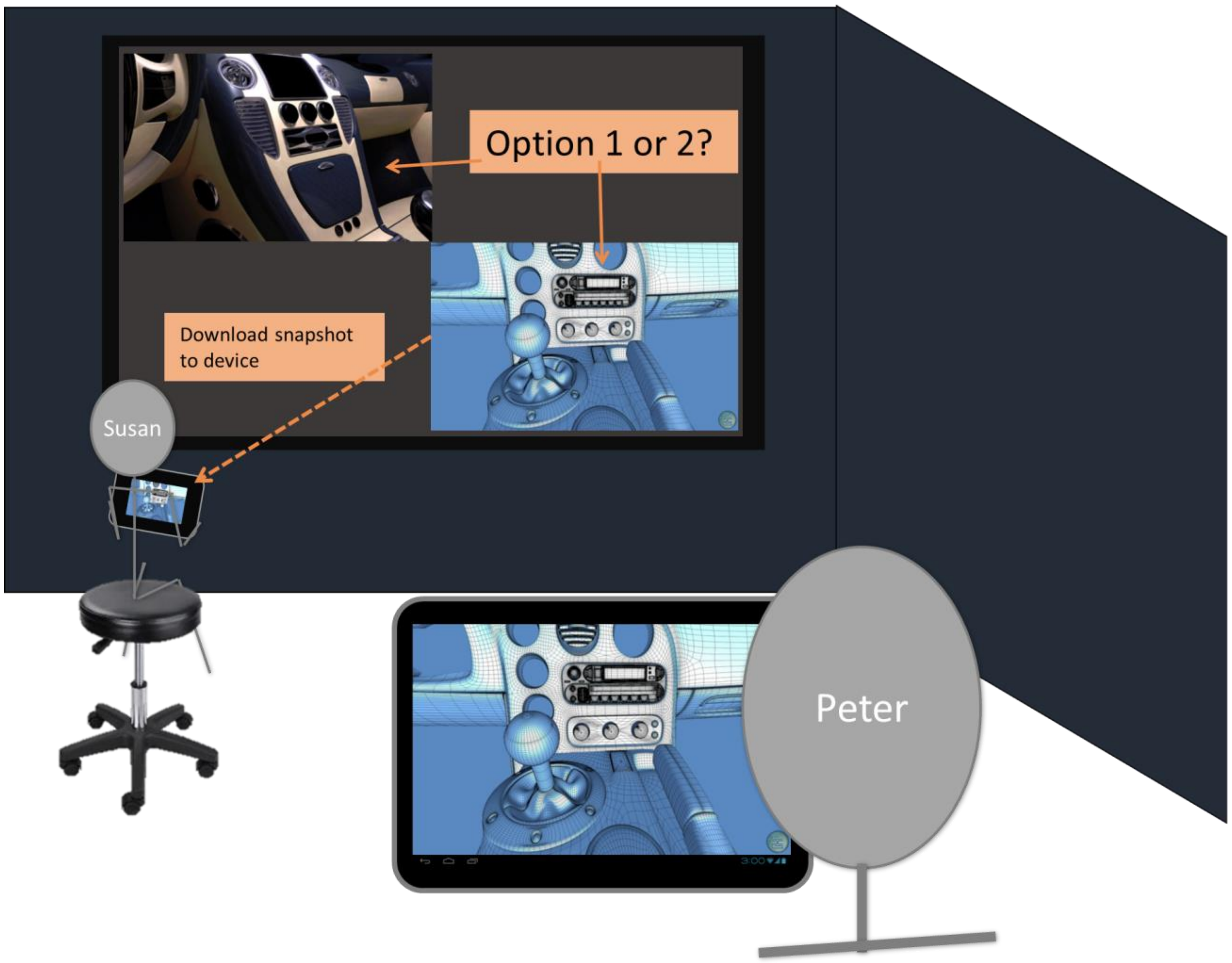


Figure 4 – Sarah downloads a snapshot of the application Peter is sharing to the display.

Based off these examples, a successful system will have the following key features:

- Users can save application snapshots to their own device
- Users can share a collection of shared snapshots to the display
- Users can use shared snapshots on the display to launch the application on their own device
- Users can use the shared display as a place to organize visually
- Users can re-share downloaded snapshots to the display

- Application snapshots with device dependent data can be re-launched on their original device.

1.3 Solution Requirements

The examples above illustrate the requirements for systems and frameworks that allow for collaboration to take place from mobile devices in a shared display setting. These requirements can be broken into two main components. First, users need a collaborative space that allows sharing, organizing, and annotating through the shared display. Secondly, there needs to be a consistent, flexible method for arbitrary applications to participate in the collaborative space.

1.3.1 Provide a Collaborative Space

A good solution will include a ‘collaborative space.’ A collaborative space is simply an application where multiple users can interact together. Often, this is implemented across multiple devices with multiple pieces of software all working together. In the cases describe above, users want a collaborative space that makes use of a large shared display. We discuss the requirements for such a collaborative space in the following sections.

1.3.1.1 Mobile Devices as primary computing device

Users are accustomed to performing most of their work on their own device: working with their own applications. When it is time for them to meet with their teammates, they should be able to use those same applications on the same device and be able to fully participate in the discussion using those tools. Their mobile devices will be the source of content shared, as well as the mechanism to interact with that content.

1.3.1.2 Multiple users connect and share simultaneously

Rather than simply using the large display for a single user to make a presentation, users need the ability to use the shared display as a focal point of a multi-person discussion. All team members should be able to contribute to the discussion freely. The space should be open for many users to share many items simultaneously.

1.3.1.3 Users take away shared items away from the meeting

A common trend in the examples above was that during the meeting, applications were shared from one user and were then useful to another user for his future work. The collaborative space needs to support any user downloading any shared item. He should then have access to that item once the meeting is over. They should be free to work with that item and bring it back to future meetings if needed.

1.3.1.4 Collaborative Interactions

Users need to be able to interact with applications shared to the display from their personal devices. They need to be able to position, resize, and organize them on the shared display to meet the needs of the discussion. They also need to be able to annotate the shared applications in a way to be visible to the group on the shared display but also downloadable so that users can take them with them away from the meeting for future reference.

1.3.2 Arbitrary Applications can Participate in the Collaborative Space

The second set of requirements of a good solution is that it allows normal, everyday applications to participate in the collaborative space. Reflecting on the two story examples above, these applications and the way they are shared must meet the following requirements.

1.3.2.1 Visual mirroring

The content of the application needs to be shown to the display and updated in real-time as the user uses the application. For example, as one user is researching a topic in his browser, his teammates should be able to see what he is doing to be able to provide feedback and direction (see Figure 2 above). The visual mirroring needs to update quickly, but without being a drain on the mobile device's resources.

1.3.2.2 Any user can work with any shared application

Regardless of whether or not a user has the application on his own device, he should still be able to annotate, download, or re-share an application snapshot that has been shared to the screen (see Figure 5). This allows users with any variety of software installed on their devices to participate together. Users should not be limited to using applications designed from the ground up as groupware, and they shouldn't be limited to only working with others who have the same software. Thus, applications need to share themselves in a way that does not require others to have the same application installed on their own device.

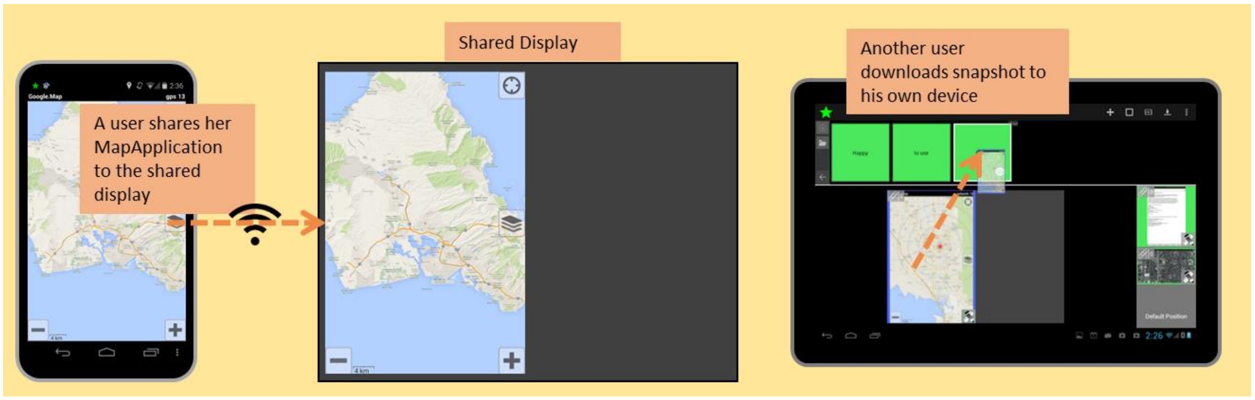

Shared Display
A user shares her MapApplication to the shared display
Another user downloads snapshot to his own device

Figure 5 – A user shares her application to the display. Another user is able to download an application snapshot even if he does not have the same application on his own device.

1.3.2.3 Shared applications can be re-launched to the state in which they were shared

Applications need to be able to share more than just their visual content. Users want to be able to take away the shared application snapshots from their group meetings, but they also want to resume working with those same applications at a later time. When one user shares to the display and another downloads the current screen mirror (see Figure 5 above), the second user should be able to use that snapshot to re-launch the application back to the same state. This could happen in one of two ways. The first case is when the user with the snapshot has the same application installed on his device. In this case, he should be able to re-launch the application and restore its state to where his teammate was working (see Figure 6). Kristen's use of the web browser is a good example of this case. She downloaded a snapshot of Henry's browser while the group was studying together, but when she was reviewing on her own, she was able to re-launch the browser on her own device.

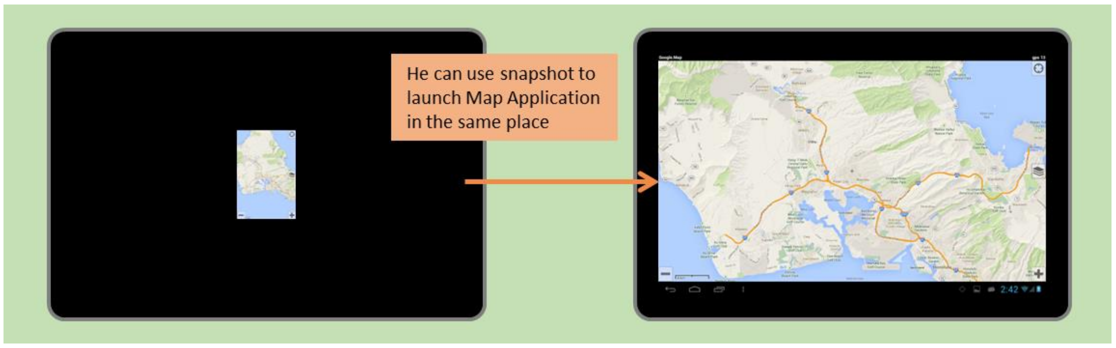

He can use snapshot to launch Map Application in the same place

Figure 6 - If a user has the same application on his own device, he can use the snapshot to launch the application to that state.

The second case that must be accounted for is when a user does not have the same application installed, or the application is dependent on another device's local files. In these cases, he will not be able to re-launch the application directly. He can, however, share it back with the original owner, and she can re-launch it on her device (see Figure 7). Sarah and Peter's use of the 3D modeling program is a good example of this case. Because the model files were exclusively on Peter's device, Sarah could not re-launch the application. She was able to, however, share the snapshot once again, and then Henry could re-launch the modeling program on his device.

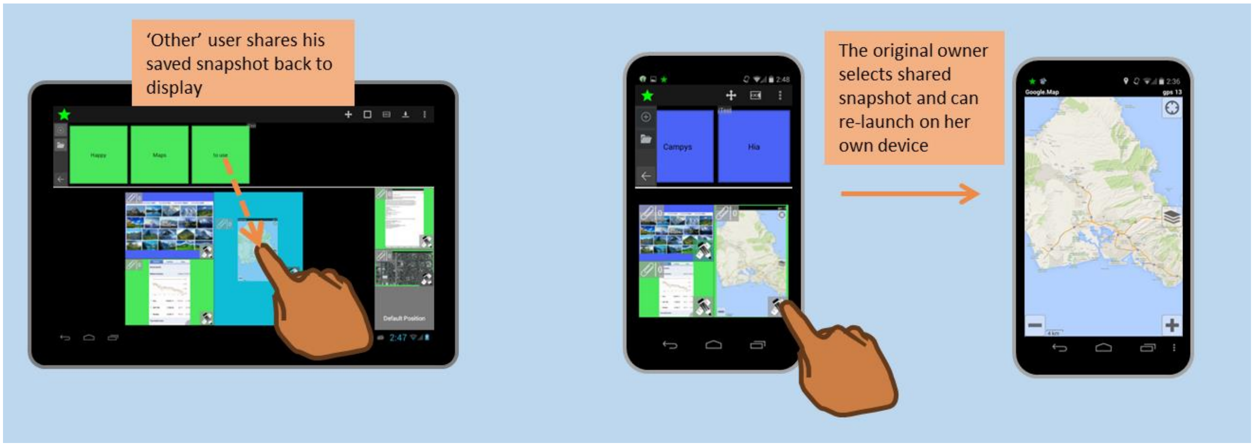

'Other' user shares his saved snapshot back to display
The original owner selects shared snapshot and can re-launch on her own device

Figure 7 - A user can re-share a downloaded application snapshot to the display, and the original owner can use it to re-launch his application on her device.

1.3.2.4 Application participation should require minimal developer effort

There is an inherent tradeoff between integration and flexibility. Specific applications designed from the ground up to have multiple users on multiple devices working with specific data are highly integrated and have powerful collaborative tools. The cost is that they have to be designed from the ground up with collaboration in mind. On the other hand, there are technologies – for example Virtual Network Computing (VNC) [22] servers and clients – which can allow any off-the-shelf application to be shared between users and devices. The cost to this approach is that there is limited collaborative functionality: screen mirroring and input redirection. Our solution requires something towards the latter end of the spectrum. We need to ensure that the cost to gain the functionality to visually mirror and re-launch shared applications remains minimal. A good solution should allow for any application to participate. It should require only minimal developer time and effort in equipping the application with the collaborative capabilities described.

1.3.2.5 Applications should be able to participate in a variety of collaborative spaces

There is no single universal technology for controlling a large shared display. Some displays might be running LearnSpace [29]. Others might have a VNC server installed. Another display could have touch input and custom software driving that. To allow the flexibility to work in different settings with different tools, the shareable applications should be able to participate in a variety of different collaborative spaces.

1.3.2.6 Applications should not be tied to a specific platform

A good solution will allow for applications to participate from more than one device type or platform. For example, it should not be limited to just Android, or just iOS, or only Windows applications, or any other application platform. Rather, the solution should allow for applications from any platform to participate.

Chapter 2 – Prior Work

Using large shared displays to perform collaborative work is not a new area of research. We have organized the related work into the four following sections:

- Application sharing from mobile devices
- Custom collaborative applications
- Shared workspaces
- Mobile devices providing input to shared displays

2.1 Mobile Application Sharing

Many different researchers and commercial products have explored sharing arbitrary mobile applications wirelessly to an external (and possibly shared) display. There are technologies which support simple one-to-one screen mirroring, and also application frameworks which allow multiple users and applications to simultaneously use a shared display.

2.1.1 Screen-mirroring from Mobile Devices

At the most basic level, sharing to a display simply involves a single device taking control of a single display. Since Android 4.2 [30], Android has supported screen mirroring via Miracast[28]. BBQScreen [25] is a third-party Android application that allows users to wirelessly pair their device to a screen. Once paired, the device will screen mirror with the PC and also accept input from the PC. Apple's iOS supports screen sharing to AppleTV devices using AirPlay [2]. All of these systems allow for visual mirroring of any off-the-shelf application. In all of these cases, however, the functionality is limited to a single user connected to the display. This limits the system from meeting most of the collaborative requirements described in Chapter 1.

2.1.2 Multiple Users

The Open Project [18] framework allows applications from mobile devices to be shared to a shared display using "software-based projection." Rather than having a physical projector on the device and projecting its content to a wall or projector screen, Open Project 'virtually projects' the device's content to a shared display by taking screen captures and sending images over the network.

The framework is designed to support multiple devices all interacting with the same shared display. Each device can connect to the display by scanning a QR code from the server running in a browser on the display. After connecting, the user positions his virtual projection by physically moving his device as he would if his device where physically projecting onto the screen. After choosing a position, the user can resume the application and continue working. His application will be updated live on the display, and input can be redirected from the display back to the application running on his device.

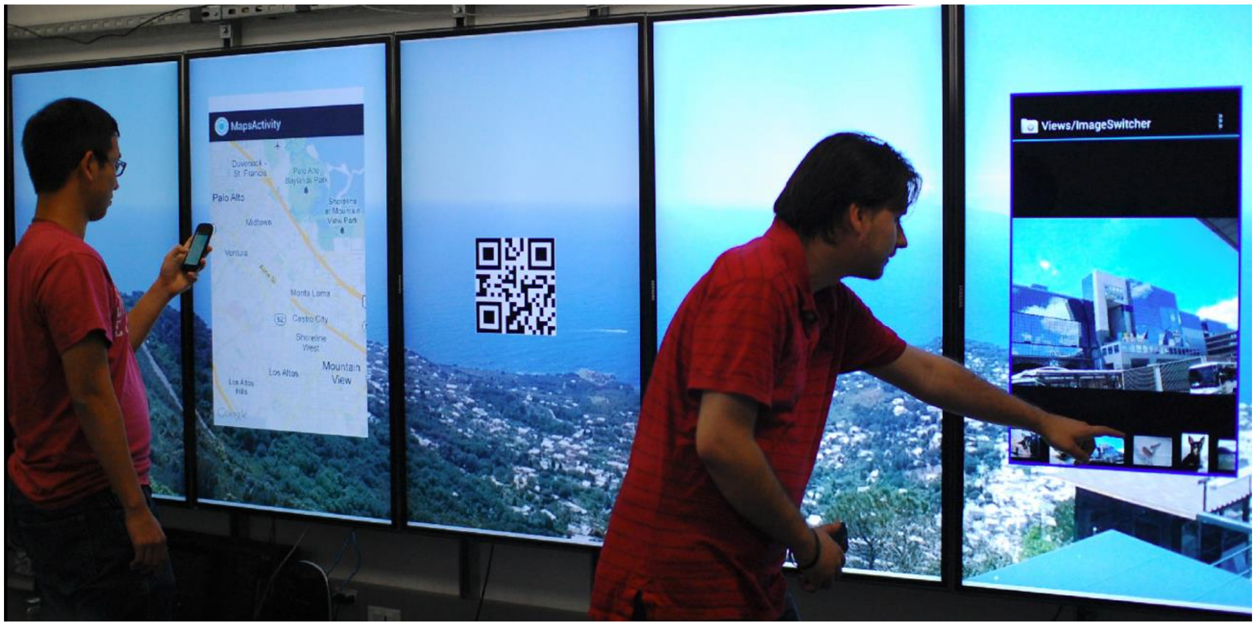


Figure 8 – The Open Project framework allows slightly modified applications to share themselves to a shared display from an Android device.

In order for the Android applications to gain this functionality, the application must include the Open Project library, and then the code must be modified (typically, no more than 30 lines of code need to be added or changed) to incorporate the functionality of the library with the rest of the application.

The XICE Windowing Toolkit [3] was designed to support nomadic computing – where a user's applications and data live on his personal mobile device, but he is able to annex displays and input devices when they are available. The toolkit allows for developers to make applications that can flexibly adapt to different sized displays and different forms of input depending on the current environment. These XICE applications can also be used in collaborative settings as the applications are already designed to connect to displays wirelessly and show content in a fitting way for the large display. XICE supports multiple users connecting to the same display server.

Both XICE and Open Project fall short in a number of similar ways. While they do support multiple users connected to the same display, neither of them support the collaborative interactions of annotating or organizing shared items, nor do they allow users to download snapshots of the applications for personal use. Most importantly, neither allows their applications to be re-launched at a future point to their saved states. Without this functionality, the applications being shared are only useful during the duration of the meeting.

2.2 Custom Applications

Another area of research has been to build custom collaborative applications running between mobile devices and a shared display.

The MobiSurf [24] concept pairs a custom web browser for Android devices with a custom web-browser running on a shared interactive tabletop display. Multiple users can explore privately on their own device. As they find things they want to share, they can tap their device on the tabletop display, which will share the web page to the tabletop's browser. MobiSurf's designers found "that the mobile devices become central interaction devices and that the interactive surface is primarily used to share information for common discussions or later use." They also found that users wanted the ability to "organize information" and "support for more than two concurrent users." Their findings support our solution requirements.

Wallace, Scott, and MacGregor [27] used a custom "sensemaking" application designed to run on either a tabletop display, tablets, or a combination of the two. They wanted to explore how the use of the combination of tablets and the table-top display affected users' performance in these sensemaking tasks. They found that the use of the shared display increased the number of "insights" the group made during the discussion. They also found that "the digital tabletop supported a group's ability to prioritize information, to make comparisons between task data, and to form tableaux which embodied the group's working hypothesis." Their results show some of the benefits of operating in a space with both mobile devices and a shared display.

CREWSpace [16], Branch-explore-merge [17], The ABB Powerwall [8], Greenberg et al.'s "Shared Notes" [10], Liu and Kao's document sharing system [14], and Finke et al.'s split UI systems [9] are all examples of other custom applications designed from the ground up to support specific collaborative tasks. The applications discussed above clearly all fall short with respect to supporting *arbitrary* applications. The conversations they support are limited to their specific functionality.

2.2.1 Dan Schulte's Work

Schulte worked on our same problem of supporting collaborative situations using handheld mobile devices and large shared displays [23]. In contrast to building a single collaborative application, his focus was building tools and a framework for the easy development of a variety of collaborative applications. These applications are built using a shared 'data store' that can be accessed from multiple devices, including the shared display, simultaneously. Thus, all users have equal access to whatever data is being discussed or shared from whatever device they are using. His work also focused on providing interactive techniques to navigate and organize shared items in the context of being on a small handheld-device but having a large view of the data on the shared screens. Figure 9 shows one collaborative application he built using his framework.

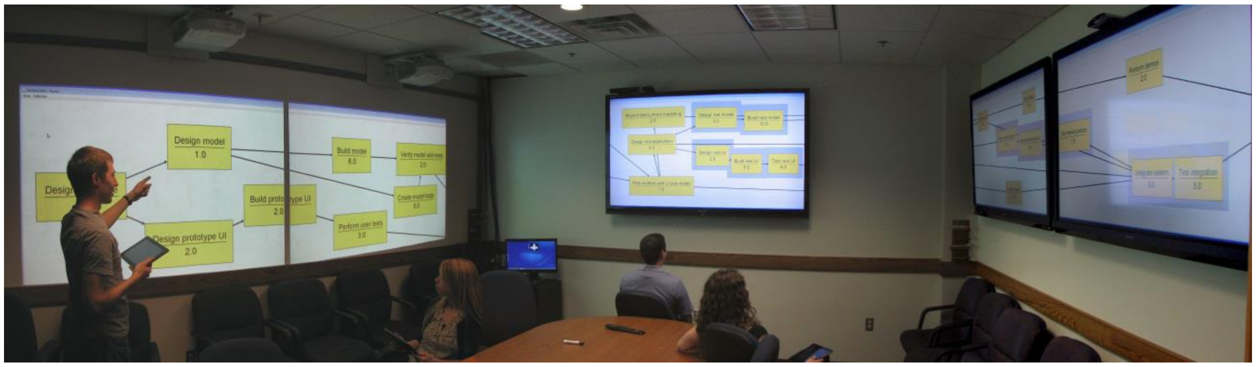


Figure 9 – Multiple users work together in a shared digital space using a collaborative schedule creator built using Schulte's framework [23].

Schulte's work provides the requisite functionality of a collaborative space – in fact, it supports the easy development of many such collaborative spaces – but does not allow for arbitrary applications to participate in the conversation. Conversations are limited to the types of data and items implemented specifically for the collaborative application.

2.3 Shared Workspaces

A popular approach for supporting collaboration across a number of devices is through a shared workspace. Traditionally, a shared workspace is made up of multiple computers networked together with software to support device, data, and/or application sharing.

The Impromptu system [5] allows users to share any off-the-shelf application to any other display within the multi display environment. It also allows for shared windows to be placed on a shared display and for users' input to be redirected to the shared display. As such, it allows for applications to be shared on the workspace's large shared displays where any number of collaborators can interact with them. While Impromptu does support many of the collaborative techniques that we are looking for, it was designed for users sitting at desks with mouse, keyboard, and a desktop operating system.

The Lacome [15] system uses VNC to allow multiple users on multiple devices to share their desktop or applications simultaneously on a shared group display. Lacome clients can connect to the server to share applications, control shared applications, manipulate the "windows" of the shared applications, and/or annotate over the shared windows. The functionality of their system is actually quite similar to our proposed solution. Because any VNC server can connect and share its screen on the Lacome server, it is even conceivable that mobile devices could share their applications on the wall display. Currently, however, popular mobile operating systems require that a device be "rooted" or "jailbroken" before allowing the installation of a VNC server. The primary shortcoming, however, is that there is no support for taking away shared items from the meeting or resuming shared applications at a later time.

LearnSpace [29] provides similar functionality as Lacome for sharing a desktop or application to a shared display, but does not support input redirection as it does not use VNC. It does support, however, a few additional collaborative capabilities such as sharing images and videos, and taking surveys among the participants.

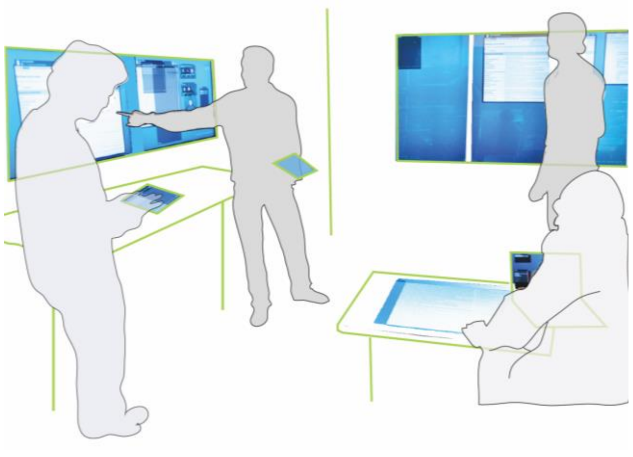


Figure 10 – The Reticular Spaces “Smart Space” [4]

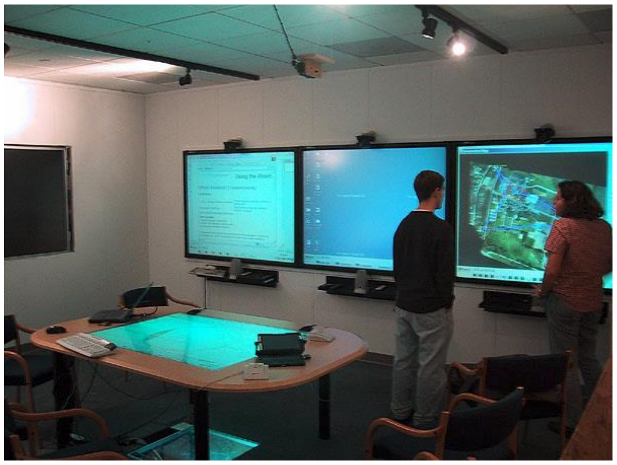


Figure 11 – The iRoom shared workspace [12]

While LearnSpace, Lacome, and Impromptu are designed for general application sharing and have minimal tools to facilitate such sharing, the iRoom [12], UbiCollab [6], and ReticularSpaces [4] frameworks are examples of more specialized shared workspaces. These specialized workspaces have much deeper frameworks that allow more rich device control and environment awareness at the expense of generality. So, while these systems do integrate mobile devices, there is no support for arbitrary application sharing from those devices.

2.4 Mobile Devices as Input

When working with large shared displays, users are often not in situations conducive to using a keyboard and mouse to work with their applications. Thus, a popular area of research regarding mobile devices has been how mobile devices such as cell phones can be used as input devices to a shared display.

One method of such interaction that has been explored in a variety of ways is using camera equipped mobile phones to perform interactions on the large display. Vartiainen et al. [26] developed interactions where they distribute “glyphs” across the entire screen of the shared display (see Figure 12**)** and then use computer vision with the phone’s camera to determine the location of the phone. To illustrate these interactions they built an interactive whiteboard application. Users were able to upload text or images from the phone to the display, drag and drop items, and draw on the "whiteboard" all at the location the phone was pointing. Another approach to use a camera equipped device was explored by Jeon et al.[11]. This approach allowed the phone to control a cursor on the screen solely on the movement of the camera (without the use of "glyphs"). Other techniques they explored used markers (similar to glyphs) attached to objects on the display in order to allow users to drag, resize, and rotate those objects.

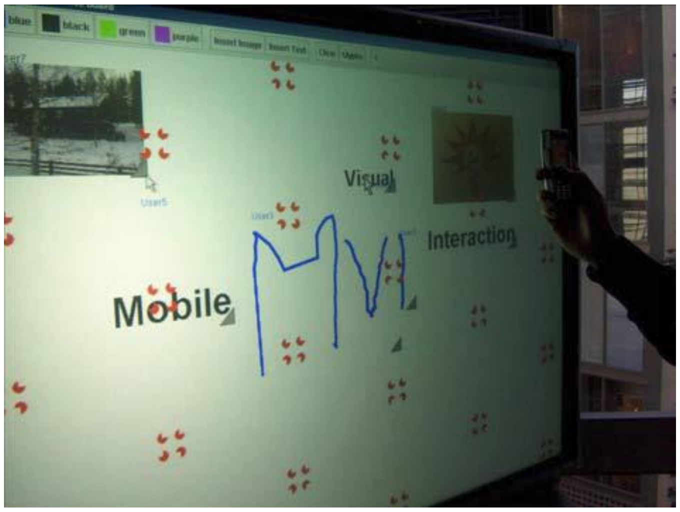
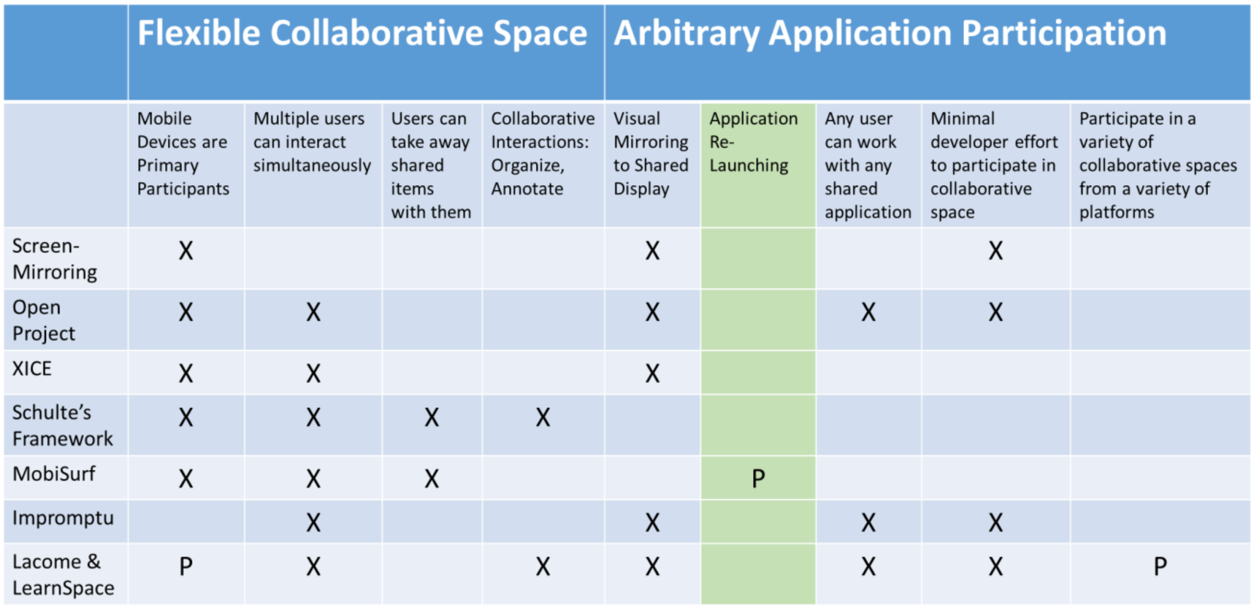


Figure 12 - Using glyphs on a shared display [26]

Another example of research in this area is the work of Paek et al. [21]. These interactions are limited by high-latency, low-bandwidth communications from the phone such as text messages. For example, a web-browser was built that allows mobile users to control the browser by sending text messages to the public display.

This and other research of its kind provide examples of the benefits of using mobile devices when interacting with large shared displays. We, however, are not interested specifically how a phone or tablet can provide input to software running on the display, but how the shared display can facilitate effective collaboration for those who use these handheld devices as their primary computing device.

2.5 Summary



	Flexible Collaborative Space				Arbitrary Application Participation				
	Mobile Devices are Primary Participants	Multiple users can interact simultaneously	Users can take away shared items with them	Collaborative Interactions: Organize, Annotate	Visual Mirroring to Shared Display	Application Re-Launching	Any user can work with any shared application	Minimal developer effort to participate in collaborative space	Participate in a variety of collaborative spaces from a variety of platforms
Screen-Mirroring	X				X			X	
Open Project	X	X			X		X	X	
XICE	X	X			X				
Schulte's Framework	X	X	X	X					
MobiSurf	X	X	X			P			
Impromptu		X			X		X	X	
Lacome & LearnSpace	P	X		X	X		X	X	P

Figure 13 – A summary of the most similar related work – showing where current systems are lacking. Xs signify that the system supports the given feature; P's signify that they potentially or partially support a given feature.

Considering the related work altogether (in Figure 13 above), it is easy to see that no existing systems fully support the collaborative situations described in Chapter 1. It is also clear that the major missing component is the functionality to share arbitrary applications, take snapshots of them away from the meeting with you, and re-launch the application to its captured state. This functionality is vital in realizing the situations described in Chapter 1, but it simply has not been explored. MobiSurf [24] seems to support this in a small way as it shares URLs back and forth between devices and the tabletop display. This is completely insufficient, however, as MobiSurf only supports a single application and does not have a mechanism for saving application snapshots. It is at least a step in the right direction.

Chapter 3 – System Architecture

In order to meet the system requirements identified in Chapter 1, we have designed and built a framework that supports the described application sharing as well as a collaborative space that provides the necessary interactions. Together, our system allows for arbitrary applications to be easily shared, used for discussion, downloaded, and later resumed to their shared state. While our solution is extensible to multiple platforms (which we discuss in Chapter 7), we have currently only implemented it for Android and LearnSpace working in tandem. As such, our discussion below will be described from that perspective.

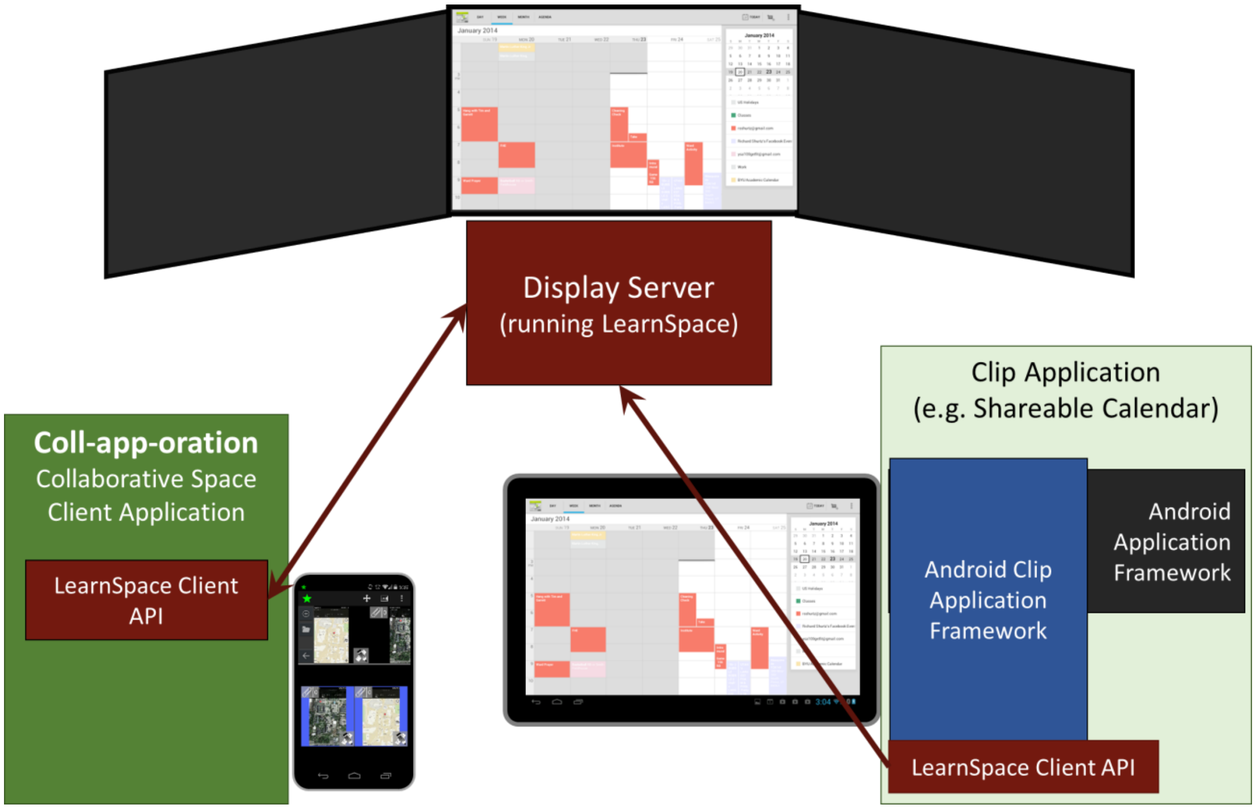


Figure 14 – The overall design of our solution. LearnSpace technology is used as the communication backbone of our solution.

There are three main components to our solution (shown together in Figure 14). Applications running on users' mobile devices share themselves to the display using our Android

Clip Application Framework. These ‘clip applications’ push application snapshots to the shared display. The shared display is driven by the LearnSpace server. Finally, the collaborative space client application allows users to organize, download, annotate, re-share, and re-launch these shared applications. The inter-device communication for our system is through the LearnSpace Client API which communicates with the LearnSpace server.

3.1 Application Sharing

At the heart of our solution are shareable applications. We need applications that can share themselves visually on the shared display with a live, continuously updating view of the application. But, just as importantly, we need applications to share themselves in a way that they can be re-launched and resumed. Our solution is to do all application sharing via ‘clips.’ A clip is a ‘snaptshot’ of an application. It is an image of the application with some additional attached ‘re-launch data’. We will discuss clips and our sharable ‘clip applications’ in detail.

3.1.1 Clips

An application ‘clip’ is an image screenshot of an application with attached re-launch data. The data contains the information to re-launch and resume the application in the same state of the clip. Clips have the benefit of being immediately useful as they share visual content which can be viewed, positioned, organized, annotated, etc. simply as an image. Images are a universal media for sharing useful information [19]. The physical media typically used in small-group meetings (printed agendas, pictures, figures, white board, etc.) can all be represented using images. Making our basic unit of collaboration as simple as an image gives us incredible flexibility and generality.

But, clips are more than just images. They also represent an application and that application's state. They are generated by applications and can be transformed back to applications by re-launching. So, sharing an application by sharing clips is inherently more useful than simply sharing pictures. A user can still explore, scroll, zoom, etc. in his application while continuously sharing it on the shared display with the group. Then, users have the ability to take the clips and use them to resume the application. The use of clips allows the flexibility and generality of shared images and also the specificity of using particular applications.

3.1.2 Clip Applications

Of course, clips only exist if there are applications which are designed to be shared in that manner. Thus, a vital component of our solution is the design of 'clip applications.' A clip application can share itself by producing clips and sending clips to the shared display. As the user interacts with the program, a shared clip application continuously sends new clips to the display to update both the visual image of the program and its re-launch data. Also, if it is provided with application re-launch data, a clip application can restore its state to match the state of the clip. Other than their ability to connect and share to LearnSpace display servers, clip applications behave just as normal applications.

Consider the web browser from our student study session example. The web browser is a clip application. As Henry shared it from his device, it sent clips to the display server so that his classmates could see what he was doing in the browser. When Kristen downloaded a snapshot of the browser, she was downloading the clip that was shared on the display. The clip was able to re-launch the web browser on her device because it contained the application re-launch data.

Rather than building a suite of clip applications, we designed and built a framework which allows for the simple creation of clip applications. Our framework is primarily an invisible extension to the Android Application Framework. The goal of the framework is to allow the developers to be able to design and build applications as they normally would and then make only minor modifications to be able to save and restore from the re-launch data.

Figure 15 shows how clip applications are built on top of the Android framework and our clip application framework. It also shows that the communication between the clip application and the display server take place through the framework using the LearnSpace Client API.

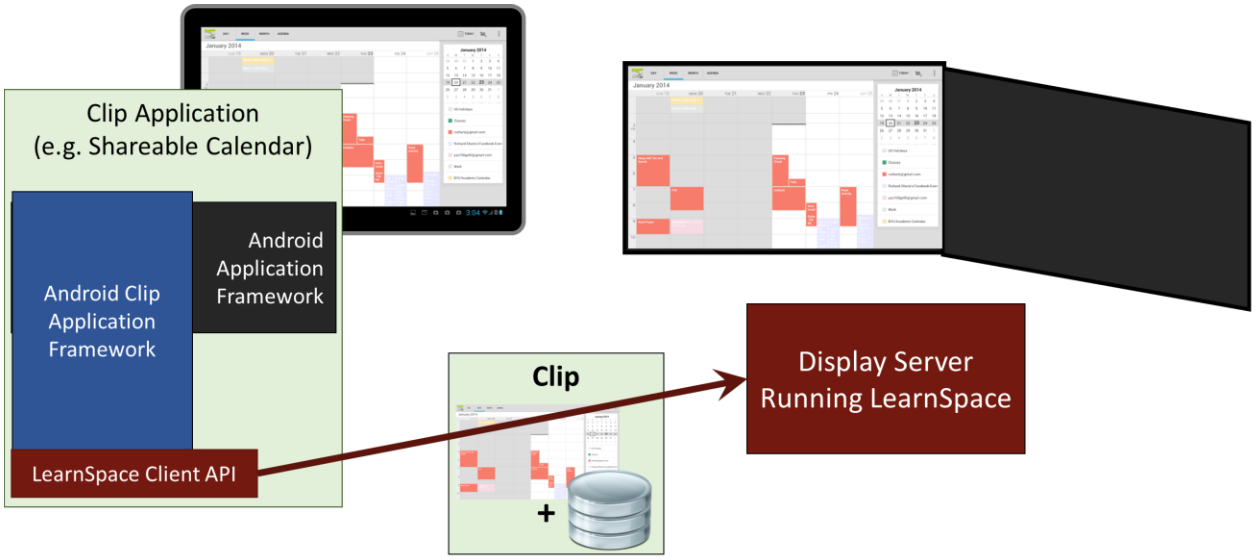

Clip Application
(e.g. Shareable Calendar)
Android Clip Application Framework
Android Application Framework
LearnSpace Client API
Clip
+
Display Server Running LearnSpace

Figure 15 – The underlying design for supporting new clip applications. Clip applications use the Android Clip Application Framework to share clips to the LearnSpace display sever.

3.2 LearnSpace Display Server

A central part of our system is the display server which hosts and displays the users' shared content on the large display. Rather than implement our own display server, we elected to use the LearnSpace server [29] as it supports the necessary functions to implement our collaborative space. With this, we did not need to worry about implementing any protocols or deal with the lower lever aspects of networking. Rather, we used the LearnSpace Client API and the server to build our solution using higher-level concepts. Figure 14 shows how the different components all communicate using the LearnSpace Server and Client API.

Listed below are features of the LearnSpace server that we used in our solution. This is, of course, not an exhaustive list of the features that the server provides. It does, however, give a general idea of the foundational capabilities that we used in our implementation.

3.2.1 Simple Windowing Framework

The LearnSpace server renders to the display according to a simple windowing framework. Clients can create windows – called sheets – on the display and then choose where the sheet is positioned. Most of the time, a sheet simply shows an image provided by a client, but can also display other things such as polylines, or videos. Sheets are identified by a unique ID, and can have arbitrary string attributes added to them to store references, descriptions, etc. In our use, we create a new sheet for any application/item shared to the display.

3.2.2 Data Assets

The server hosts files or data as assets. Assets can be any sequence of bytes, but there are also specific asset types that can be used for specific purposes. For example, to display an image on a sheet, a client needs to create an image-asset. There is also a live-image-asset which can be used to update different parts of the image. This helps minimize network communication when sharing an application live to the display. Often times, only small parts of the application's view will change, so it is helpful to need to send only these parts. We also use assets for storing annotations and application re-launch data on the server.

3.2.3 Client Notifications

The server also allows clients to register listeners for different events that might take place on the server. It will then notify clients when those events take place. For example, our client application needs to be able to show the user what sheets are where on the display. So, it registers a listener so it is notified whenever a sheet is created, moved, or destroyed.

3.2.4 Clients can moderate

One other important feature of the server is that it allows clients to 'moderate' the display. The control of what-sheets-can-go-where and who-can-delete-what is determined by this 'moderator.' When a client is registered as the moderator, then it can approve, modify, or decline requests to create, re-position, delete sheets; create, update, or delete assets; etc. We use this functionality to enforce different organization techniques and ensure that shared applications do not overlap or interfere with one another.

3.3 Coll-app-oration – The Collaborative Space Client

The final piece of our solution is our Android application that allows users to interact in the collaborative space. The clip applications just send and update clips to the shared display – allowing team members to see what they are doing. In order to implement the rest of the requisite features of our collaborative space, however, we designed and built an Android application we named Coll-app-oration. This application connects to LearnSpace using the same client API, and provides the functionality described in the following sub-sections.

3.3.1 Downloading

From within Coll-app-oration, users can download any shared clip to their own device. The downloaded clips are organized into collections (essentially directories) so that the user can keep track of downloaded clips according to her needs. Figure 16 below shows how this is done. The bottom part of the screen shows the user's access to the shared display, and the top part shows her local collections. To download, she simply drags the application from the shared display to the collection where she wants to save it.

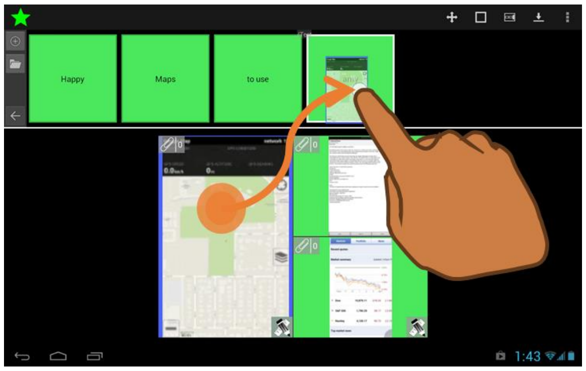


Figure 16 – A screenshot of a user using Coll-app-oration to download a shared clip.

3.3.2 Sharing

After downloading or collecting different clips on their own devices, users are free to share any clip or collection of clips to the display. Once again, this is done with a simple drag-and-drop interaction (see the screenshots in Figure 17 below). Clara, in our automotive design team example, could use this functionality to share all of her collected ideas with a single interaction.

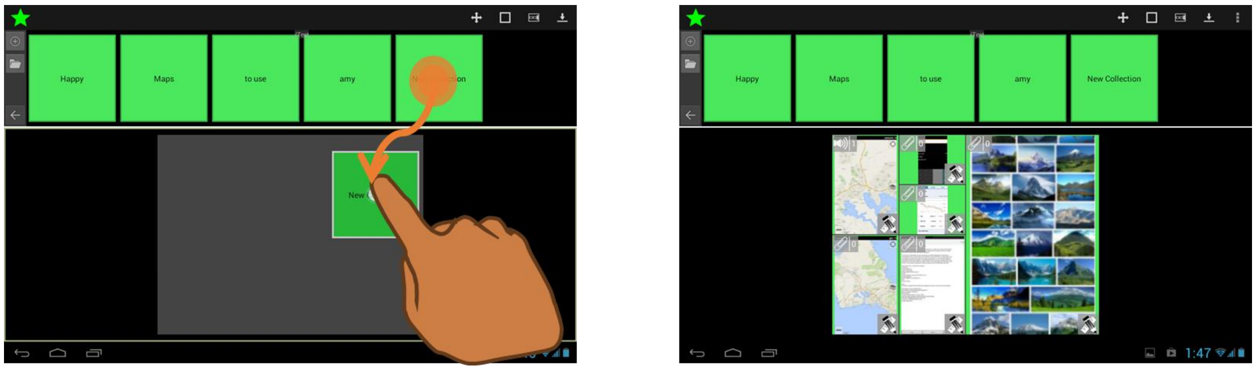


Figure 17 – *Left:* A user shares a collection of saved clips to the shared display. *Right:* The saved clips from the collection are now all shared to the display.

It is possible that two users could download the same clip, and then both attempt to re-share that clip to the display. Consider, for example, the student study session. Henry, Frank, and

Kristen decided to study again before the test. As they review a topic, however, both Kristen and Frank re-share the same clips from Henry's PDF application that he shared during their first meeting. To prevent duplicates in such cases, Coll-app-oration gives each shared clip an ID that is unique even across different devices and meeting sessions. Coll-app-oration checks to see if any clips on the display have a matching ID before sharing. It also checks IDs before downloading to prevent having duplicate clips on the device.

3.3.3 Annotations

We implemented two useful types of annotations for our system that can be created from within Coll-app-oration. The first is the ink annotation. An ink annotation can be made over any shared clip on the shared display or over any locally saved clip. If made on a shared clip, the ink annotations are visible directly to all users on the shared display, and multiple users can all participate in annotating at once. For example, in the student study session, all three students were able to draw highlights over the study guide shared from Henry's email. They each drew using their individual devices, but all were working on a single ink annotation on the display.

The second annotation type is an audio annotation. Users can leave an audio note with any clip. Either annotation can be saved for later use on the shared display or locally on a user's device. In our example where Frank recorded an audio note of Henry explaining a topic, it is only really useful if Frank can take the audio annotation home with him. We show screenshots of these annotations below in Figure 18.

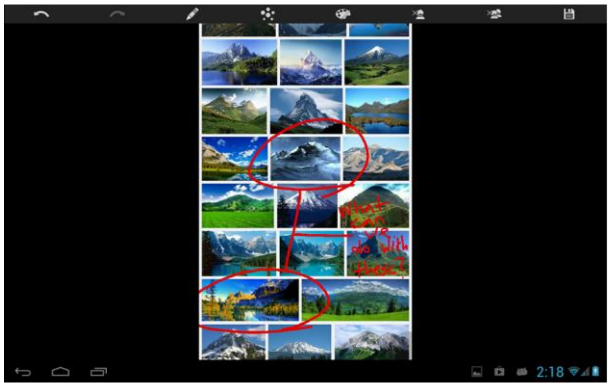
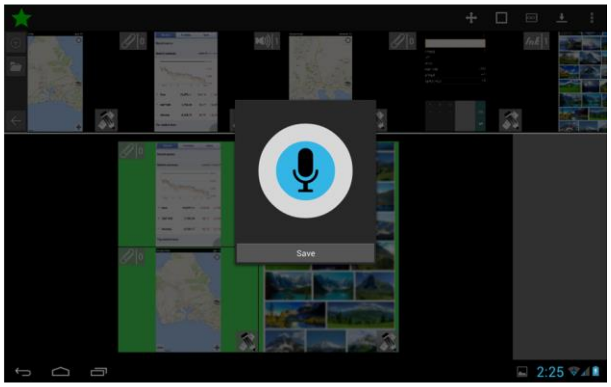



Figure 18 – *Left:* A user draws over a clip to highlight interesting points from within Coll-app-oration. *Right:* A user records audio annotations on a clip using the client view of the shared display.

If a clip saved on the local device has any annotations, those are automatically uploaded to the display when the clip is shared. Likewise, if a user downloads a clip having annotations from the shared display, the annotations are automatically downloaded along with it. For example, consider Clara sharing her ideas with the team. Teammates made ink and/or audio annotations on her shared clips. For her to take those ideas away with her from the meeting, she can simply re-download the clips. Coll-app-oration will then download all the annotations, and she will have them with the clips on her own device to continue working after the meeting.

3.3.4 Organization

Coll-app-oration has two different types of organization modes. These are enforced by setting one client to be the LearnSpace moderator so that all sheet positioning requests can be decided by a single entity.

The default mode is called Grid-Snap. When a clip is shared to the display, the moderator automatically positions it to the largest available space. Or, if there is no available space, the moderator resizes the largest sheet to half its size, and then places the new sheet in the now-empty position. (This mode can be seen on the right side of Figure 17). If the users need to keep one clip in a specific location at a specific size, they can choose to "anchor" the sheet. Once anchored, the moderator will not resize or reposition the sheet. This organization mode allows many items to be shared with minimal positioning effort required from the users – but allows users to position items explicitly when required.

The second organization mode is a simple two-by-two organization mode often used in design and brainstorming work [13]. The shared clips are resized by the moderator, and then users can position them in a two-by-two grid according to how well they fit whatever criteria the group is discussing. In this mode, the moderator creates a sheet for the axes for the grid to go behind the sheets of the clips being organized. A screenshot of Coll-app-oration in this organization mode is shown in Figure 19. The shared display matches what is seen in this screenshot.

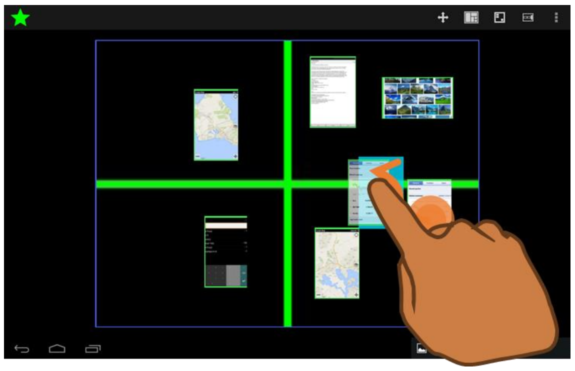


Figure 19 – A user organizes the shared clips using the Two-by-Two organization mode.

3.3.5 Re-Launching

Finally, Coll-app-oration contains the functionality to re-launch an application from a clip. Each clip (either on the shared display or in a local collection) has a re-launch button in its lower right corner (see Figure 20). If the clip cannot be re-launched on the particular device (for example, if the clip's application is not installed on that device), then this button is disabled.

Once the user clicks this button to re-launch the application, Coll-app-oration creates a call to the system to start that clip's application. If the clip is on the shared display, then the application will begin automatically sending clip updates back to the same LearnSpace sheet immediately after it is re-launched and resumed. If the clip is re-launched from a local collection, the application will only be shared to the display if the user elects to begin sharing.

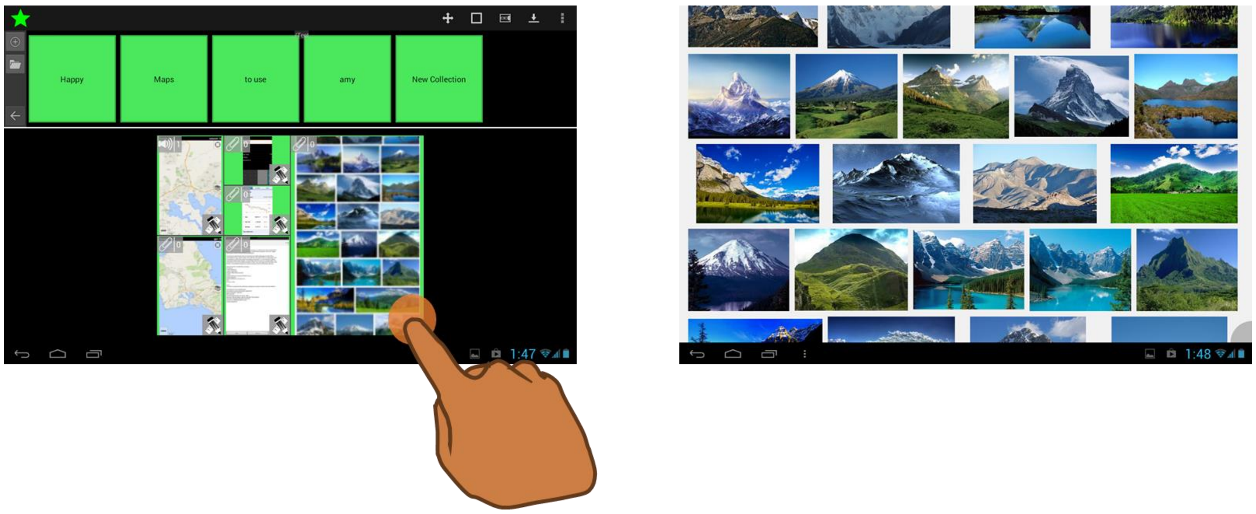


Figure 20 – *Left:* A user chooses to re-launch a clip on the shared display by clicking the clip's re-launch button. *Right:* The application is now running on his device in the same place as it was in the saved clip.

This interaction is a key component in realizing the examples given in Chapter 1. For example, when Kristen (from the student study session) was studying on her own, she was able to simply open Coll-app-oration, find the browser clip she had downloaded, and click the re-launch button. The web browser re-launched, navigated to the Wikipedia page, and restored the scroll-position. Similarly, when Sarah (from the automotive interior design refinement example) wanted Peter to update his work to match hers, she was able to use Coll-app-oration to share her downloaded clip back to the display. Peter was then was able to use Coll-app-oration on his device to re-launch and resume his 3D modeling program. Upon re-launching, because the clip was already on the shared display, his 3D modeling program resumed sharing to the display so Sarah could see his work.

In both cases, this functionality enables a new flexible way to share applications. Users are able to take each other's applications (as clips) away from the meeting, use them for personal work, and resume the application when needed.

3.4 Information Flow Summary

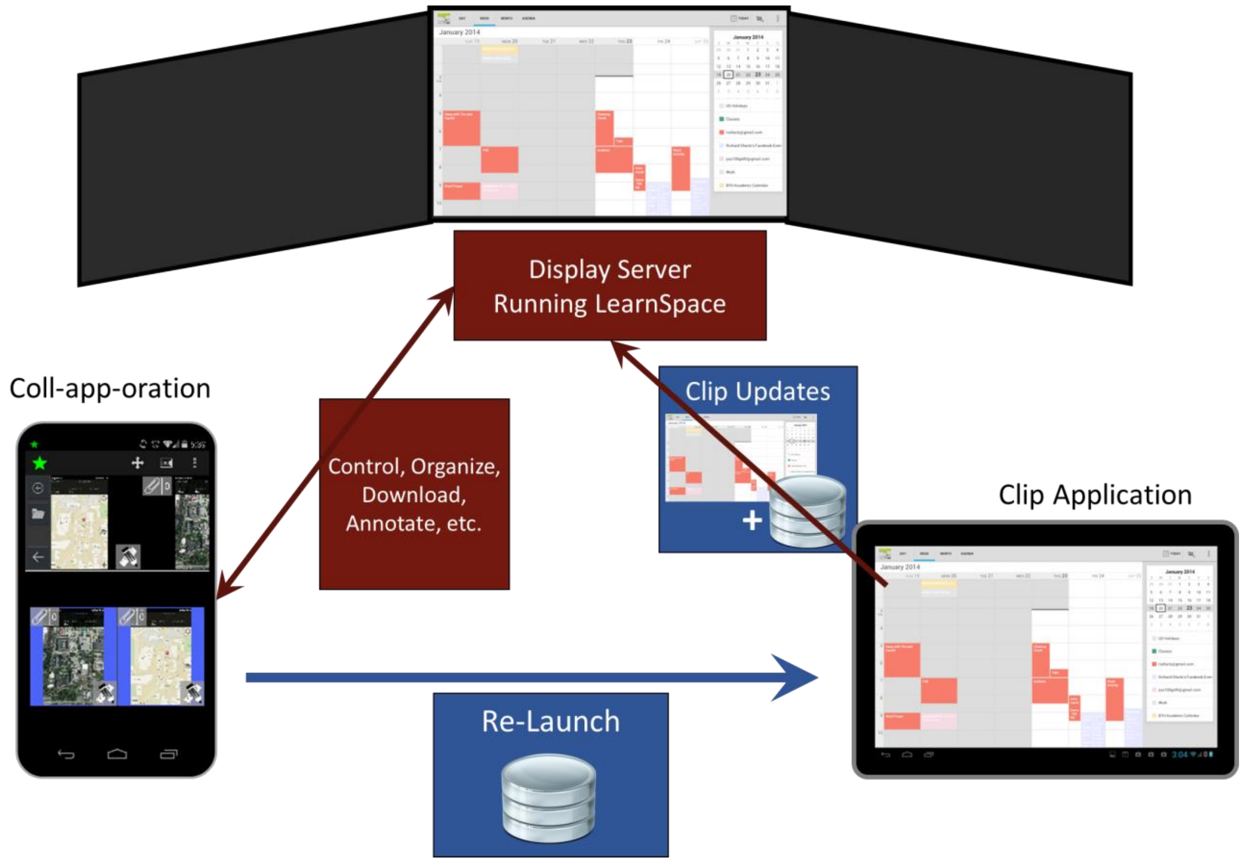


Figure 21 – A summary of how information is flowing throughout the different parts of our solution.

Figure 21 summarizes the data/information flow of clips through the system. A clip application sends clip updates in real-time to the LearnSpace display server. Coll-app-oration downloads, shares, annotates, etc. clips on the shared display. Users can select to re-launch an application from within Coll-app-oration at which point the clip's re-launch data is used to restore the application's state. When one or more clip applications are running on the device alongside Coll-app-oration, the user is able to switch among these applications using Android's built in multi-tasking system.

3.5 Key Challenges

3.5.1 Application re-launching

While sharing pixels from an application to a shared display is no new feat, sharing general applications with a mechanism to restore them at a later point is an unexplored area of research. For example, consider a user sharing an email to a wall display from his email application. Another user downloads the email to look at later. After the meeting, he writes down a few ink annotations and decides he wants to talk about it later with his teammate. When they meet, he shares it back to the display. The original owner now wants to re-launch his email application to that specific email message so he can write a reply, but he does not want to look through his inbox to find the specific email.

We need to be able to use the shared clip to re-launch the application to its saved state. But, from the clip, how do we know which application to launch – or even if we have that application installed or that it can be re-launched on the particular device? And once we have launched it, how does the application restore its state? How can we do this without saving the contents of the email to the clip, or can we? Finally, can we ensure that our solution will be robust enough to support a wide variety of applications? In order to solve these problems, we designed application clips to carry with them their application re-launch data. This data consists of a launch-action string, as well as an application-determined sequence of bytes that we call 'application-DNA.' We discuss these in detail in Chapter 4.

3.5.2 Easy framework for developers

Our solution is only as useful as the clip applications built to take part in the system. One challenge we had to address, therefore, is how to allow arbitrary applications to participate. For example, there already exist widely used note-taking applications that collaborators might want to share to the display at a meeting. One of the most popular note-taking applications is Evernote [7]. If the Evernote designers wanted to add the functionality to participate in our system, how much would they have to change? And if another application, for example, Adobe Reader [1], also wanted to participate in our system, how much would the developers need to change? How many of these changes would be common between the two?

Our challenge was to build a framework that takes care of as much of the clip sharing requirements as possible. This framework needs to work with any arbitrary application and yet allow the developer the flexibility to specify how it is integrated. The framework also needs to comply with Android's relatively complicated Activity lifecycle [31]. Our solution was to develop the Android Clip Application Framework. It makes changing existing applications into collaborative clip applications simple by only requiring a minimal number of changes to the existing code. We discuss the Android Clip Application Framework in detail in Chapter 5.

3.5.3 Compatible with other collaborative spaces

The final key challenge that we had to address was to ensure that our clip applications are not tied specifically to LearnSpace – or any other single collaborative space technology. We chose to build our collaborative space on top of the LearnSpace technology as an example for how to support the collaboration requirements that small team-colocated meetings demand. We know, however, that collaboration takes place in many different settings, and the demands of the supporting collaborative space might be different in each of those settings. For example, in

Schulte's work [23], he explored a couple different collaborative applications he implemented using his framework. The primary shared items in his applications were text notes, images, or ink drawings. What if, however, he wanted to allow any arbitrary application to share itself as one of the items? Can we use our clip applications to be participants in his collaborative space?

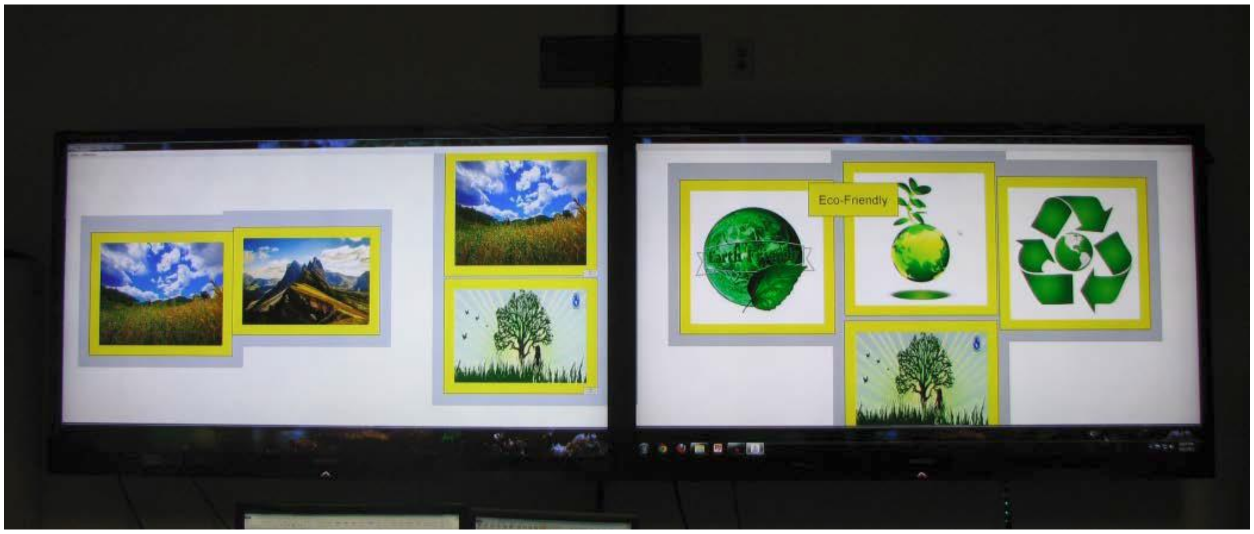


Figure 22 – Schulte's framework and applications support rich organizational and collaborative interactions, but are limited in the data types they support [23]. Pictured above is the grouping of different picture items. Can we replace those images with clips?

We believe that clips and clip applications will be a useful paradigm across many settings and tasks. We do not want to limit clip applications to our LearnSpace implementation or to our Android-specific clip application framework. To separate the specifics of our implementation from the clip paradigm, we defined the Clip Space API. We discuss the details of this API in Chapter 6. We also discuss how to extend our clip application framework to new platforms in Chapter 7.

Chapter 4 – Application Re-Launching

At the core of our research is the problem of “how do we share an application so that it can be downloaded and then re-launched to that state at a later time?” While the prior work reveals that others (such as Schulte) have successfully built both collaborative spaces and mechanisms for sharing applications visually (for example VNC), none have attempted to share applications with a mechanism for re-sharing.

In order to realize such a system, we need a consistent robust method for sharing applications visually in a way that other users can download them. We have already explained how we have met this need by using clips as our method of application sharing. We share – in real-time – visual snapshots of the application along with accompanying re-launch data.

This application re-launch data must provide a means to determine whether or not the clip can be used to re-launch on a particular device and also a way to actually start the application. It must also provide a mechanism to restore the application’s state once it has been re-launched. In meeting these needs, we must ensure that our solution is simple and lightweight. It must not require extensive work to save and restore an application’s state, and the size of the application re-launch data must be kept relatively small. Finally, a good solution will be flexible. It must work for a wide variety of applications to ensure that many existing and future applications can easily participate in our collaborative system.

The re-launch data is split into two parts. The first is what we call a ‘launch-action string.’ It is a description of the platform, the application, and the device so that the system can determine whether or not the application can be launched on a particular device. If it can be, the system uses this information to make the system call to start the application. The second part is what we call ‘application-DNA.’ This is application-specific data used to save the state of the application. It can be any sequence of bytes. We describe these both in detail below.

4.1 Launch-Action String

The launch-action string is a description of the shared application. An application designed to re-launch the clip (in our case Coll-app-oration) can use this description for two purposes. First, it can determine whether the clip can be re-launched on the current device or not. Second, it can use the description to piece together a call to the system to actually launch the application.

We will explain the specifics of how we implemented this on Android, but we believe that we could use similar launch-action strings for other platforms as well.

Our Android launch-action string consists of four parts (see the Format in row 1 of Figure 23 below). The first part is a label to say that the launch action string is specifically describing an Android application. The second part is the application package name (on Android, all applications are installed under a specific unique package). The third part is the ‘activity’ class name. (On Android, activities are the primary UI components used to create applications – for example, an email application might consist of three sub-classes of the Activity class: InboxActivity, MessageActivity, and ComposeMessageActivity.) The final piece is one of two things. If the application is only to be re-launched on the original device (if, for example, it is dependent on data stored locally on the device), then it is the device’s unique ID. If the application is independent of the device or local data, then this fourth part is a pre-specified string to indicate it can be launched on any device. We provide a couple real examples of launch-action strings below in Figure 23.

Format	**“AndroidLaunchAction::<Package Name>::<Activity Class Name>::<Device ID>”**
1. Email	“AndroidLaunchAction::com.android.email::com.android.email.activity.MessageList::AC15986123496”
2. RMaps	“AndroidLaunchAction::com.robert.maps::com.robert.maps.applib.MainActivity::$&$&”

Figure 23 – The format of launch-action and the two real examples. Note than the device/user-dependent Email application includes a device ID, while the device independent RMaps application contains the system-specified string “$&$&”.

In practice, Coll-app-oration first checks launch-action strings to determine whether it starts with “AndroidLaunchAction”. If it does, it parses the rest of the string and queries the system to see if there is a matching install for the particular package and activity class names. It also checks to see that either the application is device-independent or that the current device ID matches the device ID from the launch-action string. If these line up, Coll-app-oration enables an option for the user to interactively re-launch the application. When the user selects the option, Coll-app-oration creates a request to the system to launch the particular activity within the particular package.

4.2 Application-DNA

The launch-action string solves the problem of restarting the clip application, but it does not take care of restoring the application’s state. To do that, we also include additional data called application-DNA as the final field in the application re-launch data. Application-DNA is a sequence of bytes completely determined by the specific clip application. The application saves its state to a byte array when it makes a clip. When the application is re-launched, this same byte array is given back to the clip application so that it can extract the saved information and restore its state.

In practice, however, using application-DNA is much more developer-friendly than working with byte arrays. We provide a map data structure that allows basic data types to be stored with string key references (discussed later under the ‘AppDnaMap’ heading in Chapter 5).

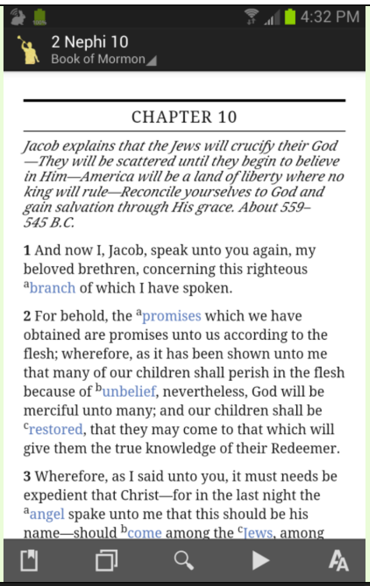
Thus, application-DNA can be thought of as saving a few pieces of simple data. For example, consider a browser application. The most important piece of state information to capture would be the URL of the web page that is currently open. If the developer wishes to include more state, such as the scroll position on the page, he can add that as well. So, for the browser, the application-DNA consists of a string value for the URL and a floating point value of the browser's scroll percentage down the page.

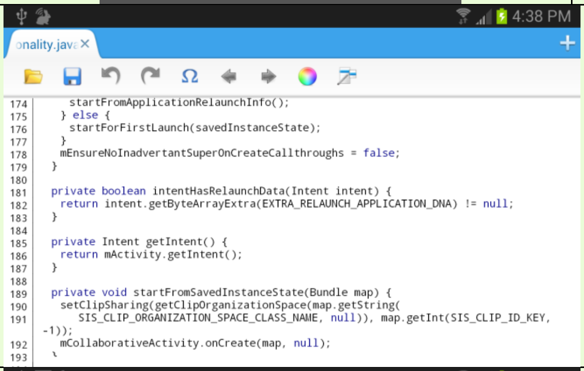
The simple example is actually a very good introduction to the concept of application-DNA. The DNA is very simple, but it captures most of the state of the application. There will be, of course, other aspects of the browser state that are not captured – such as any cookies relating to the current web page. However, the goal with application-DNA is not to recover everything, but to recover as much as possible without expending too much effort – we expect that the first 10% of the work gets 90% of the results.

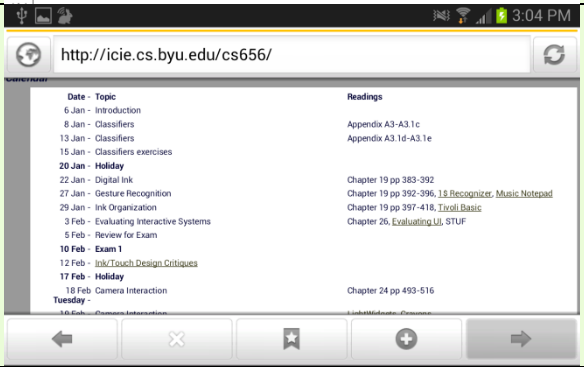
4.2.1 Examples

As part of our work, we analyzed various applications in order to determine what their application-DNA needs would be. Our analysis provides a good summary of what application-DNA would normally consist of. It also shows that application-DNA is flexible enough to support a wide variety of applications. The table (Figure 24) shows a majority of the applications we surveyed. Those with a green background are applications for which we had access to the source code and actually converted to fit our framework (see Chapter 5). For those with the red background, we did not have access to the source code, so the application-DNA is our best guess at what the application would require.

Application ***Activity***	**Screenshot (for reference)**	**Application-DNA**

Gospel Library [32] *Content Activity*		`String verseURI`
		`boolean fullscreen`
		`boolean showRelatedContent`
		`boolean showAnnotations`

920 Text Editor [33] *Editor Activity*		`String filePath`
		`int startSelectionIndex`
		`int endSelectionIndex`
		`char[] charsBefore`
		`char[] charsAfter`

Zirco Browser [34]		`String currentUrl`
		`float scrollPercentage`

4:32 PM
2 Nephi 10
Book of Mormon

CHAPTER 10

Jacob explains that the Jews will crucify their God—They will be scattered until they begin to believe in Him—America will be a land of liberty where no king will rule—Reconcile yourselves to God and gain salvation through His grace. About 559–545 B.C.

1 And now I, Jacob, speak unto you again, my beloved brethren, concerning this righteous ᵃbranch of which I have spoken.

2 For behold, the ᵃpromises which we have obtained are promises unto us according to the flesh; wherefore, as it has been shown unto me that many of our children shall perish in the flesh because of ᵇunbelief, nevertheless, God will be merciful unto many; and our children shall be ᶜrestored, that they may come to that which will give them the true knowledge of their Redeemer.

3 Wherefore, as I said unto you, it must needs be expedient that Christ—for in the last night the ᵃangel spake unto me that this should be his name—should ᵇcome among the ᶜJews, among

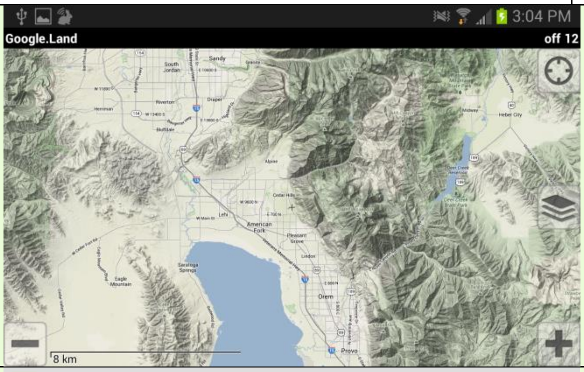
4:38 PM
onality.java ×

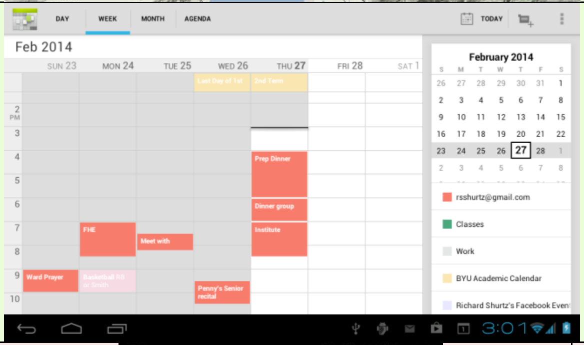
```
      startFromApplicationRelaunchInfo();
    } else {
      startForFirstLaunch(savedInstanceState);
    }
    mEnsureNoInadvertantSuperOnCreateCallthroughs = false;
  }

  private boolean intentHasRelaunchData(Intent intent) {
    return intent.getByteArrayExtra(EXTRA_RELAUNCH_APPLICATION_DNA) != null;
  }

  private Intent getIntent() {
    return mActivity.getIntent();
  }

  private void startFromSavedInstanceState(Bundle map) {
    setClipSharing(getClipOrganizationSpace(map.getString(
        SIS_CLIP_ORGANIZATION_SPACE_CLASS_NAME, null)), map.getInt(SIS_CLIP_ID_KEY,
-1));
    mCollaborativeActivity.onCreate(map, null);
```

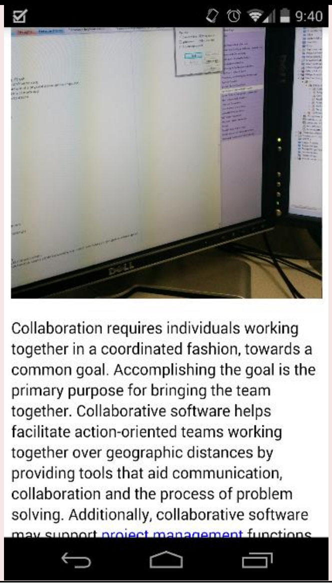
3:04 PM
http://icie.cs.byu.edu/cs656/

Date -	Topic	Readings
6 Jan -	Introduction	
8 Jan -	Classifiers	Appendix A3-A3.1c
13 Jan -	Classifiers	Appendix A3.1d-A3.1e
15 Jan -	Classifiers exercises	
20 Jan -	**Holiday**	
22 Jan -	Digital Ink	Chapter 19 pp 383-392
27 Jan -	Gesture Recognition	Chapter 19 pp 392-396, 1$ Recognizer, Music Notepad
29 Jan -	Ink Organization	Chapter 19 pp 397-418, Tivoli Basic
3 Feb -	Evaluating Interactive Systems	Chapter 26, Evaluating UI, STUF
5 Feb -	Review for Exam	
10 Feb -	**Exam 1**	
12 Feb -	Ink/Touch Design Critiques	
17 Feb -	**Holiday**	
18 Feb **Tuesday -**	Camera Interaction	Chapter 24 pp 493-516

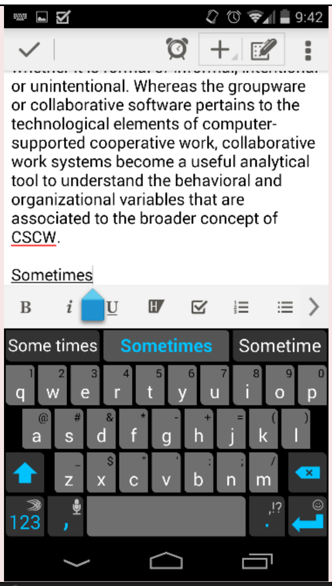
Application *Activity*	**Screenshot (for reference)**	**Application-DNA**

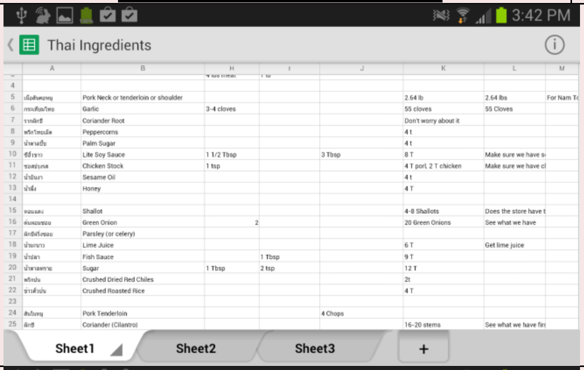
RMaps [35]		`String mapName`
		`String overlayId`
		`boolean showOverlay`
		`int centerLatitude`
		`int centerLongitude`
		`int zoomLevel`
		`boolean compassEnabled`
		`boolean autoFollow`
		`String searchResultDesc`
		`String searchResultLocation`
		`boolean showDashboard`
		`String targetLocation`

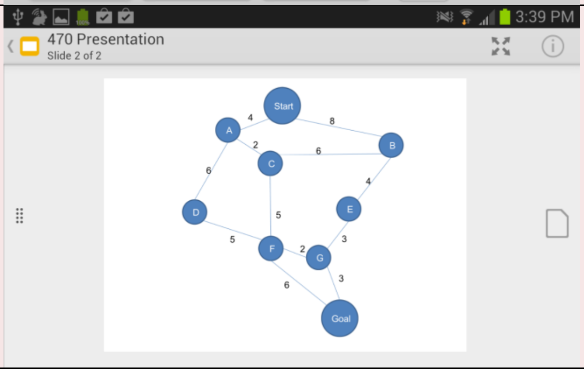
Calendar [36]	 DAY
	WEEK
	MONTH
	AGENDA
	TODAY
	Feb 2014
	February 2014
	Prep Dinner
	Dinner group
	Institute
	FHE
	Meet with
	Ward Prayer
	Penny's Senior recital
	rsshurtz@gmail.com
	Classes
	Work
	BYU Academic Calendar
	Richard Shurtz's Facebook Events
	3:01
		`long time`
		`int currentView`
		`int eventId`
		`boolean checkForAccounts`

Evernote [7]
ViewNoteActivity
	 Collaboration requires individuals working together in a coordinated fashion, towards a common goal. Accomplishing the goal is the primary purpose for bringing the team together. Collaborative software helps facilitate action-oriented teams working together over geographic distances by providing tools that aid communication, collaboration and the process of problem solving. Additionally, collaborative software may support project management functions	**`String noteId`**
		`float scrollPosition`

Application ***Activity***	**Screenshot (for reference)**	**Application-DNA**

Evernote
EditNoteActivity
		`String noteId`
		`int startSelection`
		`int endSelection`
		`char[] charsBefore`
		`char[] charsAfter`
		`boolean bold`
		`boolean italic`
		`boolean underlined`

Google Drive [37]
Spreadsheet Edit Activity
		`String documentId`
		`int sheetId`
		`int topVisibleRow`
		`int leftVisibleCol`
		`float zoomLevel`
		`int startSelectionRow`
		`int startSelectionCol`
		`int endSelectionRow`
		`int endSelectionCol`

Google Drive
PresentationActivity
		`String documentId`
		`int pageNumber`
		`boolean fullscreen`

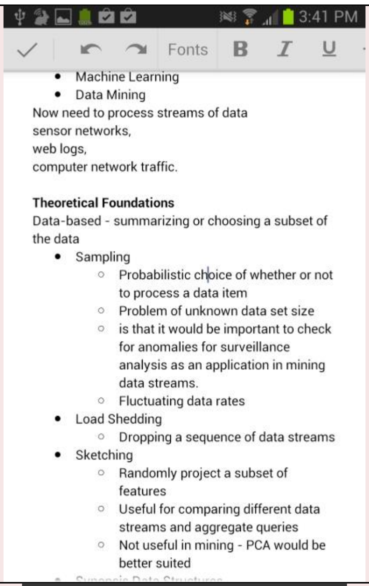
or unintentional. Whereas the groupware or collaborative software pertains to the technological elements of computer-supported cooperative work, collaborative work systems become a useful analytical tool to understand the behavioral and organizational variables that are associated to the broader concept of CSCW.

Sometimes

Thai Ingredients

	A	B	H	I	J	K	L	M
4								
5	[illegible]	Pork Neck or tenderloin or shoulder				2.64 lb	2.64 lbs	For Nam T
6	[illegible]	Garlic	3-4 cloves			55 cloves	55 Cloves	
7	[illegible]	Coriander Root				Don't worry about it		
8	[illegible]	Peppercorns				4 t		
9	[illegible]	Palm Sugar				4 t		
10	[illegible]	Lite Soy Sauce	1 1/2 Tbsp		3 Tbsp	8 T	Make sure we have s	
11	[illegible]	Chicken Stock	1 tsp			4 T porl, 2 T chicken	Make sure we have cl	
12	[illegible]	Sesame Oil				4 t		
13	[illegible]	Honey				4 T		
14								
15	[illegible]	Shallot				4-8 Shallots	Does the store have t	
16	[illegible]	Green Onion	2			20 Green Onions	See what we have	
17	[illegible]	Parsley (or celery)						
18	[illegible]	Lime Juice				6 T	Get lime juice	
19	[illegible]	Fish Sauce		1 Tbsp		9 T		
20	[illegible]	Sugar	1 Tbsp	2 tsp		12 T		
21	[illegible]	Crushed Dried Red Chiles				2t		
22	[illegible]	Crushed Roasted Rice				4 T		
23								
24	[illegible]	Pork Tenderloin			4 Chops			
25	[illegible]	Coriander (Cilantro)				16-20 stems	See what we have fir	

Application ***Activity***	**Screenshot (for reference)**	**Application-DNA**

Google Drive
DocumentActivity
	
	• Machine Learning
	• Data Mining
	Now need to process streams of data
	sensor networks,
	web logs,
	computer network traffic.
	
	Theoretical Foundations
	Data-based - summarizing or choosing a subset of the data
	• Sampling
	○ Probabilistic choice of whether or not to process a data item
	○ Problem of unknown data set size
	○ is that it would be important to check for anomalies for surveillance analysis as an application in mining data streams.
	○ Fluctuating data rates
	• Load Shedding
	○ Dropping a sequence of data streams
	• Sketching
	○ Randomly project a subset of features
	○ Useful for comparing different data streams and aggregate queries
	○ Not useful in mining - PCA would be better suited
		`String documentId`
		`int startSelection`
		`int endSelection`
		`char[] charsBefore`
		`char[] charsAfter`
		`boolean bold`
		`boolean italic`
		`boolean underlined`

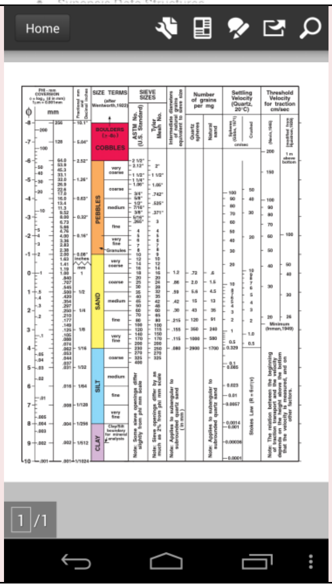
Adobe Reader [1]		`String filePath`
		`int pageNumber`
		`float zoomLevel`
		`float leftScrollPercent`
		`float topScrollPercent`

Figure 24 - A table of various Android applications and their associated application-DNA. Those in green are applications for which we actually implemented application-DNA saving and restoring. Those with red backgrounds show the expected application-DNA.

From the examples above, we can observe a few patterns about what application-DNA typically consists of. Almost all of the applications (with the exception of RMaps and Calendar) need to save some sort of a reference (either by an ID or by file path) to the "document" the user is working with. This piece of information is clearly vital in restoring the application's state.

Beyond that, the rest of the application-DNA is generally used to restore editing options and to get the application view back to where the user was working with the given document.

In the case of RMaps, the vital pieces of information are the longitude, latitude, and the current zoom level. In the case of Calendar, it is the “time” and the “view” (month, day, week, etc.) which are vital.

In all these cases, the application only saves a *reference* to its model rather than the *contents* of its model. Rather than saving the html of the webpage, we save the URL to the webpage; rather than saving the content of a text file, we save the file path of the text file. This is vital in ensuring that application-DNA remains small and can be saved in real-time and sent over the network with the clip updates to the display server. We keep the application-DNA relatively simple and lightweight but also sufficiently informative enough to restore our target “90%” of the state for “10%” of the development effort.

4.2.2 Exceptions

Of course, because files, users, credentials, and documents change over time, there will be cases where such simple application-DNA data will be insufficient to restore the application’s state. Sometimes this failure to restore is due to the fact that it is no longer possible (e.g. the file has been deleted, or the user no longer has permission to access a shared document in the cloud). Other times, the failure is due to the fact that the contents of the document have changed and no longer match what the application-DNA is “expecting.” For the first of these two cases, the best we can hope to do is provide a message to the user explaining why we were unable to restore the application’s state.

For the latter case, developers can take certain measures to add robustness. For example, in the 920 Text Editor’s application-DNA, not only do we include the current cursor/selection character position, but we also include the 50 characters before and the 50 characters after the position/selection (see Figure 24**)**. The problem with just saving the cursor/selection character position is that the contents of the file are likely to change from one meeting to another. Thus, we can not depend on the character indices matching up when re-launching the application. When we include the 50 characters before and after the cursor, however, then we have a mechanism for finding the same location in the document regardless. (We discuss this in more detail under 'Conversion Summaries' in Chapter 5.)

4.3 Evaluation

In all, we converted seven different existing Android applications to fit our framework. Most of this conversion work is documented in Chapter 5. However, here we present the data regarding the size of the application-DNA byte sequences produced in practice by these applications (Figure 25).

Application	**Application-DNA Size**	**Time to Save and Serialize**
Calendar	1001 bytes	1.50 ms
RMaps	1246 bytes	2.88 ms
Gospel Library	3015 bytes	4.24 ms
Zirco Browser	900 bytes	1.15 ms
Email	1070 bytes	1.68 ms
Arity Calculator	1000 bytes	1.98 ms
920 Text Editor	1273 bytes	1.7 ms
Average	*1358 bytes*	*2.16 ms*

Figure 25 – A summary of the typical size of application-DNA for various applications and the time to save on a Google Nexus 5 device

The average size of the application-DNA for a program is just 1358 bytes and is never more than a few kilobytes. Also, using a Google Nexus 5 device, we found that it takes on average 2.16 milliseconds to save and serialize the application-DNA. We should note that we use the built-in Java serialization method, which tends to be relatively inefficient in both size and speed (for example, a method trace reveals that 95% of the times recorded above are spent in serializing). While we could have developed a more efficient serialization method to meet our needs, in practice, users will not notice the difference between 4 milliseconds and the .5 milliseconds that we might be able to achieve. Likewise, the potential size savings would be negligible in use.

4.4 Summary

The combination of the launch-action string and application-DNA provides a mechanism for using clips to re-launch applications to the state they were in when they were shared. We have shown that this solution works for a wide variety of popular applications. We have shown that the application-DNA is relatively simple (less than a dozen distinct pieces of information in the applications surveyed in Figure 24). We have shown by the data in Figure 25 that, in practice, the application-DNA is very small and lightweight. The final requirement for application re-launching is that it should require only little additional developer effort. In Chapter 5 we show that our solution meets this requirement.

Chapter 5 – Android Clip Application Framework

The concepts of launch-action strings and application-DNA allow developers to design applications that can be shared and then later re-launched to their shared state. Practically, however, developers should not need to implement the functionality to connect to a display server, package clips and send them over the network, or any other task which will be common across clip applications. What we need, therefore, is a framework that can take care of these common tasks behind the scenes and allow developers to focus their time and effort on the few parts of the system which are specific to their own application. We built the Android Clip Application Framework to meet these needs.

The ultimate goal of the framework is to allow developers to add clip functionality to arbitrary Android applications without requiring any extensive effort on their part.

5.1 ClipActivity

As mentioned previously, the basic UI component of any Android application is an 'activity.' Developers use the Android framework's Activity class as the base class for their own activity classes. To re-iterate our previous example, an email application might have a few different activities. The developer might write code for a MessageComposeActivity, MessageListActivity, and MessageViewActivity. All of these would inherit from Activity.

We provide as the main part of the Android Clip Application Framework a new base activity class called 'ClipActivity.' We are able to implement most of the functionality of a clip application in the ClipActivity class without changing its usage much from the normal Activity class. Thus, when developers create an activity that inherits from ClipActivity rather than Activity, it gains most of the clip functionality automatically but is still written essentially the same as it would be if it had inherited from Activity. In general, the only major difference is in implementing the activity's application-DNA.

5.2 ClipActivity Functionality

There are essentially two tasks required of a clip application. One is to send application clips to the display server as the state of the application changes. This is what enables the clip application to be shared on the display in real-time. The second is to restore the state to match a saved clip when the user requests a clip re-launch. We will describe how our ClipActivity takes care of these tasks.

5.2.1 Sending Clips to Display Server

The first step in sending clips is connecting to the display server. To do this, we provide a menu option to "Share App" and then a dialog which allows the user to specify which LearnSpace display server he would like to connect to. Figure 26 below shows these components. We provide this menu option (as well as a menu option to save a "Snapshot" locally) automatically to any activity class that inherits from ClipActivity. Once sharing has started, this menu option changes to "Stop Sharing."

After having established a connection, the ClipActivity pieces together the application clip and sends it to the display server. To do so, it collects the three components of a clip: the clip image, the launch-action string, and the application-DNA. The first two can be generated automatically behind the scenes in the ClipActivity base class with no additional effort from the application developer. The ClipActivity base class has access to the root view, so it simply redraws the view hierarchy as it would be displayed on the screen but to its own private bitmap. It then has the clip image. The application package name, the activity class name, and the device

ID are all easily accessible by the ClipActivity base class as well, so piecing together the launch-action string is very straightforward. The developer just needs to specify whether the application should be allowed to be re-launched on any device or if it should be limited to its original device. To do so, she overrides a method on the ClipActivity base class (see Figure 32 – H).

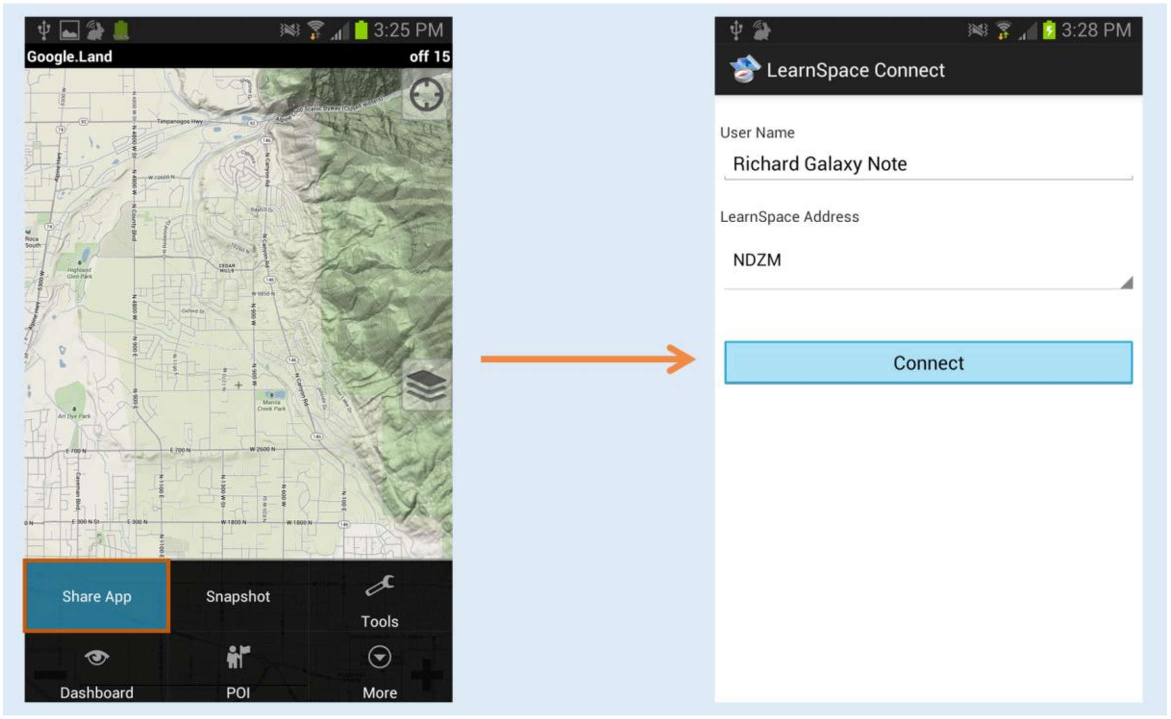


Figure 26 – On the left, the user selects to share the application to the shared display. On the right, the user is given a dialog to connect to the LearnSpace server. These are both provided by our framework.

The third clip component, the application-DNA, however, does require a specific implementation for each specific clip activity. The developer must implement ClipActivity's abstract method saveApplicationDna(AppDnaMap). (We discuss this further under the 'AppDnaMap' heading below.) The ClipActivity base class calls this method to get the application-DNA. After obtaining all three clip components, ClipActivity uses the LearnSpace Client API to upload the image and re-launch data as data assets on the server, create a sheet, and set the sheet to display the clip image.

After sending the first clip to the display server, the ClipActivity begins the process of sending clip updates. A clip update contains the current application-DNA and the parts of the image/view which have changed. The ClipActivity determines these areas by intercepting the redraw method calls to the windowing system. It can then just redraw the damaged portions of the view to its bitmap. Of course, we do not attempt to send every single view update as a clip update. Rather, we limit it to at most a few times a second and only send a new update when the previous one has finished sending. In practice, this results in sending around 2.2 to 4.2 clip updates per second. So, a user sharing his application can expect the image on the shared display to update at least a couple times a second as he interacts with his application.

5.2.1.1 AppDnaMap

As mentioned in Chapter 4, a piece of application-DNA can be any arbitrary sequence of bytes. Rather than have developers deal with byte arrays, however, we provide a data structure that allows the developer to save different pieces of information in a string keyed map. From within our framework's code, a call is made to the abstract saveApplicationDna(AppDnaMap) method. Figure 27 below shows an actual implementation of this method and an example of how the AppDnaMap object is used. After giving the specific activity a chance to save its unique application-DNA information, the ClipActivity base class serializes this object to produce the application-DNA byte sequence which is sent over the network.

```
public void saveApplicationDna(AppDnaMap outState) {
    outState.putLong("Time", mController.getTime());
    outState.putInt("View", mCurrentView);
    if (mCurrentView == ViewType.EDIT) {
          outState.putLong("EventId", mController.getEventId());
    }
    outState.putBoolean("CheckAccounts", mCheckForAccounts);
}
```

Figure 27 – A code snippet of saving application-DNA taken from the Calendar application

5.2.1.2 Dealing with Android Activity Lifecycle

Another responsibility of our framework is to take care of the clip sharing behind the scenes even across possibly complicated application lifecycles. Android applications can be paused at any time (if the user "minimizes the application" by pushing the system home button). If a user is sharing his application to the shared display, and then minimizes it to work with a different application, he will not want to stop sharing to the display. However, background threads and other resources relating to clip sharing ought to be freed. Thus, when the application is paused, the ClipActivity base class frees the resources related with sharing. Then, when the application is resumed, it resumes clip sharing to the same clip which has been persistent on the shared display.

Also, a paused application can be shut down by the system (if the system needs to free additional resources), but it is expected that the application can be restored to the same state when the application is launched next. (The goal being that the user does not even know that the application was killed in the background.) Thus, our ClipActivity also takes the necessary steps to be able to reconnect and resume sharing clip updates to the same clip on the display server. Once again, the benefit of all this effort is that the developer need not worry about the specifics of saving, pausing, resuming, etc. relating to clips, but can treat the activity as he would essentially any other activity.

5.2.2 Re-Launching from Application-DNA

Beyond sharing clips, the ClipActivity base class also helps in the process of clip re-launching.

Before describing how it does this, however, let us first describe how Coll-app-oration re-launches a clip. It uses the information in the launch-action string to put together an 'Intent' object. The Android system uses these Intent objects to launch all applications/activities. Coll-app-oration then attaches the application-DNA byte array to the Intent. If the clip being re-launched is already shared on the display, then Coll-app-oration also attaches a reference to that clip. It then gives the Intent object to the system to perform the launch. When the clip activity is launched, it has access to the Intent object that the system used. The code in the ClipActivity base class detects the attached-DNA, de-serializes it, and then makes an appropriate call to the implementing sub-class to re-store its state. This is done by calling the `onCreate(Bundle, AppDnaMap)` method. Android activities typically implement the `onCreate(Bundle)` method, but our framework has clip activities implement the method with the additional AppDnaMap parameter. Figure 28 shows (part of) an actual implementation of this method. It is the code to restore the state matching the application-DNA saved in Figure 27.

```
public void onCreate(Bundle icicle, AppDnaMap savedDNA) {
    ...
    if (savedDNA != null && savedDNA.containsKey("CheckAccounts")) {
        mCheckForAccounts = savedDNA.getBoolean("CheckAccounts");
    }
    ...
    if (savedDNA != null){
        timeMillis = savedDNA.getLong("Time");
        viewType = savedDNA.getInt("View", -1);
    } else {
        ...
    }
    ...
}
```

Figure 28 - A code snippit of restoring state from application-DNA taken from the Calendar application

If there was a reference to a clip already on the shared display, then the ClipActivity will automatically connect to the display server and begin sending clip updates. This has the effect for the user of simply resuming the application as it was on the shared display. As he continues to interact in the application, it will continuously be updated on the shared display.

The re-launching process is made more difficult by complicated Activity lifecycles – especially in the case where new Intent objects are given to an already running activity rather than starting a new activity – but our implementation takes care of these cases as well. The end result is that the application developer need not fret with any of these details, but can focus her efforts on implementing the saving and restoring of state via application-DNA.

5.2.3 Exiting an Activity

Most of the time, when a user leaves an activity or application, he would wish to stop sharing to the display. It is sometimes the case, however, that the user wishes to keep the clip shared on the display even after leaving the activity. Consider, for example, the case where Henry is sharing the study guide from his email application to the display for the group to read. He then remembers a second email from the teacher that he wants to share as well. In order to go back to the inbox activity, however, he must exit the view-email activity. In order to allow him to keep the first email on the display, he should have the option to keep the study guide shared to the display when exiting the view-email activity. We have implemented such an option in the ClipActivity base class. If the activity is currently sharing to the display, and the user chooses to exit the activity, the user is prompted whether or not he wants to keep the shared clip on the display.

5.2.4 Using a different base Activity class

Not all activities inherit from the standard Android Activity class. For example, the Android Application Framework also provides a ListActivity class. It is entirely possible that an application could use any sub-class of Activity as its base class for its own activity. Thus, we needed to provide a way to not only have a ClipActivity, but also a ClipListActivity, or a Clip<Anytype>Activity.

Our solution was to follow the delegation pattern and extract the functionality of the ClipActivity class to a separate ClipActivityFunctionality class. Then, the ClipActivity class inherits from Activity, and just makes calls through a ClipActivityFunctionality object. The functionality of the extracted class is defined by the IClipActivity interface. To make a ClipListActivity, we just inherit from ListActivity and implement IClipActivity and then make calls though a ClipActivityFunctionality object. The only difference between the code in the ClipActivity.java file and the ClipListActivity.java file are the name of the class and the name of the class it is inheriting from.

We give an example of how a developer could follow the pattern used to make the ClipListActivity in order to make a new Clip<Anytype>Activity base class. We examine converting EventInfoActivity which uses an AbstractCalendarActivity as its base class (Figure 29).

```
public class EventInfoActivity extends AbstractCalendarActivity {

    EventInfo code
}
```

Figure 29 – The class definition of EventInfoActivity

In order to keep the functionality of AbstractCalendarActivity but still add the functionality of ClipActivity, we need to create a new class: AbstractClipCalendarActivty. To implement this, we simply define the class as seen in Figure 30.

```
public abstract class AbstractClipCalendarActivity
        extends AbstractCalendarActivity
        implements IClipActivity {

    150 lines of copy and pasted code from ClipActivity
}
```

Figure 30 – The class definition of AbstractClipCalendarActivity

Once we have done so, we follow the conversion process for EventInfoActivity, but instead of inheriting from ClipActivity, it simply inherits from AbstractClipCalendarActivity (Figure 31).

```
public class EventInfoActivity extends AbstractClipCalendarActivity {

    Required changes to fit framework

    EventInfo code
}
```

Figure 31 – The modified EventInfoActivity class using a new base class to add clip functionality

For cases like this, the extra work for the developer is actually very minimal. He simply creates a new class definition with the entirety of the implementation (which is essentially just calls through to a ClipActivityFunctionality object) being copy-and-pasted from ClipActivity (Figure 30).

While it would be ideal to eliminate the need to copy-and-paste this code, it is an inherent limitation with the Android Activity architecture. The system makes application lifecycle related calls on the Activity. We need to use those calls in our ClipActivityFunctionality class. Thus we need a class that inherits from Activity to pass those through. If there were a way to add callbacks for these system calls to an Activity object, we could add the functionality easily to any Activity object. However, the base Activity class doesn't support this, and while we could create our own, it would still be incompatible with an application that uses a third party custom activity class as the base class for one of their activities. Since we cannot re-write all existing activity classes, we provide a simple solution that requires minimal developer effort even if it does require copying-and-pasting.

5.2.5 Auto-Save/Restore Intent

Some of the activity's state is already specified in the activity's current Intent. Since, Android applications are all started using Intent objects, the Intent object often specifies what piece of data an activity should open. For example, consider an email application. When the user selects a specific email in the InboxActivity, it creates an Intent to open MessageActivity and attaches the email's unique ID to the Intent object. The email ID is a needed piece of data that the MessageActivity should include in its application-DNA. After seeing this pattern multiple times, we created a mechanism to save and restore Intents to/from AppDnaMap objects. By default, the ClipActivity base class will save and restore the activity's Intent behind the scenes. When saving, it will automatically add it to the AppDnaMap before serializing it. On restoring, it will extract it from the map and set the activity's Intent before giving control to the sub-class. The end result is less work for the developer to get a working clip Activity.

5.3 Evaluation

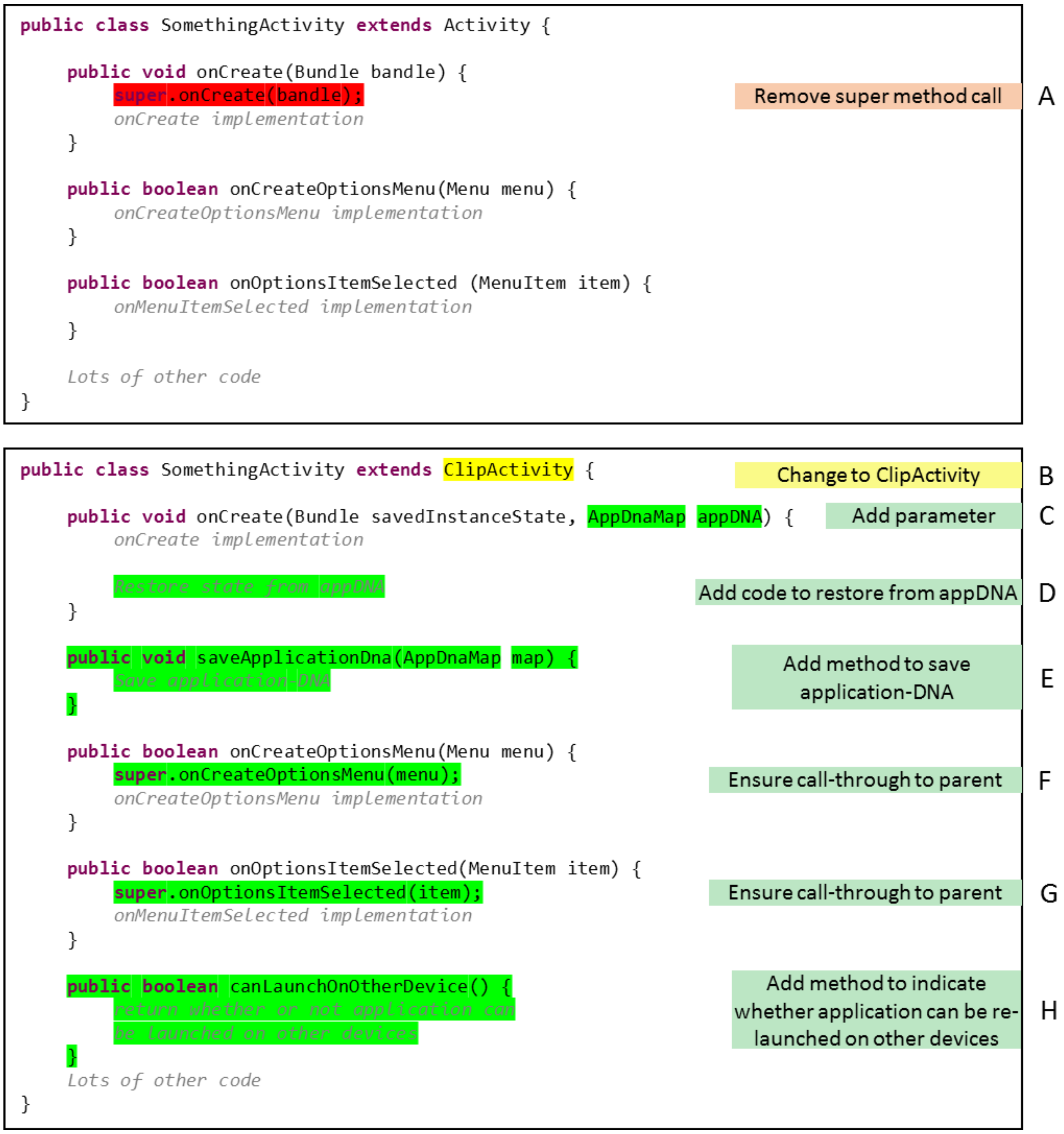
The end goal of providing this functionality through the ClipActivity base class is to enable developers to treat their application, for the most part, as any normal Android application. To show that we met that goal, we describe the process of converting existing Android applications to become clip applications using our framework.

Because Android applications are almost always comprised of multiple activities, the conversion process includes choosing which activities should be converted into shareable clip activities. We chose to convert the main, core activities, but ignored activities related to settings or other secondary activities. For example, in the Calendar application, there is a calendar-settings activity which lets the user change the calendar's appearance, or choose whether or not they would like to receive notifications. Chances are extremely slim that a user would want to share this activity.

5.3.1 App Conversion Instructions

After creating and testing our framework initially, we created step-by-step instructions for converting an existing Android application into a clip application. These are recorded in a one-and-a-half page document (Appendix A). The conversion process is relatively simple; we offer a brief overview of what the instructions include. Figure 32 accompanies these instructions.

- At an application level
    - Add AndroidClipApplicationFramework.jar and LearnSpaceClient.jar to build path
    - Make a couple minor copy-and-paste changes in the Android manifest file
- For each activity being changed to a clip activity
    - If needed, create a new Clip<?>Activity class to use for the base class of the activity to be converted (see 5.2.4)
    - Make minor one-or-two line changes in three or four places (Figure 32 – A, B, C)
    - Ensure that the parent class's (super) method is being called in up to 11 of Activity's methods that may be overridden in the child activity class (Figure 32 – F, G)
    - Implement saving application-DNA by implementing the onSaveApplicationDNA(AppDnaMap) method (Figure 32 – E)
    - Implement restoring application state in the onCreate(Bundle, AppDnaMap) method (Figure 32 – D)



```
public class SomethingActivity extends Activity {

    public void onCreate(Bundle bandle) {
        super.onCreate(bandle);
        onCreate implementation
    }

    public boolean onCreateOptionsMenu(Menu menu) {
        onCreateOptionsMenu implementation
    }

    public boolean onOptionsItemSelected (MenuItem item) {
        onMenuItemSelected implementation
    }

    Lots of other code
}
```

```
public class SomethingActivity extends ClipActivity {

    public void onCreate(Bundle savedInstanceState, AppDnaMap appDNA) {
        onCreate implementation

        Restore state from appDNA
    }

    public void saveApplicationDna(AppDnaMap map) {
        Save application-DNA
    }

    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        onCreateOptionsMenu implementation
    }

    public boolean onOptionsItemSelected(MenuItem item) {
        super.onOptionsItemSelected(item);
        onMenuItemSelected implementation
    }

    public boolean canLaunchOnOtherDevice() {
        return whether or not application can
        be launched on other devices
    }
    Lots of other code
}
```

Remove super method call
A
Change to ClipActivity
B
Add parameter
C
Add code to restore from appDNA
D
Add method to save application-DNA
E
Ensure call-through to parent
F
Ensure call-through to parent
G
Add method to indicate whether application can be re-launched on other devices
H

Figure 32 - An example of changing a normal activity to become a clip activity. The top part is the original activity code, while the bottom shows the modified activity. The highlighting and labels on the right indicate the changes made.

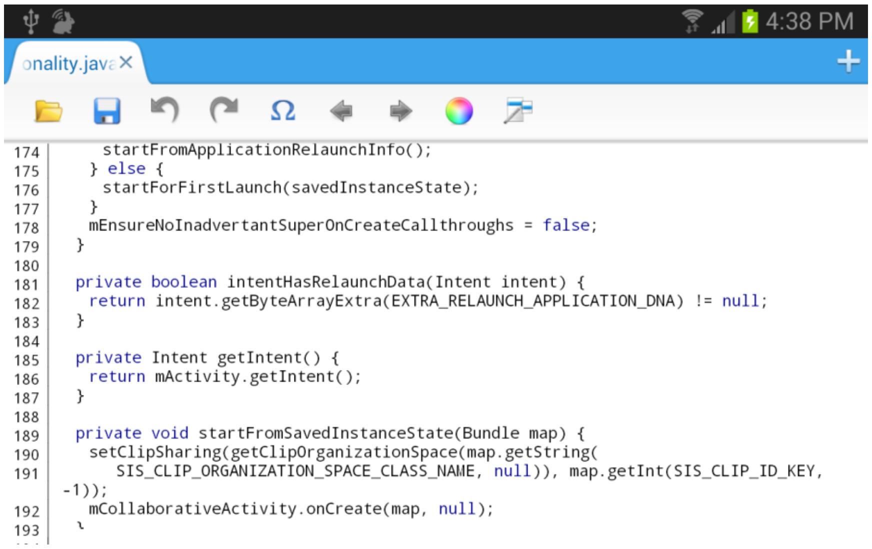
In all, most of the instructions are simple checks or changes that only require a line or two of code. The only significant modification to the application should be designing and implementing saving and restoring from application-DNA.

5.3.2 Conversion Summaries

We found seven existing open source applications that we were able to convert from normal Android applications into clip applications. We documented the conversion process for six of these applications. (The seventh was used for testing while developing the framework.) Our record indicates how much time it took and what difficulties we ran into. After completing the conversion, we compared the converted application code to the original code to see how many files were edited and how many lines of code we changed.

5.3.2.1 920 Text Editor

920 Text Editor [33] is a simple text editor which can be used for editing code files. It provides highlighting according to the syntax of a number of languages. We show a screenshot of 920 Text Editor in Figure 33.



```
      startFromApplicationRelaunchInfo();
    } else {
      startForFirstLaunch(savedInstanceState);
    }
    mEnsureNoInadvertantSuperOnCreateCallthroughs = false;
  }

  private boolean intentHasRelaunchData(Intent intent) {
    return intent.getByteArrayExtra(EXTRA_RELAUNCH_APPLICATION_DNA) != null;
  }

  private Intent getIntent() {
    return mActivity.getIntent();
  }

  private void startFromSavedInstanceState(Bundle map) {
    setClipSharing(getClipOrganizationSpace(map.getString(
        SIS_CLIP_ORGANIZATION_SPACE_CLASS_NAME, null)), map.getInt(SIS_CLIP_ID_KEY,
-1));
    mCollaborativeActivity.onCreate(map, null);
  }
```

Figure 33 – A screenshot of 920 Text Editor editing a Java source code file.

The conversion for 920 Text Editor was a little difficult for two reasons. First, because the file with which the user is working can change in content from session to session, we had to take some precautions to ensure we could still restore the state upon re-launching the application.

The second is that this application uses its own custom menu rather than the default system menu which our framework uses by default.

We had to take some precautions in our application-DNA implementation in order to ensure that we could recover the edit position even if the text of the file changes from session to session. We chose to save the 50 characters before and after the current cursor position. We could then use this to find those same characters – or the closest match of those characters – when the application is re-launched. In order to do this, however, we had to implement an algorithm that can do minimum-edit-distance string comparisons to find the closest match within a file.

Because 920 Text Editor uses its own custom menu, we had to implement changes to the application's menu code to include the two menu options normally included by default in our framework. This increased the number of changes we had to make to the application beyond what would normally be required.

Overall, the app conversion took 455 minutes of the developer time. Of that, 260 minutes were devoted to implementing and testing the minimum-edit-distance algorithm. About 30 minutes were spent editing the menu code. Overall 304 lines of code were added or edited. We created 1 new source code file (to implement the Smith-Waterman alignment algorithm), and modified six others. Only one activity was changed to a clip activity: the text-editor activity.

5.3.2.2 Arity Calculator

Arity [38] is a basic calculator with some graphing functionality. It keeps track of past expressions and allows the user to assign values to arbitrary variables. These variables can be used in mathematical expressions or in functions which can be graphed. We show a screenshot of Arity Calculator in Figure 34.

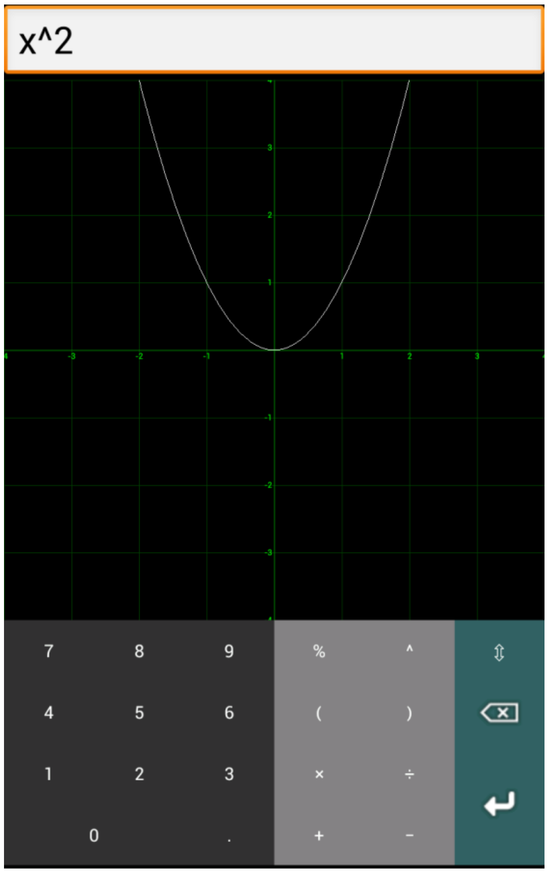


Figure 34 – A screenshot from Arity Calculator. It is graphing the function $y = x^2$.

The most difficult aspect in converting Arity Calculator [38] was figuring out what to save for the application-DNA. We want to be able to save the current mathematical expression or equation, but it may be dependent on the previous expressions or saved variables.

We decided, therefore, to save not only the current expression or function, but also the history, and the currently saved variables. This required some extra work, because the application did not have a consistent method to serialize these pieces of data. We also had to make modifications to other areas of the application due to some poor design choices in the original application. For example, the application was dependent on some global static variables which were accessed across different activities. This was a poor design choice that required extra changes on our part when we implemented restoring from application-DNA.

Overall, the process took 150 minutes of developer time, and 223 lines of code were added or modified. We created one file to help with the serialization of the function data-type, and then made changes in 5 other files. We changed two of the application's activities to clip activities: the calculator activity, and the show-graph activity.

5.3.2.3 Calendar

Calendar [36] is a Google-developed, pre-installed Android application. It allows events from multiple calendars to be imported and shown together. Users can view their calendar at a day, week, or month timeframes. We provide a screenshot from Calendar in Figure 35.

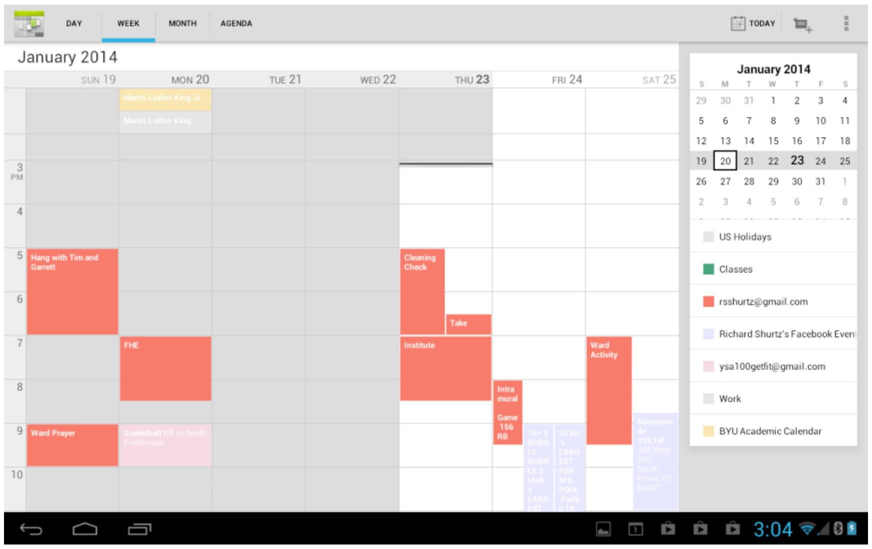


Figure 35 – A screenshot from the Calendar application. This is viewing the calendar at the week view.

Calendar is professionally designed, and uses best practices for a well-designed Android application. This made the conversion very straightforward. The largest difficulty was just in understanding how different pieces fit together in such a large code base (a total of 203 source files and 53,623 lines of code).

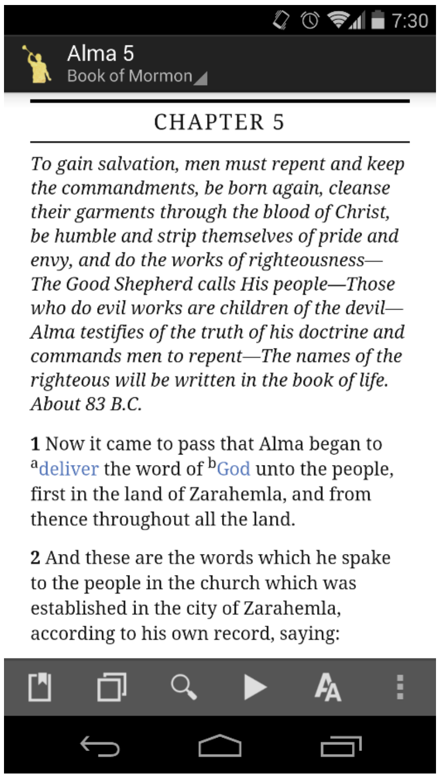
We converted 4 different activities: the calendar view activity, event info activity, the search activity, and also the edit event activity. Because the EditEventActivity class inherits from

AbstractCalendarActivity rather than the default Activity class, we had to create a new file for a ClipAbstractCalendarActivity class. As described earlier, the content of this class is essentially just copy-and-pasted from the ClipActivity class. So, while it took an extra file and 116 lines of code, the developer effort required was extremely small.

In summary, we created 1 new source file and modified 6 others. We added or edited 201 lines of code (but 116 of these were copy-and-pasted). In total, we spent 186 minutes of developer time converting the application.

5.3.2.4 Gospel Library

Gospel Library is "the gospel study app of The Church of Jesus Christ of Latter-day Saints" [32]. It functions as an e-reader and allows the user to mark passages, leave notes, make bookmarks, and much more. Figure 36 below shows a screenshot from Gospel Library.



CHAPTER 5

To gain salvation, men must repent and keep the commandments, be born again, cleanse their garments through the blood of Christ, be humble and strip themselves of pride and envy, and do the works of righteousness—The Good Shepherd calls His people—Those who do evil works are children of the devil—Alma testifies of the truth of his doctrine and commands men to repent—The names of the righteous will be written in the book of life. About 83 B.C.

1 Now it came to pass that Alma began to [a]deliver the word of [b]God unto the people, first in the land of Zarahemla, and from thence throughout all the land.

2 And these are the words which he spake to the people in the church which was established in the city of Zarahemla, according to his own record, saying:

Figure 36 – A screenshot of the Gospel Library application. This activity is viewing a chapter of scripture.

Gospel Library was the largest application (337 source files and 76,277 lines of code) we converted, and the conversion process was once again relatively straightforward as the app is professionally designed. The only difficulty was that Gospel Library uses a 3rd party Android framework extension called ActionBarSherlock [39], which interferes with our Android Clip Application Framework 'extension.'

The first area where this presented a difficulty is that the activities we converted did not inherit from the base Activity class, but rather from SherlockListActivity and WindowActivity. This meant we had to create two new base classes following the pattern described previously. The other difficulty of using the ActionBarSherlock extension was that rather than using the system menu objects, the extension expects 'Sherlock' menu objects. This just meant that we had to add the menu options explicitly rather than relying on the default implementation that the framework provides.

Other than that, the difficulty was just in trying to understand such a large and complicated application. About half of our developer time was spent experimenting with different application-DNA options to try to find one that could best recover the user's reading position. The option we ended up selecting was to use a 'bookmark' feature that the system already uses for pinpointing specific verses or paragraphs in the library content. Had we been more familiar with the application's features, the conversion time could have been cut roughly in half.

We spent 238 minutes altogether converting the app. We added or edited 428 lines of code, but 232 of them we copy-and-pasted to create the new clip activity base classes. For those, we added 2 new source code files, and, beyond that, we modified 5 other files. Overall, we converted 2 of the application's activities to clip activities: the content activity and the catalog activity.

5.3.2.5 RMaps

RMaps [35] is a map application that can access a variety of maps from different providers (for example, you can view satellite image maps, road maps, terrain maps, etc.). Users can pan and zoom, mark points of interest, measure distances, and more. We provide a screenshot from RMaps below in Figure 37.

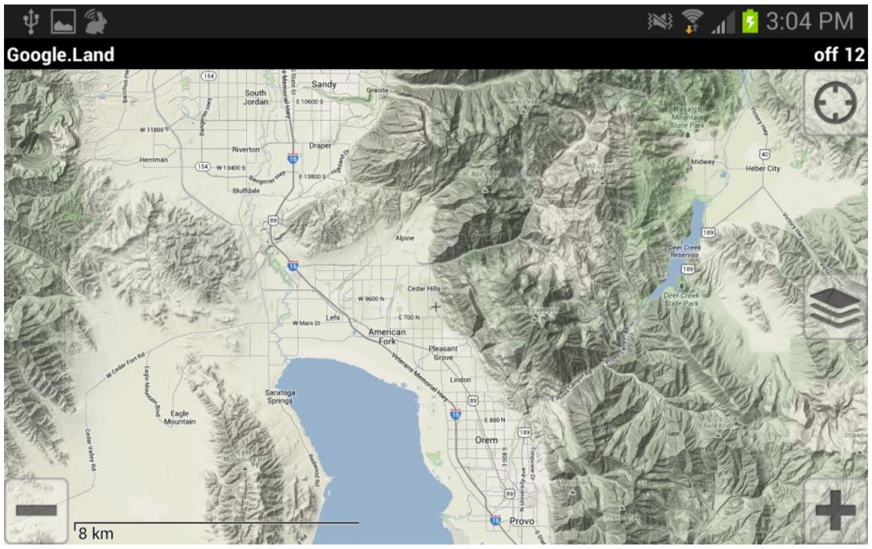
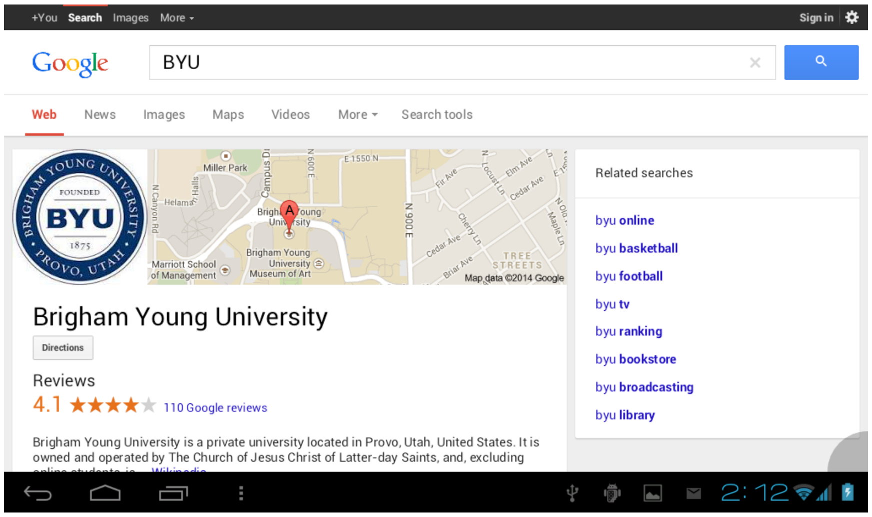


Figure 37 – A screenshot of the RMaps application. It is viewing the 'Google.Land' map.

The conversion of RMaps ended up being very straightforward and had few difficulties. Most of the time was spent between figuring out what information was most important to save for application-DNA and testing. We spent 90 minutes working on the application to convert it. We converted just the main map view activity. We modified 3 different files and added or edited 104 lines of code.

5.3.2.6 Zirco Browser

Zirco Browser [34] is a very simple web browser built using Android's default WebView [40] widget. It provides some additionally to make and use bookmarks and also has an ad-blocker. We show a screenshot of Zirco Browser below in Figure 38.



Brigham Young University is a private university located in Provo, Utah, United States. It is owned and operated by The Church of Jesus Christ of Latter-day Saints, and, excluding

Figure 38 – A screenshot from Zirco Browser viewing a Google search results page.

Zirco Browser ended up being another easy-to-convert application. We converted just the application's main activity, and it required edits to just 2 files. Overall, we added or edited just 28 lines of code. The total developer time was 78 minutes.

5.3.3 Conversion Measurements

Figure 39 below shows a detailed summary of how many files and lines of code were edited or added for each of the application conversions.

	Original Number of Source Files	Original Number of Lines of Code	Number of Files Modified	Number of Files Added	Number of Lines of Code Added	Number of Lines of Code Modified	Total Lines of Code Touched
920 Text Editor	88	9,841	6	1	285	19	304
Arity	30	2,723	5	1	209	14	223
Calendar	302	53,623	6	1	175	26	201
Gospel Library	337	76,277	5	2	411	17	428
RMaps	224	26,323	3	0	92	12	104
Zirco Browser	142	13,781	2	0	22	6	28

Figure 39 – A summary of the code edits we made to convert these six applications to clip applications

In Figure 40 below, we summarize the difficulty and scope of each app conversion. Rather than report the total lines of code added or edited, we adjust these values to discount the lines of code which were just copy-and-pasted to create new base clip activity classes.

	Number of Activities changed to Clip Activities	Number of hours spent implementing and testing conversion	Original Number of Source Files	Original Number of Lines of Code	Number of Files Edited or Added	Total Lines of Code Edited or Added (adjusted for copy-and-paste classes)
920 Text Editor	1	7.6 (3.25 without min-edit-distance)	88	9,841	7	304
Arity	2	2.5	30	2,723	6	223
Calendar	4	3.1	302	53,623	7	85
Gospel Library	2	4	337	76,277	7	196
RMaps	1	1.5	224	26,323	3	104
Zirco Browser	1	1.3	142	13,781	2	28

Figure 40 – The measured data summarizing the difficulty of converting each of the applications

Looking at the numbers in the two figures above, it is evident that none of the conversions took a significant amount of developer effort. The most time was spent on converting 920 Text Editor, but even that was still shorter than a single work day. In the case of large well-designed applications (Calendar and Gospel Library), the conversion still took fewer than 4 hours, which is significant considering that we were not familiar with the workings of these applications beforehand.

The effort required by the developer can essentially be broken into two parts: the effort of fitting the application to the framework (making the minor edits in the activity class file and in the manifest file), and the effort of implementing the application-DNA functionality. We suspect that because we were already familiar with the framework, we were probably able to complete the task of fitting the application to it relatively quickly. But, because we were unfamiliar with the data required by the application, we probably were relatively slow in designing and implementing the application-DNA components. Because the two tasks are highly interconnected, we do not have clear records on how much time was spent on each aspect. We provide our best estimate of the effort split below in Figure 41**.**

	Estimated Effort Split (Framework/DNA)	Total Time Converting Application	Estimated Number of Hours Spent Fitting to Framework	Estimated Number of Hours Spent on Application-DNA
920 Text Editor	25/75	7.6	1.9 (menu)	5.7 (string-alignment algorithm, need to edit different files)
Arity	40/60	2.5	1	1.5
Calendar	70/30	3.1	2.17 (large application, 4 activities)	.93
Gospel Library	50/50	4	2 (menu, large application)	2 (did not know about bookmark)
RMaps	50/50	1.5	.75	.75
Zirco Browser	60/40	1.3	.78	.52

Figure 41 – A table showing the split of developer effort between ‘fitting the framework’ and ‘implementing application-DNA’ during the conversion of the six applications. The notes in parentheses summarize some of the issues that required extra time.

Even with a large application like Calendar, where we converted four different activities to clip activities, we still spent fewer than two and a half hours fitting to the framework. The time spent on application-DNA seems to vary a bit more, with the time spent on 920 Text Editor being a clear outlier. Even though it took somewhere around 5.7 hours of developer time to implement and test the application-DNA, that time is minimal compared to the overall development time of an application. It was also done by someone who was, prior to the conversion, unfamiliar with the application.

5.4 Summary

Using our framework, and with relatively little developer effort, we were able to convert a broad range of existing Android applications into clip applications. They are able to fully participate in our collaborative system. They can connect to the LearnSpace server, and then share clips to the server as the user continues to use the application. These clips can be downloaded by others, and then be used later to re-launch and restore the state of their respective applications. All of this is done seamlessly so that the application behaves just as any normal Android application – even accounting for the difficulties of the Android activity lifecycle. All of this functionality is gained for less than one day of developer effort.

Chapter 6 – Clip Space API

The ability to use clips as a mechanism to share arbitrary applications with other users in a meaningful way creates new interesting opportunities for collaborative work. Because clips are such a simple data type (an image, a byte sequence, and a string), virtually any application on any platform should be able to share itself as a clip. For the same reason, creating systems that can use clips should be no harder than creating systems that deal with images. Inherent in the simplicity of clips is the flexibility to be used in many possible ways. We already have given the example of using clips as a data type in Schulte's collaborative applications. We can also envision other areas where they might be used.

For example, consider a large table-top touch display (see Figure 42). The display already lets users work together by passing images and applications across the display. Users can use the touch input to interact with the applications. However, the system is limited to only the applications installed on the table-top computer. If, however, we opened up the system to accept clips from clip applications, then it would open up a new world of possibilities. Users could share applications from their mobile devices, and then position, rotate, or provide input to them just as they would normally (the pink 'application' in Figure 42). We could develop a method to redirect application input on the display back to the application running on the mobile device. In this way, the table-top becomes an interactive surface to work from a variety of applications coming from a variety of mobile devices. Of course, the main benefit of using clips is that users can save them at any point and then use them to resume working later.

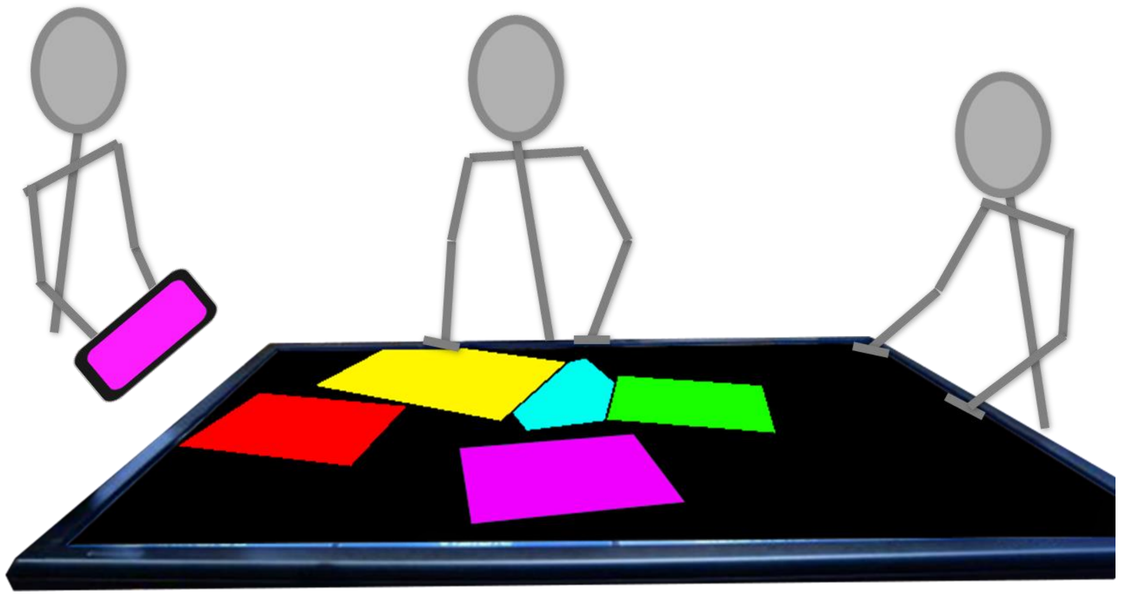


Figure 42 – Users interact on a large table-top touch display.

Clips and clip applications should not be limited, therefore, to a single collaborative space. Clip applications should be able to connect and share to a variety of systems. This will open up opportunities for future work to explore other areas (collaborative or otherwise) where application sharing via clips is a useful paradigm. Thus, we need to ensure that there is a simple interface that is easy to work with for both the clip application, and the 'clip space.' A 'clip space' can be any application that uses clips. Our collaborative space we have implemented using the LearnSpace display server and Coll-app-oration would be one such clip space. The table-top display discussed above would be another such clip space. We have defined such an interface and call it the 'Clip Space API.'

6.1 Clip Space API

There are essentially three main components that interact in the API. The first is a clip application, the second is a clip space, and the third is a clip re-launcher. The clip application produces clips, and sends those clips to a clip space. The clip application can connect to any clip space that fits the API. It functions the same whether or not it is our LearnSpace Clip Space, or if it is the hypothetical table-top display clip space. It just produces and shares clips.

The clip space can do whatever it likes with the clips, and we have already given some examples of what it might do. If it wishes to re-launch a clip, however, it does so by communicating with the clip re-launcher. The clip re-launcher is a component that is responsible for interpreting the launch-action string to determine whether or not the clip can be re-launched on a specific device. The re-launcher is also then responsible for making the system call to re-launch the application. In Figure 43 we provide a diagram summarizing how these three components interact.

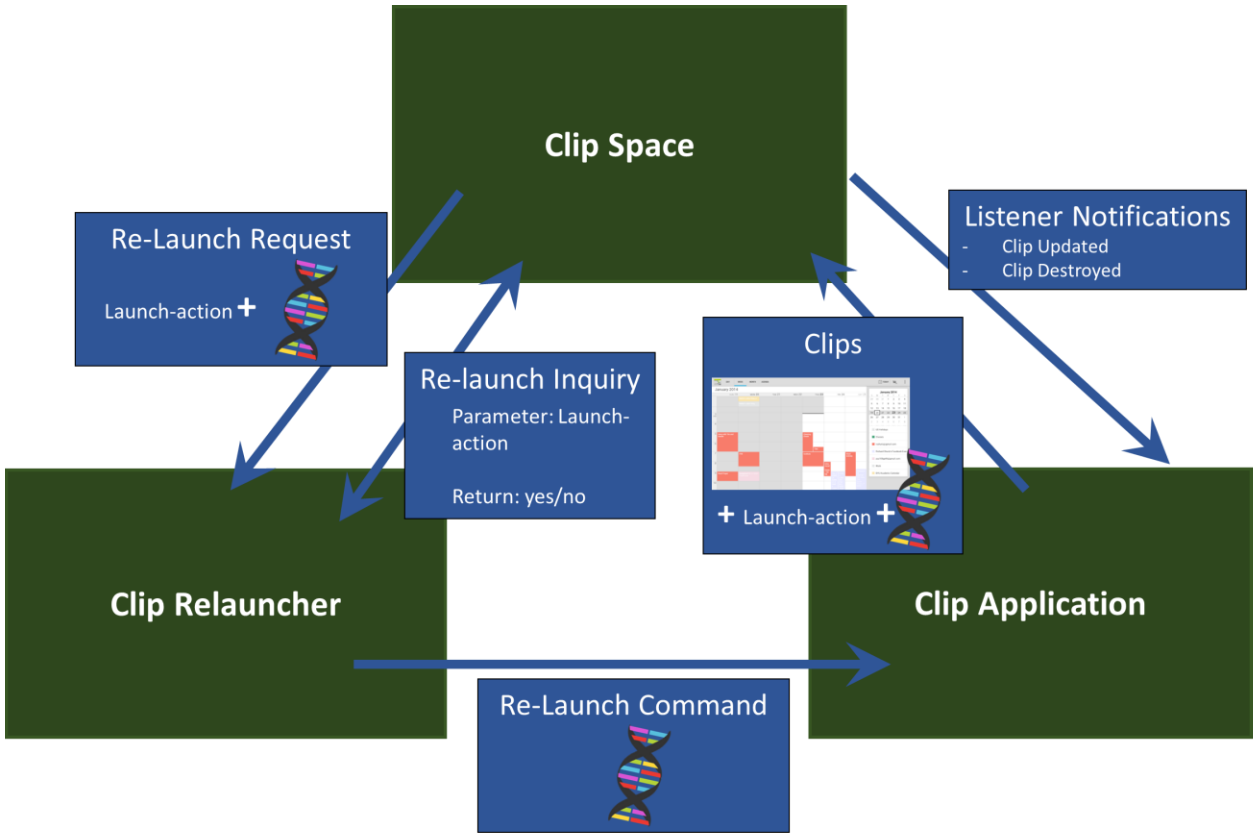


Figure 43 – A diagram showing how the three main components interact.

We provide briefly some of the details of how these pieces interact and the methods involved. Essentially, there are two areas of direct interaction in our API. There is the clip-application/clip-space interaction, and the clip-space/clip-launcher interaction. The re-launch command is dependent on the clip application's platform so we cannot define anything specific in that regard.

6.1.1 Clip Application & Clip Space Interactions

All interactions between a clip application and a clip space are initiated by the clip application. It chooses to connect to a clip space, share clips, etc. The clip space communicates back to the clip application only through listeners registered by the clip application. All of these methods are defined on the ClipSpace interface. We discuss these methods below.

6.1.1.1 Connecting

The first step in interacting with the clip space is to establish a connection. The clip application simply calls the connect() method on the clip space object. If the clip space needs information about where to connect (for example, what server to connect to), it is its own responsibility to prompt the user for that information. There are also methods to disconnect, register and unregister connection listeners, and query whether or not the clip space is already connected or in the process of connecting.

6.1.1.2 Working with Clips

When a clip application is connected to the clip space, it can then request for the space to create a new clip. The clip application provides the clip image, launch-action string, and application-DNA, and once the clip space has added the clip, it provides an asynchronous callback to return a unique identifier for the clip. The clip application uses the clip's ID for all further interaction with that clip.

From there, the clip application can update a specific clip. It identifies the clip by its ID, and then provides a new clip image (or part of a clip image by specifying the part of the image which has changed) and new application-DNA. Once again, there is an asynchronous callback to let the clip application know when the clip space has finished updating the clip. This functionality allows for continuous clip updates to create a live sharing experience. The ClipActivity base class can continuously update the same clip so that the clip on the shared display always shows the current state of the clip application.

There is a method to destroy a clip (when, for example, the clip application is done sharing). The clip application can make queries about whether a clip with a specific ID still exists within the clip space, or if a specific clip is done updating. Finally, there are methods to register/un-register clip listeners. The listeners are notified when a clip is destroyed or done updating.

6.1.2 Clip Space & Clip Re-Launcher Interactions

The clip re-launcher has just three methods (defined in the ClipApplicationLancher interface). The first is to query the re-launcher whether or not the clip can be re-launched. It takes the launch-action string as a parameter, and then returns whether or not it can be launched on the current platform and device.

The other two methods are to perform the re-launch of a clip. The first case is for a simple application re-launch, while the second is to re-launch and then continue sharing to a specific clip already in the clip space. For the latter, the clip space must provide the clip's ID and a reference to the clip space in addition to the launch-action string and application-DNA.

6.1.3 Classes and Interfaces in the API

Figure 44 below gives a quick overview of the interfaces and classes defined in the API. It is not a comprehensive description of the methods, but it does illustrate how the main components work together. The ClipSpace interface is where all of the methods are located described in section 6.1.1. The ClipApplicationLauncher contains the methods described in 6.1.2. Other classes and interfaces are defined to support these two main components.

Class or Interface	Methods

`ClipSpace`
`interface`
	`void connect()`
	`void createClip(ClipImage, ClipRelaunchData, ClipUpdateCallback)`
	`void updateClip(int clipId, ClipImage, ClipRelaunchData, Rectangle, ClipUpdateCallback)`
	`void addClipListener(int clipId, ClipListener)`
	`...`

`ClipApplicationLauncher`
`interface`
	`boolean canRelaunchClip(String launchActionString)`
	`void relaunchClip(String launchActionString, ClipRelaunchData)`
	`void relaunchClip(String, ClipRelaunchData, int clipId)`

`ClipRelaunchData`
`class`
	`String getLaunchActionString()`
	`byte[] getApplicationDna()`

`ClipImage`
`interface`
	`int[] getPixels()`
	`int getWidth()`
	`int getHeight()`
	`...`

`Rectangle`
`class`
	`int getLeft();`
	`int getRight();`
	`int getWidth();`
	`int getHeight();`

`ClipListener`
`interface`
	`void onClipUpdated(int clipId);`
	`void onClipDestroyed(int clipId);`

`ClipUpdateCallback` `interface`	`Void onClipDoneUpdating(int clipId);`

Figure 44 – A summary of classes and interfaces defined by the Clip Space API.

6.2 LearnSpace Clip Space

Our own Android Clip Application Framework is clip space agnostic. Figure 21 (back in Chapter 3) shows how the different components in our solution interact and is actually fairly similar to Figure 43 above. Figure 45 below, however, shows how our implemented system fits the Clip Space API.

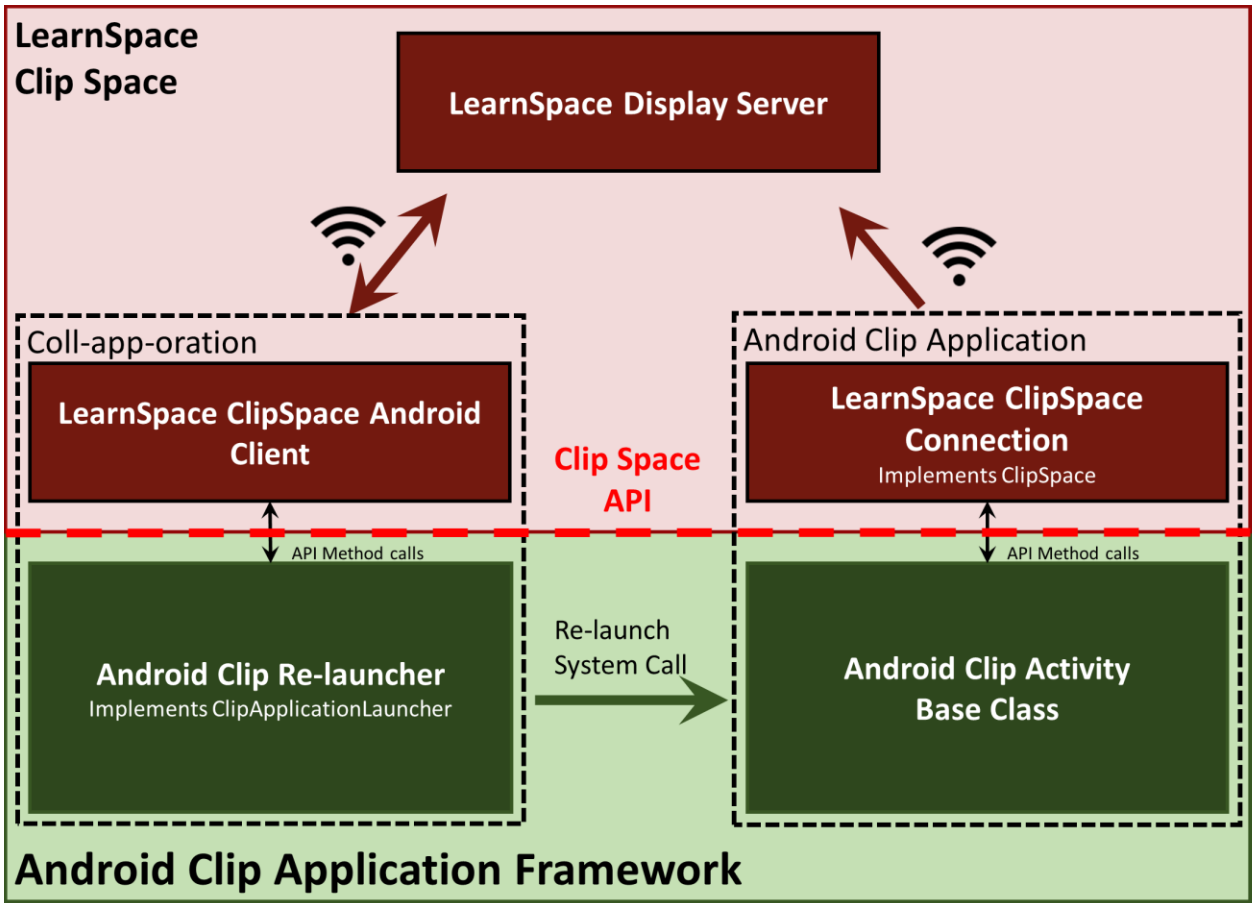
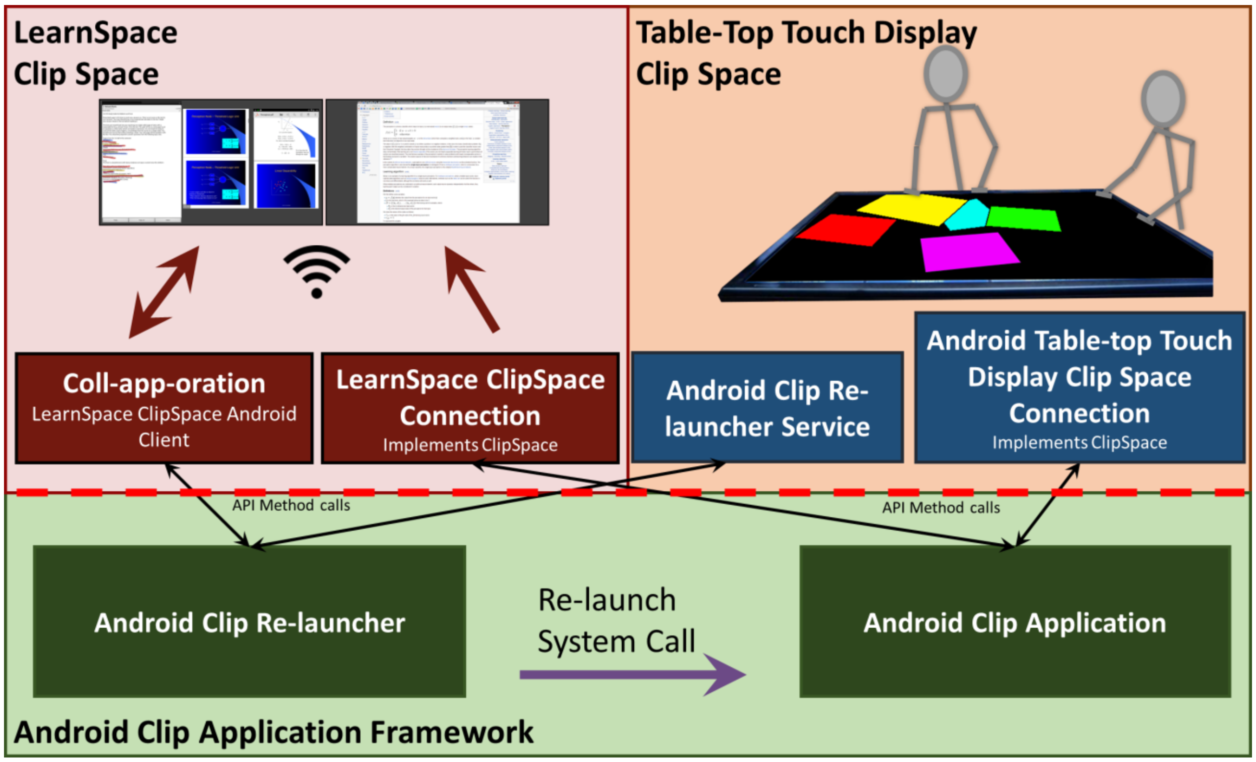

LearnSpace Clip Space
LearnSpace Display Server
Coll-app-oration
LearnSpace ClipSpace Android Client
Clip Space API
Android Clip Application
LearnSpace ClipSpace Connection
Implements ClipSpace
API Method calls
API Method calls
Android Clip Re-launcher
Implements ClipApplicationLauncher
Re-launch System Call
Android Clip Activity Base Class
Android Clip Application Framework

Figure 45 – A diagram showing how our LearnSpace Clip Space and our Android clip applications fit together with regards to the Clip Space API. The red line marks where the method calls are defined by the API.

The figure shows that the LearnSpace clip space is comprised of multiple components. The LearnSpaceClipSpaceConnection class, however, is what implements the ClipSpace interface that we have defined. This means that the ClipActivity base class does not worry about LearnSpace sheets, or data assets or anything else described earlier in the Chapter 3 ‘Display Server’ section. Rather, it just connects to a ClipSpace object, and then interacts with the clip space using the methods in the API. The LearnSpaceClipSpaceConnection class contains the code to establish a connection with the LearnSpace server, create and update sheets, and to deal with anything related to LearnSpace.

6.3 New Clip Spaces

Having fit the Android Clip Application Framework to the Clip Space API, it is now possible to switch out the LearnSpace/Coll-app-oration clip space for any other clip space. Figure 46 below shows one such possibility. The Android clip application is agnostic about what clip space it is connected to: either way, it still calls the same methods through the API. Likewise, either clip space can use the same code for the Android Clip Re-Launcher.



LearnSpace Clip Space
Table-Top Touch Display Clip Space
Coll-app-oration
LearnSpace ClipSpace Android Client
LearnSpace ClipSpace Connection
Implements ClipSpace
Android Clip Re-launcher Service
Android Table-top Touch Display Clip Space Connection
Implements ClipSpace
API Method calls
API Method calls
Android Clip Re-launcher
Re-launch System Call
Android Clip Application
Android Clip Application Framework

Figure 46 – Using the same code for our Android Clip Application Framework, we can connect to any type of clip space. The clip space implementation is interchangeable. Here, as an example, we show our LearnSpace Clip Space implementation as well as our hypothetical Table-top Touch Display Clip Space example.

6.3.1 Extending Schulte's Framework

To show the simplicity of creating a new clip space that would be compatible with clip applications, we describe what it would take to add clips to Schulte's collaborative applications.

The underlying concept in his work is that participants all have access to a shared ‘item store.’ An ‘item’ is essentially a content file (of any type of data) along with a set of tags to help identify the item. The item store is a ‘cloud’ repository of items. His Android application already includes an item store implementation that connects and synchronizes items with the cloud item store. Thus, to integrate clip applications into Schulte’s framework would be as simple as implementing the ClipSpace interface to take clips and turn them into ‘items.’ Other than that, we would use our Android Clip Re-launcher to add the option to re-launch clip items to Schulte’s Android application.

We expect the entire process would include the following changes.

- Create a ClipItem class
- Create an ItemClipSpace class that implements the ClipSpace interface
- Create a ClipItemView class (probably a sub-class of ImageItemView) that has UI option to re-launch
- Make slight modification to allow clip items to be involved in discussions along with existing item types (image, ink, text, etc)

With those simple changes, we are able to create an entirely new clip space without making any changes to the clip application framework or any clip applications. The clip space is left to decide what it does with the clips, and how it deals with them (for example, we do not enforce what format the images have to be in, or what network protocols are used). When it is time to re-launch them, it is able to use our existing implementation so it does not have to deal with the specifics of how to re-launch clip applications on Android.

6.3.2 Installing New Clip Spaces

In our current implementation, the ClipActivity class is not dependent on the LearnSpace clip space implementation. However, the LearnSpaceClipSpaceConnection class is only accessible by including it in the .jar file of our framework which developers use for making their own applications. (If this were on desktop Java, the clip space implementations could be loaded dynamically at runtime, but this is not an option on Android.) Thus, adding a new clip space implementation would actually require the addition of those class files to the .jar. This means that any clip application dependent on our framework would need to be re-compiled and reinstalled to be able to connect to the new clip space. This is, of course, not a very elegant solution for adding new clip spaces.

We have, however, designed a solution which can remove this dependency entirely. Rather than including the clip space implementation as part of the .jar file of the framework (which is included in every clip application), we would provide the clip space implementation as an Android 'Service.' For example, in our case, the LearnSpace clip space implementation would be installed as part of Coll-app-oration as an externally useable service. Now, when the clip activity is choosing to connect to a clip space, it would query the system to see what installed 'clip space services' there are. It would establish a connection with the chosen service, and then all communication would take place through the service's interface. We have not yet implemented this 'service' design, but we could fit in this service interface layer so that different clip spaces can be installed in the system without needing to change clip applications at all.

6.4 Summary

With the Clip Space API, we have created an interconnection. On one side of the interface, there are a huge number of possible clip applications. For very little effort, virtually any existing Android application can be modified to be a clip application. Any future Android application can also use the framework to become a clip application. On the other side, there are many potential clip spaces to use these clips as a basis for collaboration. Working with clips is essentially as simple as working with images. Any new clip application is automatically compatible to work within existing clip spaces, and any new clip space can make use of all existing clip applications. The Clip Space API ensures that the connection between the two is cleanly supported.

Chapter 7 – Extending to Other Platforms

Just as the concept of clips should not be limited to a single collaborative space, neither should it be limited to a single application platform. In our work, we have only implemented a clip application framework for Android. We expect, however, that clip application frameworks could be designed and built for many other platforms as well.

For example, we could create a Java AWT clip application framework. Just as we designed the ClipActivity class to replace the Activity base class on Android, we could create a ClipFrame class to replace the JFrame base class used in AWT. It would be responsible for creating clips, intercepting windowing redraw calls, and drawing and sending clip updates in the background. We would also need to design a launch-action string. It might look something like "AwtLaunchAction::<file path to jar>::<main class>::deviceID". We would then create an 'AWT Clip Re-launcher' that implements the clip re-launcher component of our Clip Space API. Overall, we would follow the same pattern for the desktop Java framework as we used in our Android framework.

Below in Figure 47 we show how this hypothetical desktop Java AWT clip application framework would fit in with our existing LearnSpace clip space. Everything with the Android clip application framework and its interface with the clip space remain the same. In order to get the Java AWT clip applications fully involved in the clip space, however, we need to implement LearnSpace clip space components that can run on the new platform.

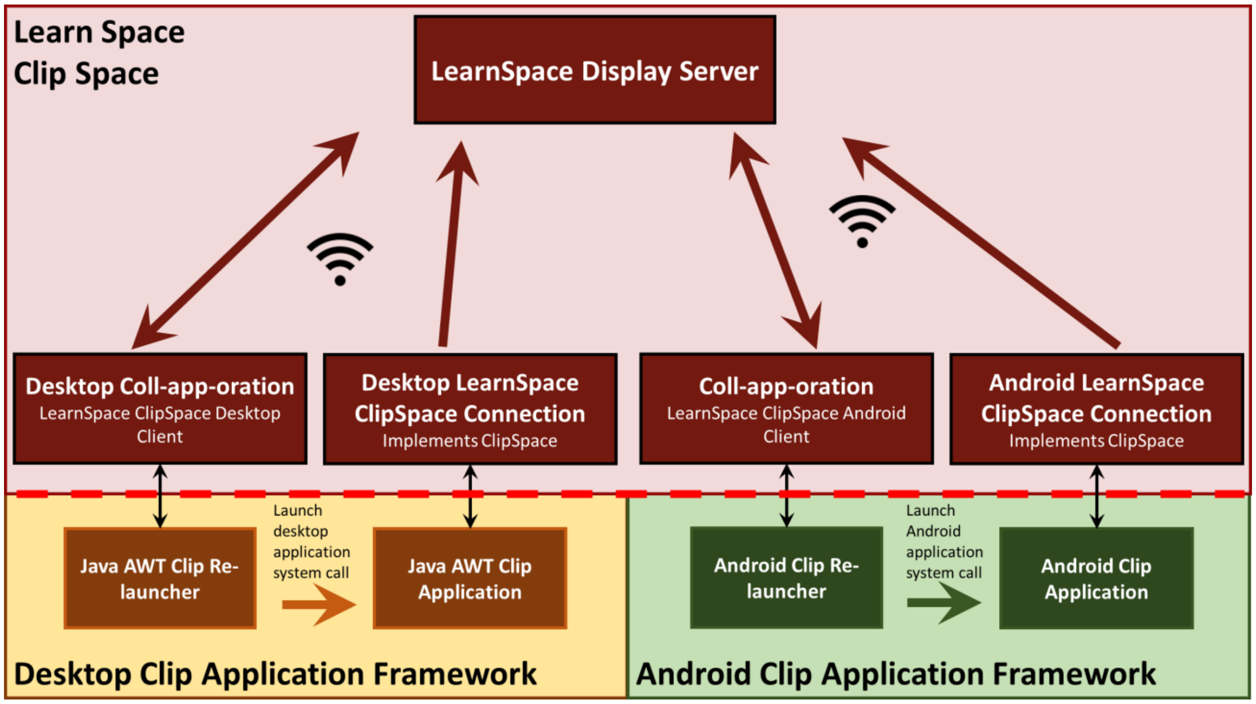

Learn Space
Clip Space
LearnSpace Display Server
Desktop Coll-app-oration
LearnSpace ClipSpace Desktop Client
Desktop LearnSpace ClipSpace Connection
Implements ClipSpace
Coll-app-oration
LearnSpace ClipSpace Android Client
Android LearnSpace ClipSpace Connection
Implements ClipSpace
Java AWT Clip Re-launcher
Launch desktop application system call
Java AWT Clip Application
Android Clip Re-launcher
Launch Android application system call
Android Clip Application
Desktop Clip Application Framework
Android Clip Application Framework

Figure 47 – The same clip space can connect to clip applications running on different platforms. This shows the hypothetical case where we develop a desktop clip application framework along with a desktop Coll-app-oration client.

Figure 47 illustrates a cumbersome aspect of including multiple platforms. A clip space must provide two components for each platform. The first is the 'clip space connection' component, which takes clips from the clip application and adds them to the clip space in whatever format the clip space is expecting (creating sheets in our LearnSpace implementation, or creating items for the hypothetical implementation for Schulte's space). The second is the 'clip space client application.' This component is necessary for clip re-launching. If there is no way of getting clip info from the clip space back to the device, then there is no way to re-launch. Thus, there must be an application running on the device that can connect to the clip space.

Both components are necessary, and, for the most part, they cannot be shared across different platforms. This imposes a relatively high cost for implementing clip spaces that can work with clip applications on multiple platforms.

7.1 Add a network API

One option to try to eliminate the need for each clip space to implement a new client and connection for each platform is to extend the Clip Space API across device boundaries by developing a 'Clip Space Network API.' The network API would primarily define some HTTP GETs and POSTs to perform the operations already defined for the Clip Space API (see Chapter 6 for a summary of these interactions). Any server that implements those methods would be considered a 'clip space server.' With this network API, the entire interaction with the clip space could take place through network calls rather than method calls on local objects.

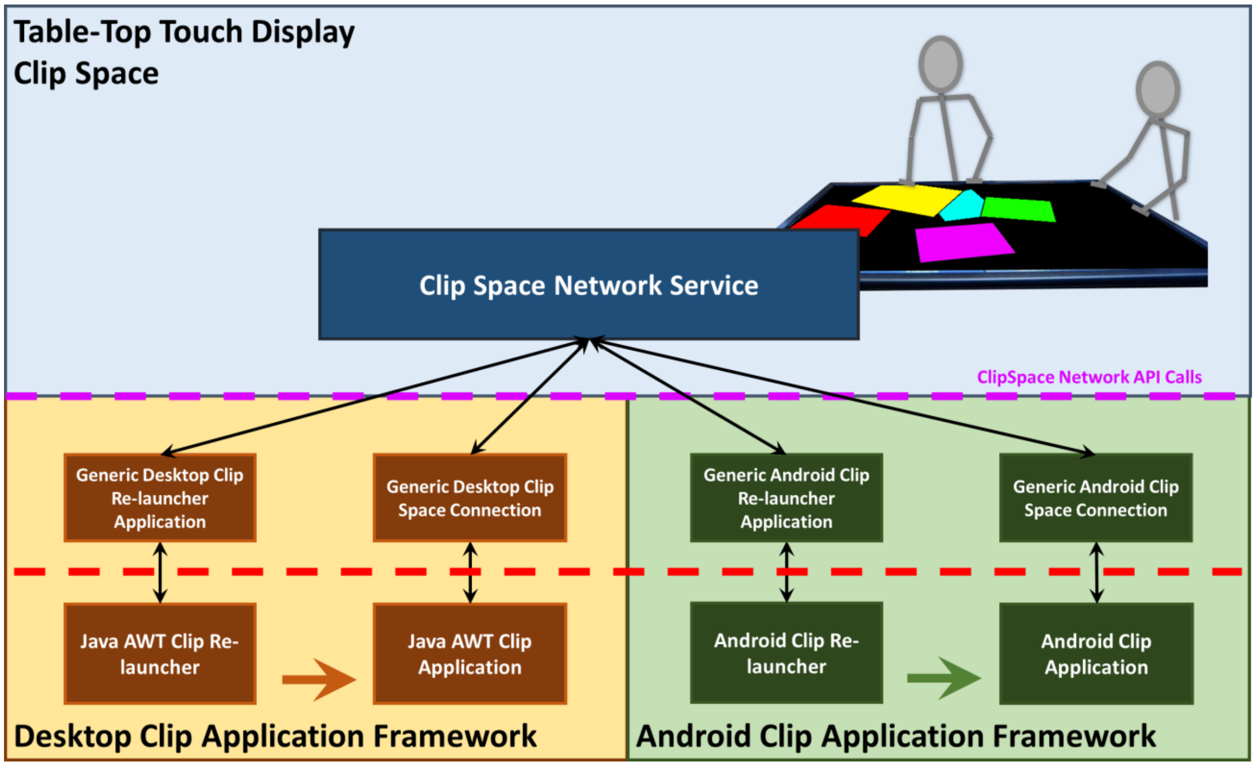


Table-Top Touch Display
Clip Space
Clip Space Network Service
ClipSpace Network API Calls
Generic Desktop Clip Re-launcher Application
Generic Desktop Clip Space Connection
Generic Android Clip Re-launcher Application
Generic Android Clip Space Connection
Java AWT Clip Re-launcher
Java AWT Clip Application
Android Clip Re-launcher
Android Clip Application
Desktop Clip Application Framework
Android Clip Application Framework

Figure 48 – A diagram showing how a well-defined network API could simplify the problem of developing a clip space that can interact with multiple platforms.

It would require us to develop a generic clip space connection that can connect (probably by IP address and port) to a clip space server. It would also require a generic clip space client application that can connect in the same fashion in order to download clip re-launch info to the local device. Figure 48 shows how these could remove the need for specific clip space components for each platform by including the generic components in their respective clip application frameworks.

Compared to the system in Figure 47, this architecture adds a little more work in building out a clip application framework for a new platform, but immensely simplifies the work required in implementing a clip space that can work with clips from multiple platforms.

If all clip spaces must operate through the Clip Space Network API, however, we do lose some flexibility in how we implement a clip space. For example, our LearnSpace Clip Space was able to take clips and immediately “convert” them to LearnSpace sheets. It then let the LearnSpace Client API take care of all the networking. In our hypothetical extension of Schulte’s work, we were able to simply treat clips as items and use his classes to take care of transferring them over the network. Conveniently, we need not eliminate these existing solutions by adding in the support for a network API.

Figure 49 below shows how these components can all work together without conflict. Our clip application (ClipActivity) and clip re-launcher still operate entirely though our existing Clip Space API. Just as before, the clip application can connect to the LearnSpace Clip Space Connection Service through this API. Now, however, it also has the option of connecting to the generic Android Clip Space Connection Service. This generic connection can connect to any clip space server using the Clip Space Network API. Likewise, both Coll-app-oration and the generic re-launcher client application can both access the Android Clip Re-launcher through the Clip Space API.

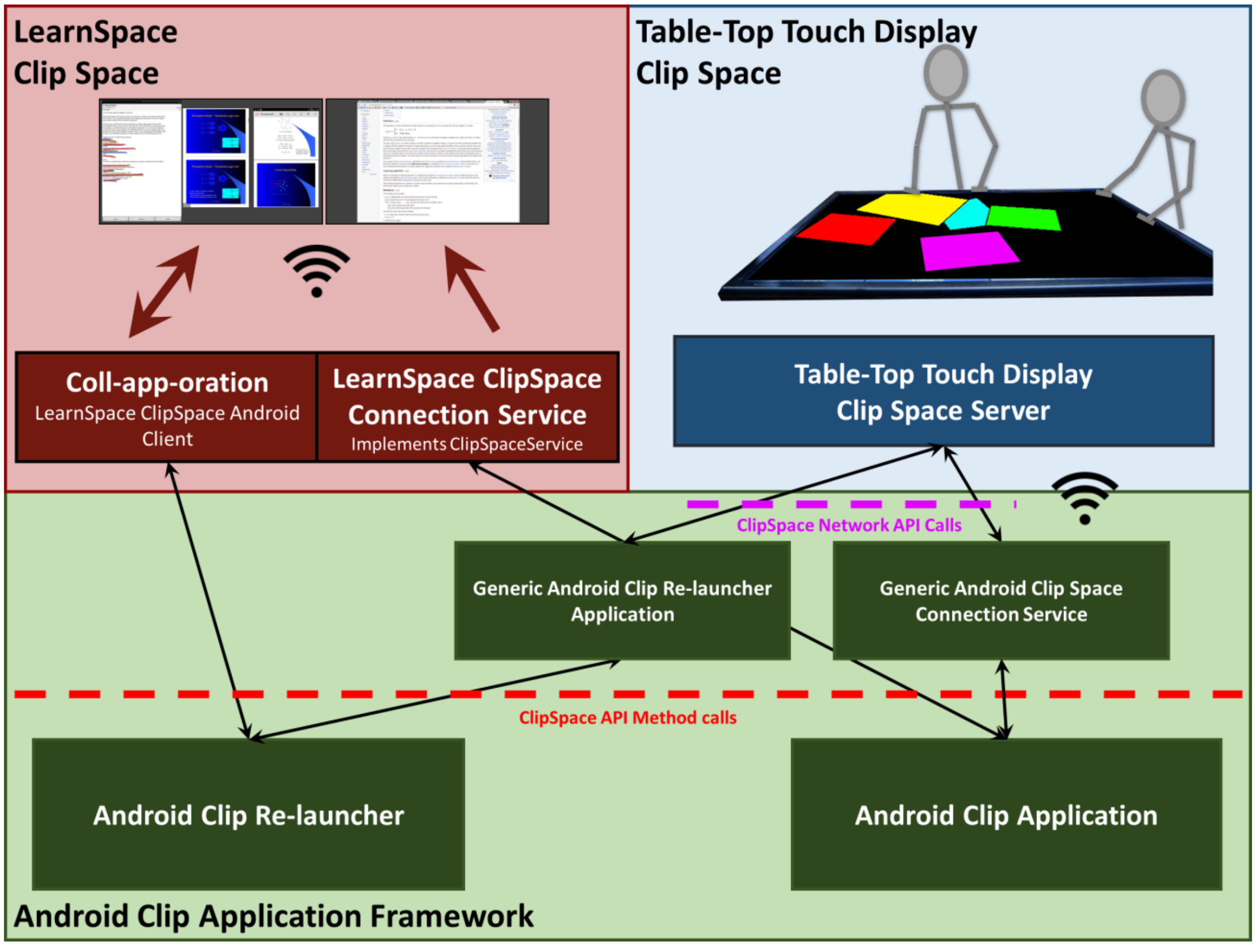

LearnSpace Clip Space
Table-Top Touch Display Clip Space
Coll-app-oration
LearnSpace ClipSpace Android Client
LearnSpace ClipSpace Connection Service
Implements ClipSpaceService
Table-Top Touch Display Clip Space Server
ClipSpace Network API Calls
Generic Android Clip Re-launcher Application
Generic Android Clip Space Connection Service
ClipSpace API Method calls
Android Clip Re-launcher
Android Clip Application
Android Clip Application Framework

Figure 49 – This diagram shows how the generic Clip Space Connection and Clip Re-launcher Client components can fit within the existing solution. All interactions with clip applications to the clip space and from the clip space to the re-launcher still take place through the Clip Space API.

The ultimate benefit of this solution is that clip spaces have the option of building a single clip space server rather than connection and client components for each application platform.

7.2 Summary

In summary, implementing a clip application framework on a new platform would require the following components.

- A new base class (following the pattern of ClipActivity) to replace the base application class for that platform

- A well-defined launch-action string for the platform and an accompanying clip re-launcher in compliance with the Clip Space API
- A generic clip space connection that implements the ClipSpace interface and communicates with a clip space server via the Clip Space Network API
- A generic clip space client application that can connect to a clip space server and receive a list of clips it can re-launch

We have not done these for any platform other than Android, and we have not built the generic connection and client components because none of our own work requires a clip space server. We present this design to show that our Clip Space API and clip applications can exist and interact cleanly on platforms other than Android.

Chapter 8 – Summary

In Chapter 1, we described the needs of collaborators in small-group meetings. Designers, students, and other professionals want to share applications from their own mobile devices in meaningful ways. They want to be able to share, consider, critique, and explore each other's ideas as a group. They want to be able to take the applications/ideas with them away from the meeting so they can continue working with them. Finally, users want to be able work across multiple devices and platforms and in multiple collaborative spaces.

Thus, we need a solution that allows visual application sharing to a large display. Applications should be able to be downloaded and re-launched at a later point. This collaborative system should support organizing and annotating shared applications as tools to support the users' discussion. As such a system is only as useful as the applications which can be shared in it, it is necessary that the cost to develop participating applications be small.

To meet the needs of the collaborative space, we designed and built an Android application called Coll-app-oration using the LearnSpace server and client API. Our system allows multiple users to connect and share simultaneously from their Android mobile devices. Coll-app-oration provides tools to organize and annotate the shared applications as they are shared on the display. It also allows users to download any shared application clip to their own devices. Thus, it meets all of the needs described in Chapter 1 of a good collaborative space.

To allow application re-launching, we designed launch-action strings and application-DNA which we described in detail in Chapter 4. Together, they provide an extremely flexible yet simple mechanism to allow applications to be re-launched and restored to their prior state. In order to easily create applications that can share themselves visually and use this re-launch ability, we designed and build the Android Clip Application Framework (descried in Chapter 5).

It takes care of the visual sharing aspect of an application automatically, helps developers implement application-DNA, and largely adheres to the existing Android Application Framework. By converting a number of existing open-source Android applications to fit the framework, we have shown that the app conversion is simple and works for a wide variety of applications.

Finally, we have developed the Clip Space API which we described in detail in Chapter 6. It enables developers to create new clip spaces which can leverage clip applications for collaboration in new ways. We have also described, in Chapter 7, the process of extending our solution to support applications on platforms other than Android. Because all applications are shared simply as clips, there is no difficulty in having applications from multiple platforms all shared together in the same clip space.

There is, of course, much that could be done to build on our solution. Two obvious choices would be to build clip application frameworks for other platforms and to build or convert more clip applications. This would certainly increase the functionality of our system. Another area to build out the current system would be to investigate rendering applications at different resolutions as they are shared to the display. In our system, the application clips shared to the display are pixel-for-pixel matches with the Android device's own display. However, the Android Application Framework is designed to support applications running on a wide variety of devices with different screen sizes, resolutions, and pixel densities. It would be worth investigating leveraging this functionality to render applications differently to better fit the screen real-estate of the shared display when producing clip images.

In terms of future research, it may be interesting to look into how we might be able to extend the usefulness of clips. Clips represent a new paradigm in application sharing. For the most part, it is as simple as sharing an image, but it also has the ability to “be turned back into” an application. The underlying question is, “what can we do with the clip that might be meaningful when it is time to re-launch?” For example, can we use image annotations on the clip to somehow provide meaningful input to an application when it is re-launched?

Another area of future research would be to explore how other clip spaces can make use of clips in new and novel ways. Is the real power of clips in the ability to work with many at the same time? If so, perhaps there is a need to develop more advanced organization methods. What existing systems could benefit from using clips (for example, Schulte’s collaborative applications)? Or, how could clips be useful for a single user? The exploration of using clips in new situations could unlock new methods of interaction.

Bibliography

1. Adobe Systems Incorperated. Adobe Reader mobile app. 2014. http://www.adobe.com/products/reader-mobile.html.

2. Apple Inc. iOS: Use AirPlay Mirroring. 2014. http://support.apple.com/kb/ht5209.

3. Arthur, R. and Olsen, D.R. XICE windowing toolkit. *ACM Transactions on Computer-Human Interaction 18*, 3 (2011), 1–46.

4. Bardram, J.E., Gueddana, S., Houben, S., and Nielsen, S. ReticularSpaces : Activity-Based Computing Support for Physically Distributed and Collaborative Smart Spaces. *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*, ACM Press (2012), 2845.

5. Biehl, J.T., Baker, W.T., Bailey, B.P., Tan, D.S., Inkpen, K.M., and Czerwinski, M. Impromptu: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development. *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*, (2008), 939–948.

6. Divitini, M., Farshchian, B.A., and Samset, H. UbiCollab : Collaboration support for mobile users. *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04*, (2004), 1191–1195.

7. Evernote Corporation. Evernote. 2014. http://evernote.com/.

8. Fallman, D., Kruzeniski, M., and Andersson, M. Designing for a collaborative industrial environment: the case of the ABB Powerwall. *2005 conference on Designing*, (2005), 2–16.

9. Finke, M., Kaviani, N., Wang, I., Tsao, V., Fels, S., and Lea, R. Investigating distributed user interfaces across interactive large displays and mobile devices. *Proceedings of the International Conference on Advanced Visual Interfaces - AVI '10*, ACM Press (2010), 413.

10. Greenberg, S., Boyle, M., and Laberge, J. PDAs and Shared Public Displays: Making Personal Information Public, and Public Information Personal. *Personal Technologies 3*, March (1999), 54–64.

11. Jeon, S., Hwang, J., Kim, G.J., and Billinghurst, M. Interaction with large ubiquitous displays using camera-equipped mobile phones. *Personal and Ubiquitous Computing 14*, 2 (2009), 83–94.

12. Johanson, B., Fox, A., and Winograd, T. The Interactive Workspaces Project : Experiences with Pervasive Computing Magazine Special Issue on Systems. (2002), 1–17.

13. Kolko, J. Wicked Problems: Problems Worth Solving. 2012. https://www.wickedproblems.com/.

14. Liu, C. and Kao, L. Handheld Devices with Large Shared Display Groupware: Tools to Facilitate Group Communication in One-to-One Collaborative Learning Activities. *IEEE*

International Workshop on Wireless and Mobile Technologies in Education (WMTE'05), (2005), 128–135.

15. Liu, Z. Lacome: a cross-platform multi-user collaboration system for a shared large display. 2007. https://circle-prod.library.ubc.ca/handle/2429/378.

16. Lutz, R., Schäfer, S., and Diehl, S. Using mobile devices for collaborative requirements engineering. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, ACM Press (2012), 298.

17. McGrath, W., Bowman, B., McCallum, D., Hincapié-Ramos, J.D., Elmqvist, N., and Irani, P. Branch-explore-merge. *Proceedings of the 2012 ACM international conference on Interactive tabletops and surfaces - ITS '12*, ACM Press (2012), 235.

18. Negulescu, M. and Li, Y. Open project. *Proceedings of the 26th annual ACM symposium on User interface software and technology - UIST '13*, ACM Press (2013), 281–290.

19. Olsen, D.R. Interacting in chaos. *Interactions 6*, 1999, 42–54. http://dl.acm.org/ft_gateway.cfm?id=312720&type=html.

20. Olson, G., Olson, J., Carter, M., and Storrosten, M. Small Group Design Meetings: An Analysis of Collaboration. *Human-Computer Interaction 7*, 4 (1992), 347–374.

21. Paek, T., Agrawala, M., Basu, S., et al. Toward universal mobile interaction for shared displays. *Proceedings of the 2004 ACM conference on Computer supported cooperative work - CSCW '04*, ACM Press (2004), 266–269.

22. Richardson, T., Stafford-Fraser, Q., Wood, K.R., and Hopper, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (1998), 33–38.

23. Schulte, D.L. Interactive Techniques between Collaborative Handheld Devices and Wall Displays. *Brigham Young University*, August (2013).

24. Seifert, J., Simeone, A., Schmidt, D., et al. MobiSurf. *Proceedings of the 2012 ACM international conference on Interactive tabletops and surfaces - ITS '12*, ACM Press (2012), 51.

25. The BBQDroid Team. BBQScreen. 2013. http://screen.bbqdroid.org/.

26. Vartiainen, P., Chande, S., and Rämö, K. Mobile Visual Interaction Enhancing local communication and collaboration with visual interactions. *Proceedings of the 4th international conference on Mobile and ubiquitous multimedia - MUM '06*, ACM Press (2006), 4–es.

27. Wallace, J.R., Scott, S.D., and MacGregor, C.G. Collaborative sensemaking on a digital tabletop and personal tablets. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, ACM Press (2013), 3345.

28. Wi-Fi Alliance. Wi-Fi CERTIFIED Miracast^{TM}. 2014. http://www.wi-fi.org/wi-fi-certified-miracast?

29. LearnSpace. http://www.pixelture.com/info/education.

30. Jelly Bean | Android Developers. http://developer.android.com/about/versions/jelly-bean.html.

31. Managing the Activity Lifecycle. http://developer.android.com/training/basics/activity-lifecycle/index.html.

32. Gospel Library. https://play.google.com/store/apps/details?id=org.lds.ldssa.

33. 920 Text Editor. https://play.google.com/store/apps/details?id=com.jecelyin.editor.

34. Zirco Browser. https://play.google.com/store/apps/details?id=org.zirco.

35. com.robert.maps. https://f-droid.org/wiki/page/com.robert.maps.

36. Google Calendar. https://play.google.com/store/apps/details?id=com.google.android.calendar.

37. Google Drive. https://play.google.com/store/apps/details?id=com.google.android.apps.docs.

38. arity-calculator. https://code.google.com/p/arity-calculator/.

39. ActionBarSherlock. http://actionbarsherlock.com/.

40. WebView. http://developer.android.com/reference/android/webkit/WebView.html.

Appendix A – Application Conversion Instructions

Add the following app permissions to the manifest

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

Add the following Service to the manifest

```
<service android:name="com.pixelture.wipix.SessionService" />
```

If you app doesn't already use it, add the AndroidSupportLibrary

For every Activity that you want to turn into a ClipActivity

- If necessary, create a new base `Clip_<?>_Activity` to extend
    - For example, if you're activity extends `Activity`, then you don't need to do anything, use `ClipActivity` from our package (or use `ClipListActivity` instead of `ListActivity`)
    - But, if, for example, your activity extends `SpecialActionBarActivity` (which extends `Activity`), then you'll want to create a new class called `ClipSpecialActionBarActivity`.
        - To do so, copy and paste the code from `ClipActivity` into a new class, and instead of having it subclass Activity, have it subclass `SpecialActionBarActivity`
- Change your activity to subclass the desired base `Clip_<?>_Activity` class
- Modify `onCreate(Bundle savedInstanceState)` to be instead `onCreate(Bundle savedInstanceState, AppDnaMap savedAppDNA)`
    - Remove the call to `super.onCreate(Bundle)`
- Look at your activity declaration in AndroidManifext.xml
    - Ensure that `android:exported="true"`
    - If you have any intent-filters, this will default to true, so you're okay
    - If you have no intent-filters, you'll need to add this xml attribute yourself
- Decide whether or not your Activity should be launch-able from one device to another. This really only makes sense if your application doesn't use any locally saved data.
    - Implement `canLaunchOnOtherDevice()` to return whether or not it can be launched on a different device
- If you Activity overrides `onNewIntent(Intent)`, change it to `onNewLocalIntent` and
    - Remove the call to `super.onNewIntent(Intent)` if it is there
- Design and implement saving application-DNA and re-launching the activity from the saved application-DNA
    - Decide whether or not you want the activity to automatically save and restore the Intent data for you (see documentation)
        - If not, override `autoSaveAndRestoreIntent()` to return `false`
    - Implement your solution by implementing `saveApplicationDNA()` and using the `AppDnaMap` parameter in `onCreate(Bundle, AppDnaMap)`

    - Consider the ways that the activity might not be able to be re-launched from the application-DNA (for example, if the DNA is referencing an item that has since been deleted). If possible, provide the user with some feedback or explanation and stop the activity from fully launching

- Ensure your activity calls through the super implementation on the following methods:
    - `onBackPressed()`
    - `onSaveInstanceState()`
    - `onCreateOptionsMenu()`
    - `onPrepareOptionsMenu()`
    - `onOptionsItemSelected()`
    - `onMenuItemSelected()`
    - `setContentView(int)`
    - `setContentView(View)`
    - `setContentView(View, LayoutParams)`
    - `addContentView(View, LayoutParams)`
    - `onCreateDialog()`

    Most applications will either already be calling through the super implementation, or not overriding these methods anyway

Test

- Ensure that screen sharing works
    - Ensure that rotating the screen doesn't stop screen share
- Ensure that snapshot works
- Ensure that others can download a snapshot of the app
- Ensure the restoring on the same device works
- If you allow re-launching on different devices, then test that extensively as well