Gerard Martin

Peer-to-Peer Video Sharing with Android

Computer Science Tripos, Part II

Trinity Hall

13th May 2010

Proforma

Name:	**Gerard Martin**
College:	**Trinity Hall**
Project Title:	**Peer-to-Peer Video Sharing with Android**
Examination:	**Computer Science Tripos, Part II, June 2010**
Word Count:	**11,646**
Project Originator:	**Dr Jon Crowcroft**
Supervisor:	**Dr Jon Crowcroft**

Original Aims of the Project

The design and implementation of a mobile application that sends video to phones in the immediate vicinity over a peer-to-peer network using local data transmission protocols like Wi-Fi and Bluetooth. The video stream should be tagged with a number of interests, and any phones within range running the application should be able to receive the video if they have declared a mutual interest with the video. The video should be broadcast live to other phones in the network as it is still being recorded.

Work Completed

The project has been successful as per the success criteria originally set out. A user can specify the particular interests they want to apply to a video stream and start recording it. This will broadcast the video throughout the ad-hoc network of phones. Other phones with those interests registered can then select that particular broadcast to view while it is still being broadcast live. The user is presented with a selection of relevant video broadcasts they can watch.

Special Difficulties

None.

Declaration of Originality

I Gerard Martin of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Table of Contents

List of Figures

Code Listings

Chapter 1

Introduction

My project utilises the architecture of the Haggle Project[9] to broadcast live video from an Android mobile phone to other interested users within range. Broadcasters can tag their videos with keywords, and viewers who subscribe to those tags will receive the video. In this section, I will discuss some of the motivations for the project, as well as look at some similar research in the area.

1.1 Motivation for the project

Mobile phone usage has been increasing constantly over the past few years, especially the usage of smartphones—newer models having more advanced features, like wireless connectivity and cameras capable of recording good quality video.

Recording and sharing video with friends (and complete strangers) is becoming more and more common, thanks both to these new features becoming more ubiquitous in mobile phones, and social networks like Youtube and Facebook giving people the platform to share their video clips with each other easily. This is made easier by tagging videos, which makes them more easily found by an interested party.

Mobile phones are perfect for recording these videos that are being shared, but the process of actually sharing them involves more than just one step. The video needs to be saved to a desktop or laptop computer via a USB cable or memory card, then it needs to be uploaded to a video sharing web site like Youtube. In the middle of recording a live event, this just isn't possible.

My project is a mobile application that records video and broadcasts it to other interested application users, using both ad-hoc Wi-Fi and Bluetooth to transfer video locally from phone to phone. The motivation for this is that mobile phones are by nature intermittently connected to the Internet. When it is available, however, it is in the form of the mobile data network or Wi-Fi access points.

The mobile data network includes both the original GPRS network and the faster 3G network. The GPRS network is unusable for transfer of soft real-time data like video, due to its low speed and latency. The 3G network is usable, but the large amounts of data inherent in video streaming and the possibility of users moving cells will lead to a congested and strained network, and there is also a relatively high financial cost to upload to it. Publicly accessible Wi-Fi access points, when they can be found, often also have a large associated financial cost, which isn't cost-efficient for small incremental uploads. These are also motivations for utilising a local peer-to-peer network for my project.

There are also other motivations for restricting the scope of the application to local, peer-to-peer connections. There are a number of interesting use cases for streaming video to mobile phones in the vicinity. For instance, a number of application users could be attending a rock concert, where one is a front-line soldier and has managed to get to the front row, while the others can barely see the stage. The person at the front could stream the video from his mobile phone to the users further back over ad-hoc Wi-Fi, for instance. What's more, those users could then forward that video stream to users even further back (and out of range of the initial broadcaster).

Another possible use case is at a sports event, where a user could broadcast an alternative angle of, say, a baseball pitch to users from the other side, providing a second view of the event. This could be facilitated by intermediate users forwarding the video stream on to the interested user.

1.2 Related work

Personally, this project gives me a great chance to look at computer science related to the formation of temporal, ad-hoc networks, the patterns of nodes being connected (or disconnected) in these networks and how data can be forwarded out to all of these nodes.

The peer-to-peer framework I am using in my application is the Haggle Project[9]. It is a peer-to-peer architecture that forwards data objects from the originating device to other devices based on an interest system—if a potential receiver is listed as interested in the attributes of a particular data object, then that data object will be forwarded to the receiver. A *data object* is defined as a packet of some data (e.g. a video file) and metadata including the interests of the data object. In this way, a data object can be broadcast out from the originating device by propagating the data objects through intermediate devices to interested parties. Every device, or Haggle node, keeps state of all of the other nodes it is connected to, allowing data objects to be transmitted between them.

Haggle uses a number of algorithms to propagate these data objects. The default behaviour is to simply forward data objects on to any encountered nodes, ensuring that within the *interest community* of a certain tag (the set of devices that are interested in that particular tag), data objects are epidemically forwarded. Using only this algorithm, the entire interest community would need to be connected at some point, so Haggle also uses *delegate forwarding*, which transmits a data object to a (possibly better connected) device that isn't interested in it to further the dissemination of the data object to other interested devices, predicting which devices will be best at this based on their neighbour history in the disconnected ad-hoc network.

The application broadcasts video to multiple interested users over a peer-to-peer network. Similar ideas and motivations for video broadcasting were put forward by Microsoft Research in Mobicast[8]. This is similar to my system in that it allows the user to capture video using their mobile phone, but it also aims to take multiple streams from the same location and stitch them together in the cloud, meaning the Internet is an integral part of this system. My project is aiming to have less sophisticated features than Mobicast, but to perform these without needing to use the cloud.

The paradigm used in the architecture is that of the publish/subscribe model[4]. This loosely coupled model of data transmission suits mobile applications more than static point-to-point links, which would be difficult to maintain in a mobile network where user churn is much larger than wired networks (thanks to factors such as portability and limited battery life). The publish/subscribe model also works well with the tagging mechanism of the project, as each video tag can be used as a subject for publishing and subscribing. It is a much more scalable paradigm to use with my application, giving more scope for a greater number of users in the local area.

Chapter 2

Preparation

This section is concerned with showing the steps taken before beginning the project to ensure the implementation phase went off without any issues. The initial requirements are discussed, and the refinement of the original proposal is laid out.

2.1 Background research

2.1.1 Data transmission framework

The original project proposal[1] mentions the Haggle Project as having a "nice infrastructure" for a project of this kind, though I needed to ensure that there were no other alternative frameworks I could have used that would have been better suited to the application.

In the current model of the application for a single video stream, there is one broadcaster from where the live video originates, and there are a number of viewers to that video, many of whom will also be forwarding the broadcasted data on to other viewers. One can draw parallels with the paradigm of BitTorrent[3], where data is split into discrete chunks and is distributed from an initial *seeder* (the broadcaster), who then propagates these chunks on to waiting *leechers* (the viewers). Leechers can also forward particular chunks that they have already received on to other leechers still waiting to receive them. This design would work very well over a connectionless, ever-changing network like the movement of mobile phones in a local area. For this reason, I thought BitTorrent might be a good framework for the application.

[1]See Appendix A.

BitTorrent generally works by having a *tracker* that the seeder and all leechers make and maintain contact with. This tracker is a web server that has a list of contact information for peers so that everyone can connect to each other to download chunks of the file. This presents a problem to our mobile application—how can we deal with an ever-present tracker? We need to assume that this won't always be available in a mobile environment.

This is somewhat addressed in using a *Distributed Hash Table*, which is a decentralised and scalable way of replacing the constant connection to the tracker required with normal BitTorrent operation. DHT uses a table of $(key, value)$ pairs, where any node can efficiently retrieve the associated value with the key. Maintenance of the mappings is delegated to all nodes, meaning the system is robust to user churn. In cases of high user churn however, DHT suffers from high maintenance cost to maintain an orthogonal set of neighbour nodes.

The issue of requiring a tracker is also solved by the trackerless torrent system[5]. This idea involves utilising protocols based on random walks over the nodes in the system. Performing an unbiased random walk of sufficient length and selecting the last node in the walk will select each node with probability proportional to the degree. It allows nodes to communicate without ever needing to contact a central tracker.

These implementations would be well suited to my application in theory, however I have decided to use the Haggle Project instead. Firstly, the BitTorrent codebase would need to be implemented for my chosen platform, as well as one of the trackerless BitTorrent schemes I have looked at. These would be complex to implement, especially on a mobile platform. With limited time to complete the project, I would need to dedicate massive amounts of time on research and testing for the underlying implementation, and the other areas of the project would suffer as a result. Secondly, the Haggle Project already implements similar semantics in a different way, and also provides an API for my chosen platform. To implement one of the other frameworks would be to re-invent the wheel.

Also, because of the relative immaturity of the Google Android platform, Java methods to access Bluetooth are only supported in the 2.0 release of the platform, which currently doesn't include support for video recording in the open source repository. Furthermore, access to ad-hoc Wi-Fi doesn't have any implemented Java methods at all, meaning my application would need to implement a Java shim on the undocumented native C++ code, further increasing complexity. Using the Haggle Project on Android would avoid these issues.

To further refine the original project proposal, one idea was that "subscriptions might be distributed using the cellular net". This is an interesting idea, and would maximise the resources available for transmitting video. However, the Haggle Project uses the data object protocol to distribute *all* information about nodes around the network. As data on subscriptions only needs to be transmitted to mobiles in the vicinity, and we want to keep financial costs down, it makes sense to distribute metadata on the Haggle network.

2.1.2 Hardware platform

I have decided to use Google Android as the hardware platform for my application. Google Android is an open-source platform for mobile development that mainly uses Java. It freely provides an SDK that allows for easy application development inside the Eclipse IDE. This fits well with my prior experience, as Java is my strongest object-oriented programming language and I have been writing it using the Eclipse environment.

As mentioned in section 2.1.1, my chosen framework provides an API for Android, allowing flexibility in my code and meaning less boilerplate code needs to be written. I have also had previous experience coding with the Android SDK outside of the Tripos course. This means I have a head start in understanding the general coding style of Android, the terms used and the way applications are put together.

Another advantage in using Google Android is the detailed documentation available for its SDK. Good documentation that explains the semantics of its methods leads to a greater understanding of the platform and a tendency to write better code. Because of the popularity[10] of the platform, any gaps in the documentation are filled in by readily-available guidance.

One of the disadvantages of using Google Android for this project is the fact that the platform itself is still relatively young. There have been constant updates to the platform with newer features and different semantics, and there are certain features lacking from the platform which will inevitably be added later on, making the platform a moving target in terms of writing an application and generally leading to wasted time. For instance, the simulator, which simulates an Android phone for testing purposes, does not yet support either Wi-Fi or Bluetooth in its environment. For an application such as this one that utilises local wireless connectivity so heavily, it means we are limited in the simulator testing we can achieve with regards to data transmission.

I have been able to offset this issue by acquiring some HTC Magic mobile phones, which run Android and will be useful in on-device testing of real-world application behaviour. They will also be useful in evaluating the performance of the application on a real device, as described in chapter 4.

2.2 Requirements analysis

The requirements for the application should be determined from the original proposal of the project, and from thinking about what a user of the application would want from the use cases already mentioned.

- To be able to broadcast live video to other application users.
- To be able to tag a stream with interests.
- To be able to choose a stream to watch if more than one are available.
- To be able to watch a video while it is still being broadcast live.
- To be able to declare interests and receive videos based on those interests.
- To have a clear, easy to use graphical user interface.
- To receive broadcast video within seconds of it happening live.

2.3 Application design

To achieve the requirements of the project, my application needs to be able to record video while concurrently transmitting video already recorded over Haggle.

The Android *activity*[2] that actually records the video has a Camera object representing the phone's camera, and a SurfaceView class that represents the current preview of the camera's view displayed on-screen. Android provides a Camera .PreviewCallback interface, where the onPreviewFrame() method can be overridden. This method is called for every preview frame that is displayed on the SurfaceView, so if the frame rate of the preview is set at 15 fps, this method will be called fifteen times per second. This method provides the captured frame as a **byte** array, encoded in the YC_BC_R 4:2:0 Simple Profile format[3].

Unfortunately, without throwing away chromaticity information in each frame, the format used needs to be decoded to the RGB format for the higher-level methods of Android to be able to use it.[4] The Java virtual machine Android uses is incapable of converting these frames at a high enough rate to be able to transmit acceptable quality video.

[2]See section 3.1 for more on activities.

[3]The scheme stores four bytes of luminance (Y) data and two bytes of chromaticity (C_B and C_R) data for every four pixels, meaning frame size = $width \times height \times \frac{3}{2}$ bytes.

[4]`http://code.google.com/p/android/issues/detail?id=823#c4`

Furthermore, this would only save the frames of the camera preview—we still need to record the audio. This is possible using the MediaRecorder class, but it would be complicated to exactly synchronise the audio recorded with the frames saved from the preview. Very fine timing information would be required or else video clips like people talking or singing may appear obviously out of sync. There is also an issue with packaging these frames and audio clips up as a video file to play on each of the viewer's mobile phones.

I instead chose to use the MediaRecorder class to directly record video encoded in a specified format. However, this class only records a video to a static file, so we need to record a chunk of video at a time, transmitting the already recorded chunks over Haggle while recording new ones. This is better than the previous scheme in that footage taken is recorded directly in to video files, meaning any space benefits from the compression of the chosen video encoding scheme are utilised in transmission, rather than sending uncompressed video frames and audio separately at a larger cost. It does mean that, dependent on the duration of each video chunk, the stream seen by viewers will be behind the live event thanks to the length of time taken to record the first file and send it.

2.3.1 Video and audio codecs

In choosing to send video files over the Haggle network, the choice of video codec becomes important. This needs to perform techniques to reduce file size of videos so that they can be sent over the network as quickly as possible to reduce the delay of a stream behind its broadcast. However, it also needs to have broad support across the Android platform.

The Android platform has a number of core audio and video codecs that are supported by all devices[5], as well as device-specific codecs that are supported by certain devices. For encoding, the main codecs in the platform are H.263 video[7] and Adaptive Multi-Rate Narrow Band (AMR-NB) audio[1].

H.263 is a video codec that was originally designed as a low bit rate format for teleconferencing, but has been used recently as a format for video on the Internet on the aforementioned video sharing sites such as Youtube. It uses both inter-picture prediction to take advantage of the relative similarity of adjacent video frames in a live recording, and transform coding of the signal to reduce spatial redundancy. AMR-NB adapts the bit rate it encodes audio at depending on the information required, and can reduce bandwidth used during periods of silence.

As these codecs are natively supported by all Android mobile phones, they will

[5]`http://developer.android.com/guide/appendix/media-formats.html`

Name	Value
"tag"	A string representing one particular facet of the video, used to match the video to an interested device. At least one required.
"seqNumber"	An integer specifying the video chunk's place in the stream, depending on the counter used. Exactly one required.
"id"	A string uniquely identifying the video stream. Exactly one required.
"startTime"	A long with the Unix timestamp of the video stream start time. Exactly one required.
"isLast"	A boolean value that, if true, denotes the end of the stream. Exactly one required.

Table 2.1: Table of attributes included with each data object of video

ensure that the application has wide support. The compression offered by both codecs also fits in with reducing the size of video chunks as much as possible without a drop in quality, meaning they should be transmitted across the peer-to-peer network faster. For this reason, I have chosen to use these codecs in encoding the video that I will send across the network.

2.3.2 Data objects

As I mentioned in section 1.2, Haggle uses the concept of a data object to send data over the network to other Haggle nodes. To ensure subscribers get the video relevant to them, data objects contain a bag of *attributes*, one of which is a $(key, value)$ pair of strings that determine whether the data object should be forwarded to a device. If a device has a recorded value for a given key that matches the data object's value for the same key, the data object is forwarded to the device. There can be more than one data object with the same key.

Table 2.1 shows the data structure of the data objects sent from the application. This shows that the application is greatly extensible with many different attributes possible, for example social network ID, location data, or directional compass data.

A problem arises when you consider the video stream ID needs to not only disambiguate video streams started at different times from the same mobile phone, it also needs to disambiguate mobile phones from each other. But how do we achieve this in an ad-hoc, connectionless network? One possible way is to use both the start time of the stream and the MAC address of the mobile phone.

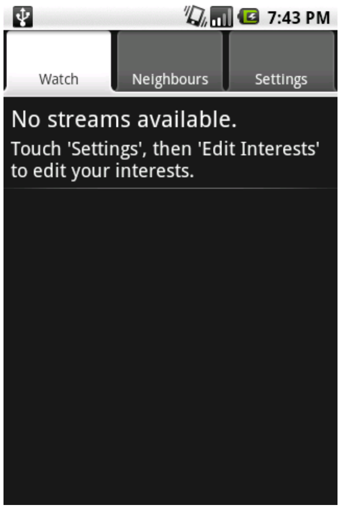
The standard for MAC addresses[6] says that the first 24 bits of a MAC address are "organisationally unique", in that each organisation is given a unique 24 bits. The following bits of the address can be assigned by the organisation in any way, with the constraint that they all need to be unique. This nominally shows that using MAC address in the stream ID will give uniqueness. Unfortulately, a technical limitation of Android's current implementation of ad-hoc Wi-Fi means that it currently needs to be turned on manually after phone boot by editing files on-device through a shell script. This has the side effect of returning **null** when calling the Android method getMacAddress(), so we cannot currently use it.

Another alternative would be to use the IMEI or other SIM card-related identifier, which we know would be unique, as phone manufacturers are "responsible for ascertaining that each IMEI is unique"[2]. However, for this to work, our application requires the READ_PHONE_STATE permission, which is a very wide permission that lets the application read a lot of private information that users may not agree to the application requiring.

To get around these limitations, the application uses the special Android ID, which is a 64-bit hex string uniquely assigned to secure storage in each mobile phone when they access the Android Market for the first time, and then stays in persistent memory. There are issues with guaranteeing uniqueness in mobile phones which have been *rooted* (root access has been gained on the phone), as the user can edit the secure storage where the Android ID is stored. However, this should be beyond the limitations of most target users in regards to my use cases, and it is an edge case which is incredibly unlikely to ever occur. Even assuming a set of users had set their Android ID to be equal—they would also need to start streaming at exactly the same millisecond for a collision to occur.

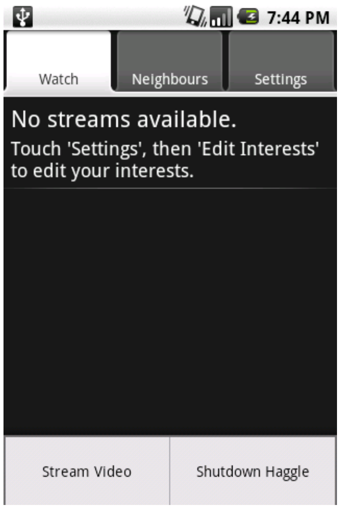
I have therefore chosen to create the video stream ID by taking the Unix timestamp at the beginning of the stream, concatenating it to the Android ID and running the resulting string through the SHA-1 hash function. As the seed for the function should be unique, the generated hash for each video stream should also be unique, barring any possible collisions from SHA-1. Hashing the Android ID ensures this secure value isn't leaked.

I also considered using the lollipop sequencing scheme[11] to implement the counter for the sequence number, as it doesn't eventually run out of sequence numbers like a sequential counter. Lollipop sequencing counts through the negative numbers, then counts through the positive numbers indefinitely. I then chose to use a simple sequential counter as it is unlikely video streams will last long enough to require a loop around the positive numbers, given the size of the integer number space in Java, the size of each small video file and the memory available on each mobile phone to store that much data.



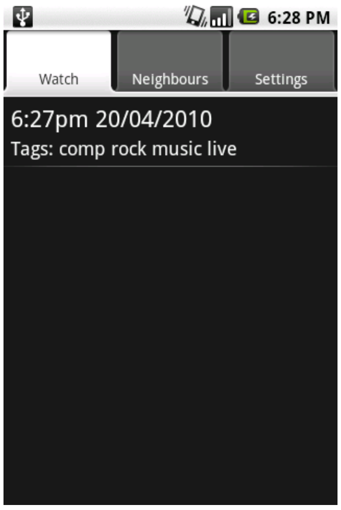
No streams available.
Touch 'Settings', then 'Edit Interests' to edit your interests.

(a) Without options menu visible



No streams available.
Touch 'Settings', then 'Edit Interests' to edit your interests.

(b) With options menu visible



6:27pm 20/04/2010
Tags: comp rock music live

(c) With a stream available to watch

Figure 2.1: Images of the front view of the application

2.4 User interface design

As a program that will be used by many end users, the user interface is very important so that the program's features are visible and easily found, the application is easy to use and understand and the application is consistent with other paradigms of general Android applications.

Figure 2.1a shows the main user interface of the application when it is launched. It launches to the "Watch" tab directly, so that any relevant streams will be displayed instantly. Figure 2.1b shows the state of the application when the menu button is pressed to bring up the options menu, a paradigm common to all Android applications, and hence familiar to Android users.

The tabbed system allows a user to navigate to any information view from wherever they are in the application. I chose to use the tabbed system as this allows for clearer navigation between different components of the application. When streams become available to watch, they appear on the stream list, along with the date and time the stream was started, and the interest tags that have been assigned to it. (See Figure 2.1c.) This gives clear information about each available stream and the content within them.

2.5 Backup and version control

Both of these were important to have in place early on in the project, as any data loss even during the preparation stage would impact on the timeline of the project.

Both Git and Subversion were possibilities for version control. Were my project being developed by more than just myself, I would choose Git as I think its semantics of cloning the repository to work on locally perform better for large projects than Subversion's method of having a central repository. However, as I am working on this project myself, and I have more experience with Subversion, I decided to use that as a version control system.

Subversion includes a number of "hooks", which are scripts that are executed at certain times on the repository. These will be useful for backups, as I will use my repository's post-commit hook to perform automatic backups after each revision. I will make an incremental backup for every revision, and a full backup of the repository every five revisions. These will be made to the Dropbox on my local machine (which synchronises local files with files in cloud storage), and remotely sent to a separate server. This will mean I will have backups in two remote places.

I also have my Subversion post-commit hook send an e-mail with details of changes for each revision to my own e-mail address. This will help in tracking changes for the project.

Chapter 3

Implementation

In this chapter, I will discuss the parts of the application, how they were designed & written and how they work together to meet the project requirements.

3.1 Android programming

There are a number of common elements to writing any Android program that are required for the code to run properly on all devices. Firstly, all applications have an XML manifest file that determine their name, default package, system permissions (Internet or camera access, for example) and the activities that are part of the application. Each application has one Application class that is launched when the application is opened and remains in memory until the application is destroyed. There is also one main Activity class that is launched along with the application, but this activity (or indeed, any activity) can launch many other activities to perform certain tasks.

New activities are launched and data passed between them by using Intents. Listing 3.1 shows an example of putting data in to an intent and using it to start a new task in a new activity. As it was initiated by startActivityForResult(), this new activity will eventually return with a result and any new data packaged in another intent,

```
String mapKey = (String) obj;
Intent i = new Intent();
i.setClass(getApplicationContext(), StreamViewer.class);
i.putExtra(STREAM_ID_KEY, mapKey);
startActivityForResult(i, Vidshare.WATCH_STREAM_REQUEST);
```

Listing 3.1: Code executed when a stream is selected to watch

invoking onActivityResult().

One of the difficulties with having modular code in different activities is the issue of sharing data between them. Android provides for this in a number of different ways, like saving preferences for each application and using Content Providers to allow access to stored data[1]. For the uses of my program, I only required shared resources during run time, so any state that needed to be accessed by multiple activities I stored in the Application class. Activities could then get a handle to the application class by calling getApplication() and accessing the shared state. I also ensured that the state was robust to multiple concurrent accesses, as there is no guarantee this will not happen.

As defining user interface elements programmatically would result in long blocks of static code, Android also allows definition of layout elements using their own format of XML separate from activities. This promotes a model-view-controller style system, decreasing dependencies between graphical user interface and code. In the main layout XML of the project[2], I have defined the layout of the first view of the application. The attribute android:id="@+id/stream_list" in the ListView element is used to refer to and change that particular list in the code itself—Android allows it to be found programmatically via a unique resource number allocated to each new element and the findViewById() method.

3.2 Application overview

In this section I will give a brief overview of the main classes I have written, along with their role in the overall application. I will go further in depth with their behaviour in later sections.

- **Vidshare**: the Application subclass. It is the first class to be instantiated and is in memory throughout the entire program life cycle.
- **VSActivity**: the main Activity subclass. This is the first activity to be launched. It displays and is responsible for the user interface. Other activities are launched from it.
- **VideoStream**: an Activity subclass. This part of the application is launched when a user wants to broadcast a video. It records the video chunks and publishes them to the Haggle network.

[1] `http://developer.android.com/resources/faq/framework.html#3`
[2] See Appendix B.

- **StreamViewer**: an Activity subclass. This is launched when a user selects a stream to watch from the stream list. It marks a Stream as being viewed, receives the video chunks from that stream and plays them sequentially.
- **Stream**: the class that identifies and represents a stream of video chunks being transmitted to the device. Sends the video chunks to the StreamViewer if the stream is currently being viewed.
- **InterestView**: an Activity subclass. Displays the current interests of the user that will decide what video content they receive. Allows them to add and delete entries.
- **AddVideoAttributeView**: an Activity subclass. This is launched before a video is broadcast to add information about the stream, like its tags.
- **DataObjectListener**: This is a custom listener that is called by a Stream that is currently being watched and invoked on the StreamViewer. The StreamViewer receives a **DataObjectEvent** object with the details of the new video chunk.

I have also written some small helper classes, namely **DateHelper** to be able to format the date in a readable way from a Unix timestamp and **Counter** to provide a consistent sequencing scheme for sending video chunks over the network.

3.3 Starting the application

The main Application class (Vidshare) is the class that is first instantiated when the application is launched. As such, it is the class that needs to ensure Haggle is running, and if not, initialise it. To do this, we need to start the Haggle daemon that will run in the background collecting updates from neighbour devices and new data objects. We then need to get a handle on to this daemon so that we can use it to publish and receive data objects and neighbour updates from the Haggle network. As this communicates with the native code of Haggle, it needs to be safe from failure at all stages of intialisation. The initialisation code returns a specific error code if it fails at any particular point in the process, allowing the user to diagnose issues in a clearer way and provide better information when/if seeking support.

As seen in Listing 3.2, once the handle to the Haggle daemon is successfully attained, we can register for certain updates from the Haggle network and begin the process of watching these for updates. The class implements the Haggle EventHandler, which allow us to override methods associated with each event to be called each time that event is triggered. This means that all updates will go

```
// hh is the Handle to the Haggle network.
hh.registerEventInterest (EVENT_NEIGHBOR_UPDATE, this);
hh.registerEventInterest (EVENT_NEW_DATAOBJECT, this);
hh.registerEventInterest (EVENT_INTEREST_LIST_UPDATE, this);
hh.registerEventInterest (EVENT_HAGGLE_SHUTDOWN, this);
hh.eventLoopRunAsync();
```

Listing 3.2: Code showing the types of update we can receive from Haggle

through the Vidshare class and can then be sent to the main activity (VSActivity) to update the state of the application.

3.3.1 Application state

There can only be a singleton reference to the Haggle handle, which is stored in the Vidshare class. Other activities need to access this handle to publish data objects and get other information about the network. As mentioned in section 3.1, state stored in the Application class (subclassed as the Vidshare class) can be accessed from activities via a method call. By ensuring all accesses on the Haggle handle are synchronised, we avoid problems with two or more activities trying to access it concurrently and possibly causing non-deterministic behaviour.

The state that we need to store in the application class is a ConcurrentHashMap of Stream objects called mStreamMap, indexed on the unique ID of the stream. A Stream is a custom object that collates received data objects within the same broadcast into one easily accessible place for playback. For reasons I will discuss in section 3.4.1, we also need to store a ConcurrentHashMap of booleans called mStreamAliveMap, indexed on the unique IDs of streams. This indicates whether the indexed stream is still streaming (value **true**) or has stopped streaming (value **false**).

When important activities get a handle to the Vidshare class, they also give the Vidshare class a handle to themselves. This is important as the application will need to update state in some of the activities and needs to have a reference to them to do this. We need to ensure there is only a singleton of each activity known to the application class. I have ensured they are singletons by having a single reference to them in the application class and by using the Android onStart() and onStop() methods in each activity to set the reference and clear the reference in the Vidshare class respectively.

```
// act is the reference to VSActivity, dObj is the newly arrived DataObject.
act.runOnUiThread(act.new DataUpdater(dObj));
```

Listing 3.3: Code to update the user interface after an event

3.4 Main activity

After the application class is initialised, the main activity is started and the user interface for the program is fetched from the main XML layout file[3] and displayed. As seen in Figure 2.1, the user interface has a tab widget allowing the user to select from video, neighbours and settings. The video and neighbours tabs have a list of video streams and connected devices respectively. But as mentioned in section 3.3, all updates from the Haggle network go through the Vidshare class. We need a way of both getting those updates to the main activity, and automatically updating the lists.

Inside VSActivity, there is a Runnable inner class defined called DataUpdater. When it is run, either for a new data object or for a change in devices connected (neighbour update), this class calls the custom update method on the *adapter* for the respective list. These adapters (StreamAdapter for the video stream list and NodeAdapter for the neighbour list) are subclassed from the basic BaseAdapter class. They provide another way of decoupling user interface elements from programmatic ones by inserting a shim between the data structure and the UI list.

3.4.1 Processing a new data object

This section describes the steps that occur when a video chunk arrives at the application. Figure 3.1 shows a diagrammatic representation of these steps.

When a new data object arrives and the matching event method in Vidshare is triggered, the program first of all verifies that the information in the attributes of the data object is not null, and also checks if the video file in the new data object has been successfully saved in the phone's cache. It then uses its reference to VSActivity to send the data object to the DataUpdater. It does this using the call in Listing 3.3.

Android requires the DataUpdater to be run using runOnUiThread() as it will be changing user interface elements of the application, namely the lists. As the Android user interface toolkit is not thread-safe, all UI changing calls need to happen on the

[3]See Appendix B.

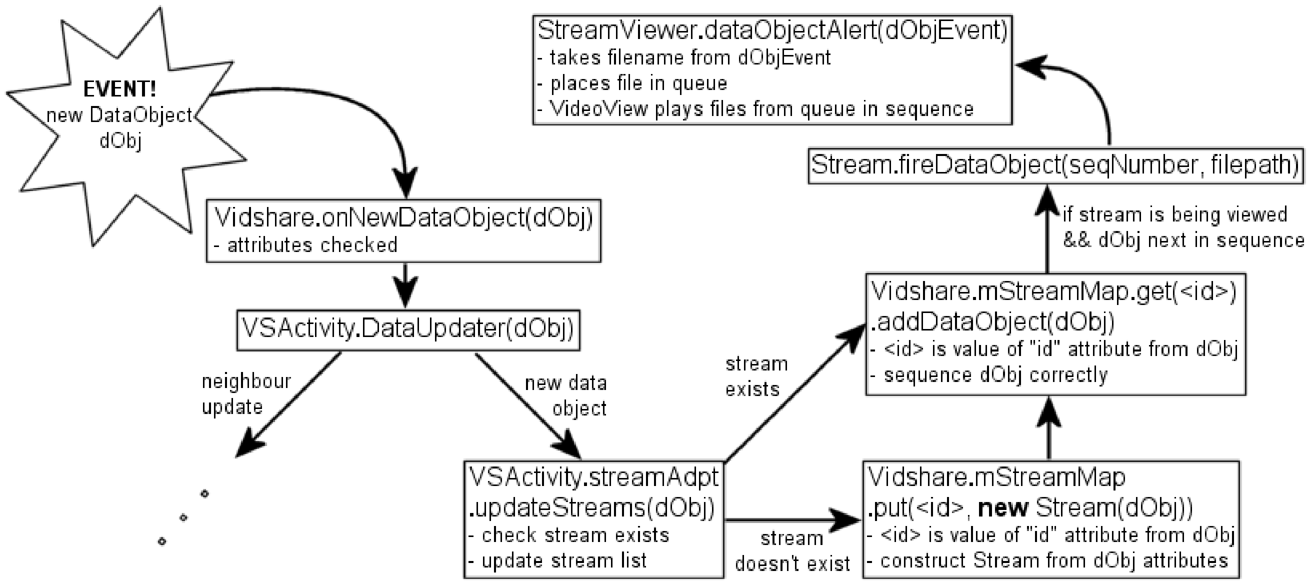

EVENT!
new DataObject
dObj
Vidshare.onNewDataObject(dObj)
- attributes checked
VSActivity.DataUpdater(dObj)
neighbour update
new data object
VSActivity.streamAdpt
.updateStreams(dObj)
- check stream exists
- update stream list
stream exists
stream doesn't exist
Vidshare.mStreamMap
.put(<id>, new Stream(dObj))
- <id> is value of "id" attribute from dObj
- construct Stream from dObj attributes
Vidshare.mStreamMap.get(<id>)
.addDataObject(dObj)
- <id> is value of "id" attribute from dObj
- sequence dObj correctly
if stream is being viewed && dObj next in sequence
Stream.fireDataObject(seqNumber, filepath)
StreamViewer.dataObjectAlert(dObjEvent)
- takes filename from dObjEvent
- places file in queue
- VideoView plays files from queue in sequence

Figure 3.1: Diagram showing the main method calls when a new video chunk arrives in a data object

main UI thread[4]. The DataUpdater then invokes the updateStreams(DataObject dObj) method on the StreamAdapter.

Using the state from the Vidshare application class defined in section 3.3.1, the method firstly checks whether the mStreamMap contains a key that matched with the video stream ID in the newly received data object. If it does, then this data object is part of that video stream. In most cases, we can simply add the data object to the Stream, however we first check if the stream has already ended—if (final data object with ("isLast" == **true**) has already been received && there were no more data objects waiting out-of-sequence).

If it has ended, we disregard the new data object and remove that stream from mStreamMap. Importantly, we also set the value of the stream ID key in mStreamAliveMap to **false**. This is because if we receive a data object from this stream out-of-sequence after the final data object has been received, without the mStreamAliveMap we have no record of ever having had that particular video stream before, as it will already have been removed from the stream map. That will mean a stream that has already ended will be displayed on the video stream list, and it will never be removed as the final data object that removes it will have already been processed.

In the case where mStreamMap does not have a record of the stream ID from the newly received data object, we first check with the mStreamAliveMap to ensure this

[4]`http://developer.android.com/resources/articles/painless-threading.html`

is not an out-of-sequence data object from an already finished video stream, to avoid the issues mentioned above. If there is no record of the stream there, we can go ahead and create a new Stream object in the correct place in the stream map.

We then notify the adapter that the underlying data set has changed. This invokes a redraw of the list according to the data the adapter gives to it. Because we can exactly define the View that becomes the list item, the stream list can display extra information like the date the stream started (formatted in a nice, human-readable way using the DateHelper) and the tags associated with the stream. This entire process is triggered by a new video chunk being received by the device.

Once the stream list has been populated with streams from mStreamMap, we can define an OnItemClickListener that will invoke a method that gives us the original adapter of the list, and the position of the item that was pressed. Using these, we can call getItemAtPosition() on the adapter and get the video stream ID, which can then be passed on to the StreamViewer through an Intent object, as seen in Listing 3.1.

3.5 Representing a video stream

As the small chunks of video come in to the application as data objects over the Haggle network, they are each added to a Stream object, which represents all of the data objects from one particular broadcast. For this to work effectively, the Stream class needs to implement techniques used in network data transmission protocols like TCP. It needs to provide its own way of re-ordering data objects so that complication can be abstracted away from the StreamViewer. It also needs to provide a timeout mechanism to employ in the case where the remote device has crashed, or the network has disappeared, or we have a hardware problem, or any other situation where data objects cannot be transmitted over the peer-to-peer network.

When a data object with a stream ID that isn't in the stream map arrives, subject to stipulations already mentioned in section 3.4.1, it will be used to create a new Stream object. The ID for each video stream comes from the "id" attribute of this first data object. Having this ID in each new data object ensures any device can pick up a stream from whatever point in its broadcast. Section 2.3.2 explains how I chose the video stream ID to be unique.

The program manages video streams finishing in a certain way. When the broadcasting user decides to end the broadcast, the last video file is published and an empty sentinel data object is then published to Haggle with no payload data attached, but with the normal ID, sequence number and tag attributes, along with

```
// chunks is a HashMap of video filepaths indexed on sequence number.
if (!streamEnding) {
    chunks.put(seqNumber, filepath);
    sequenceChunks(seqNumber, filepath);
} else {
    if (!expected.isEmpty()) {
        chunks.put(seqNumber, filepath);
        sequenceChunks(seqNumber, filepath);
    } else {
        streamEnded = true;
        fireStreamEnded();
    }
}
```

Listing 3.4: Code to process a new data object coming in to a Stream

the "isLast" attribute now set to **true**. When the addDataObject() method in the Stream object sees this attribute, it sets the stream into "ending" mode. Listing 3.4 shows that in "ending" mode, the program checks if there are any data objects that have not yet arrived before finishing the video stream completely.

When a Stream is created, it instantiates a Timer object, which runs as a daemon and runs scheduled tasks at certain times. Then, when a data object is added, it schedules a TimeoutTask to be ran after a certain timeout period. This countdown is restarted when a new data object arrives. When the timeout reaches zero, it will fire a timeout event if the stream is currently being watched, which will let the user know there has been a problem with the video stream before gracefully exiting. I have set the timeout length to be 15 seconds, considering the default video length is 3 seconds and the maximum size of the toFire is three, this should mean the timeout is activated when we really have lost communication.

3.5.1 Data object sequencing

Also shown in Listing 3.4 are calls to the sequenceChunks() method. Because of the possibility of data objects arriving out of sequence due to the nature of the wireless media being used to transmit and the possibility of contention, there needed to be an algorithm to deal with re-ordering data objects before they were played back in sequence. The full sequenceChunks() algorithm is in Appendix C.

The algorithm first deals with the general case of the data objects arriving in sequence and fires the data to the StreamViewer if the user is watching the current video stream. The code in lines 14–27 checks the ordered toFire tree for the next number in the sequence to see if it can quickly fire off a chain of waiting data objects already received in sequence. The case in line 28 deals with received data objects sequenced later than the expected one. We work out the data objects in between that should have arrived first and put them in the expected set, then we add the received data object's sequence number to the toFire ordered tree, indicating it is ready to be fired to the viewer when the expected data objects before it have arrived.

When lots of video chunks are received out-of-order, or in the case where one data object is very late and the subsequent ones are properly received, the algorithm currently continues to collect new data objects in the toFire list, but won't fire any of these until it has received and fired the missing data object. This is bad, as that data object may never arrive, and even if it does, it will mean the video stream is now unacceptably behind the live video broadcast.

To get around this problem, the **while** loop starting on line 37 ensures that the toFire tree size can never grow larger than a certain defined constant. Once it goes above this size, the next waiting data object is automatically fired if the stream is being watched, no matter which was played before. While it will be jarring for the user for the video to skip ahead, in most cases this will happen during a "Loading" screen on the StreamViewer, which is displayed when it has no more video chunks left to play. It also ensures the video stream is as close to the live broadcast as possible while still having a buffer zone before having nothing to play. The maximum length of the toFire list is currently set at three, ensuring there is never a pause in the video longer than $maximum\ size\ of$ toFire $\times\ length\ of\ video\ chunk$. After the first data object in the toFire tree is fired, in lines 43–56 we again try to quick fire waiting data objects in a chain if they are already in sequence.

3.6 Watching a video stream

The StreamViewer is the actual activity that plays back a chosen video stream to the user. When a video stream is selected from the stream list in the main user interface, it starts the StreamViewer activity and passes it the stream's ID in an Intent object. The stream viewer then gets the Stream object from the stream map, sets it as the current stream in its own activity and also sets it as being viewed. This has the effect of the Stream starting to fire newly sequenced data objects' filepaths as part of DataObjectEvent objects to the StreamViewer.

We ensure that we only have one singleton reference to one Stream by setting the reference and setting the Stream as being viewed in the onStart() method, while clearing the reference and setting the Stream as not being viewed in the onStop() method, both of which are called in the lifetime of the activity.

The dataObjectAlert() method deals with the events coming in from the video stream currently being viewed, including new data objects in sequence. It also handles timeout events sent from the stream, when new data objects stop coming in. In this case, it displays a timeout message and falls back to the main menu, now devoid of this timed out video stream. Any new data objects in this stream will place it back on the stream list, ready to select again. If it received an event saying the stream has ended, it also finishes the activity and returnd to the main menu.

When a data object is received, it is places in a ConcurrentLinkedQueue of filepaths. I have used this data structure as data objects arrive sequentially in the correct order, and are removed to play back in this same order, like in a First In First Out queue. Because filepaths may be added to the queue at the same time they are removed for playback, the data structure needs to be robust to concurrent accesses. A VideoView surface is used to play back these files. On completion of a video chunk playback, the queue is polled for the next filepath, which is then played back. If the queue is empty, a loading screen is displayed, with a message letting the user know that the video is loading and will be displayed soon.

3.7 Broadcasting video

Figure 3.2 shows the steps the application goes through to initialise a video stream before the process of recording begins. If the user cancels while adding tags, the result RESULT_CANCELED will be passed back to the main activity, which then knows not to continue with starting the next activity. If the result passed back is RESULT_OK, it starts the VideoStream activity.

The interface of this activity consists of a SurfaceView constantly showing a preview of the camera, with a camera shutter button in the corner of the screen, used to start and stop streaming video. Figure 3.3 shows the user's view when broadcasting video.

As mentioned in section 2.3, I am using the Android MediaRecorder class to record each small video file. This involves instantiating a new MediaRecorder (or resetting an old one), passing it the necessary options for recording, preparing the recording, then starting it. After it has stopped, we create a new Haggle data object with the resulting video file and publish it to the Haggle network with the necessary attributes. One of the issues with using MediaRecorder to record small files back to back like this on Android 1.5 is the length of time between stopping the previous recording and starting the next. There are a few reasons for this.

First of all, Android protects the camera when it is being used by locking and unlocking access to the hardware. This version of Android doesn't provide any public Java methods to lock or unlock the camera, meaning that the Camera reference held in the class can't be reused by the MediaRecorder object. This lengthens

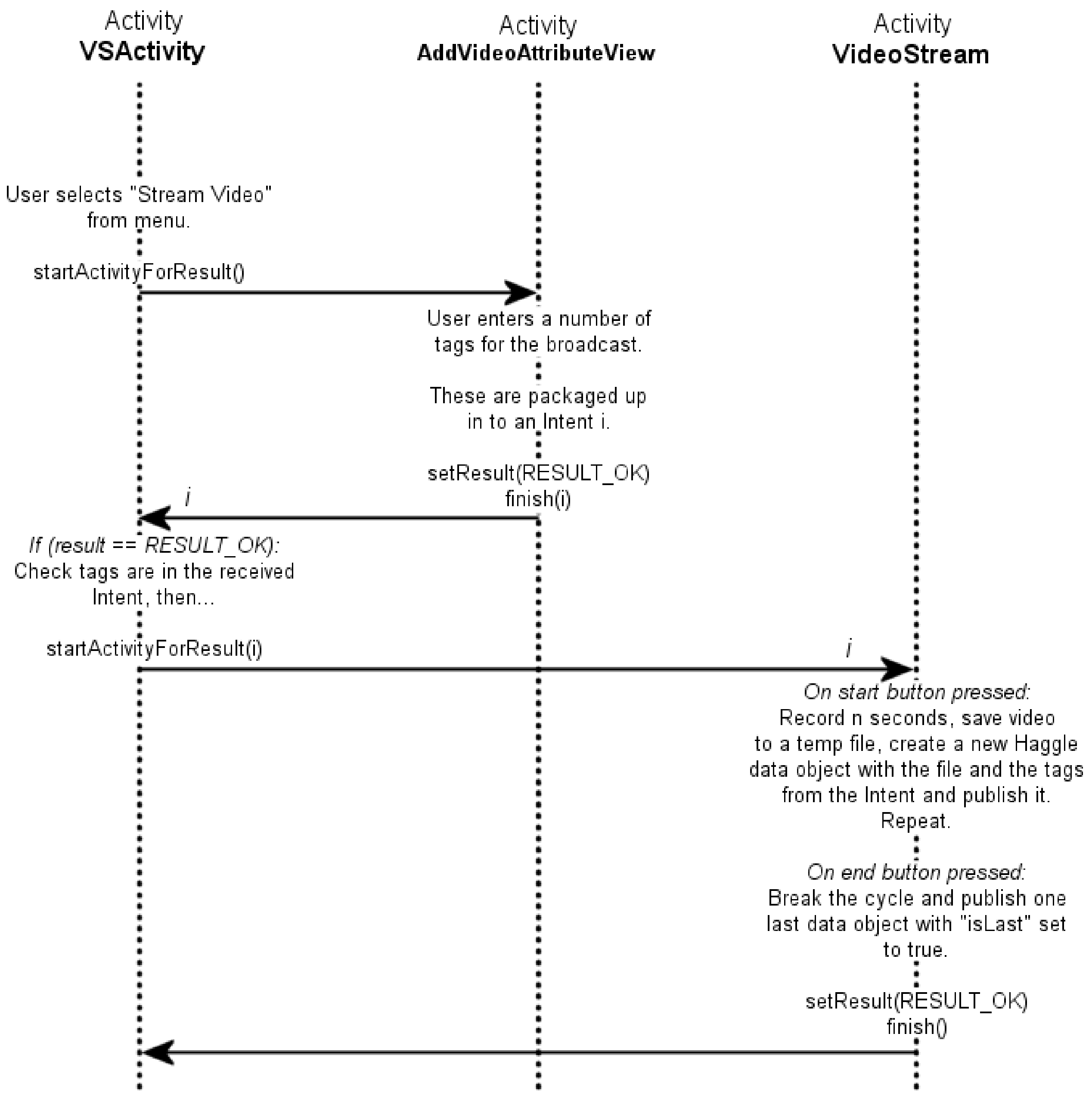

Activity
VSActivity
Activity
AddVideoAttributeView
Activity
VideoStream
User selects "Stream Video" from menu.
startActivityForResult()
User enters a number of tags for the broadcast.
These are packaged up in to an Intent i.
setResult(RESULT_OK)
finish(i)
i
If (result == RESULT_OK):
Check tags are in the received Intent, then...
startActivityForResult(i)
i
On start button pressed:
Record n seconds, save video to a temp file, create a new Haggle data object with the file and the tags from the Intent and publish it. Repeat.
On end button pressed:
Break the cycle and publish one last data object with "isLast" set to true.
setResult(RESULT_OK)
finish()

Figure 3.2: Diagram of the control of the application during video streaming

Figure 3.3: Image of the application while broadcasting video

any pause between recording video chunks because the class needs to get its own reference to the camera each time. Also, after every time the MediaRecorder class is used, it needs to be reset and all of the options given to set it up need to be given again. As well as these options, it also needs to be prepared again before recording. This also takes up considerable time between recordings.

I have taken a number of steps to ameliorate these issues. Firstly, I have made the activity as multi-threaded as possible, so that steps that can be taken in parallel are performed in seperate threads while video is recording. As soon as each video chunk is recorded, a new thread is launched to create the Haggle data object, add the attributes to it and publish it to the peer-to-peer network. This ensures that the next video chunk's recording is started as soon as possible after the previous one ends, while the data object is still published to the network promptly in a separate thread. An evaluation of how effective this is is shown in section 4.3.1. Figure 3.4 shows the semantics of the threads in this activity.

To further increase switching speed between recording each video chunk, I have also decided to instantiate two MediaRecorders in a two object synchronised ArrayList . I chose to use a synchronised ArrayList as opposed to a normal array as there is an issue with multiple threads accessing this construct at once. To control access between both MediaRecorders, I have an integer variable that is used as an index in to the ArrayList, which is flipped from 1 to 0 (and vice versa) by XOR-ing the value with 1 after operations on it are finished. By only accessing the ArrayList through this variable, I can ensure no changes occur on the currently recording one.

This means I can set up the unused MediaRecorder to record the next video chunk while the other one is recording the current video chunk. This is also performed in a separate thread to ensure as little time as possible is used switching to the next MediaRecorder. The majority of operations on the unused MediaRecorder can be performed in parallel to the operation of the other, however the prepare() method—required just before recording—needs to happen after the previous recording has ended, or else undefined behaviour can occur. The operations that can be performed in parallel are separated into the setUpNextRecording() method, while the

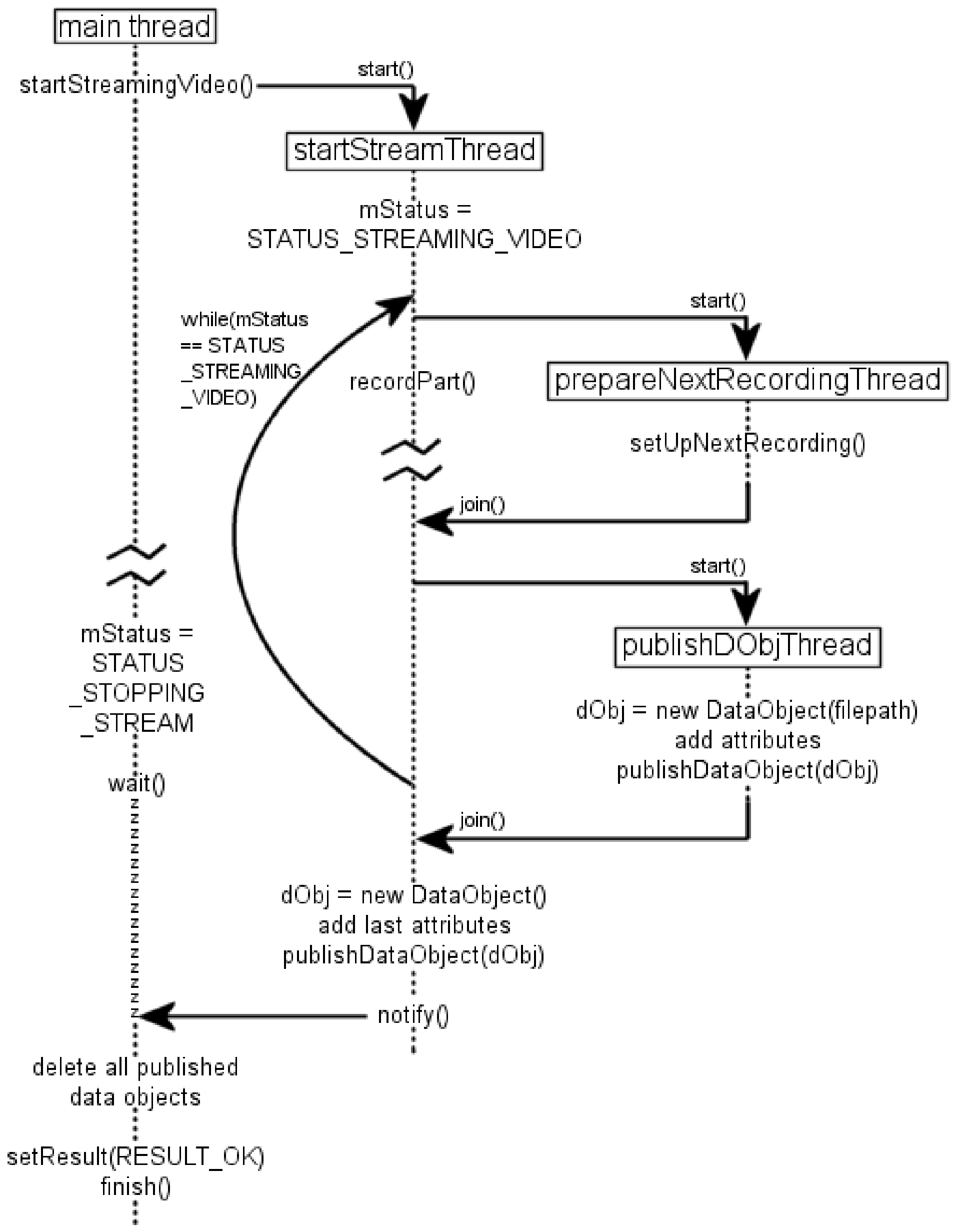


Figure 3.4: Diagram of thread control during video streaming

operations that need to run sequentially are kept in the recordPart() method.

This method of recording is set up in a **while** loop, which acts on a **volatile** status variable. The variable is volatile as it needs to be directly read from memory on each loop, as it may have been changed by another thread. When the user stops recording by pressing the shutter button, it changes the volatile variable from STATUS_STREAMING_VIDEO to STATUS_STOPPING_STREAM and then calls wait(), to wait for the final data objects to be published before continuing. The change in status variable breaks out of the loop, executes join() on the previous data object publish thread to ensure that completes first, then publishes the final sentinel data object with no payload data and with “isLast” == **true**. It then calls notify() to wake up the previous method, which then issues commands to the network to delete this video stream’s data objects. This occurs because cache sizes of each node would soon grow too large in a store-and-forward network such as this. So we have reference to them all to delete them, we keep a list of sent data objects during streaming. The activity then returns to the main user interface.

Chapter 4

Evaluation

This section will present an evaluation of the project in regards to the original specification, showing empirical results and examples to show how the original aims of the project were met.

4.1 Overall results

In the project proposal[1], a number of success criteria were defined to measure the success of the project.

To be able to send a video file from one phone to another using the system.

The project achieves this via being able to record a video file (many small ones, in reality) with it being saved on the Android filesystem which the Haggle architecture then publishes to the network. This results in the other phone mentioned receiving the same video file, which is consequently saved in its memory. A further evaluation of how successful this has been is in section 4.3.

I have also tested the behaviour mentioned in section 1.2 of these video files being forwarded through an intermediate phone to another subscriber not directly connected to the broadcaster. I was able to do this by acquiring another Android mobile phone to test on-device with the others I have been using already. This was successful, with empirical testing showing similar peer-to-peer transfer times as in section 4.3.

To be able to play this video file while part of it is still being received.

[1]See Appendix D.

The model of splitting up video files described in section 2.3 and then publishing them in parallel with continued recording using the multi-threaded model shown in Figure 3.4 achieves this criterion. In this case, the receiving phone can simply play these files back-to-back.

To be able to tag video with relevant interests for others to receive.

This is achieved by the AddVideoAttributeView activity taking user input on the tags they want to put on each video stream, then by adding these inputted values as attributes to each published DataObject, which can then be inspected by other phones to see if they match their own interested tag list.

To be able to select between received videos if more than one match at any given time.

The updating stream adapter connected to the stream list on the main application UI described in section 3.4.1 allows a user to select from numerous video streams. The singleton reference to the stream being viewed ensures we don't have problems with more than one video trying to play.

In general, the results of the project have been positive. Further examination of test benchmarks also shows a better than expected data transmission rate using the Haggle architecture. However, there were a few issues with the framework used that reduced the effectiveness of the project.

In regards to non-functional requirements, the application's tagging system along with the publish/subscribe model shows how scalable and extensible the project is, while user interface qualities of the system are discussed in section 4.4.

4.2 Testing

As I mentioned in section 2.1.2, because of the network-oriented nature of my project, I was limited in the amount of normal testing I could achieve, based on the architecture not being operable with the still-evolving Android SDK. However, testing the application was still very important, as there are many things that can go wrong when dealing with communication between remote machines.

To perform on-device testing, I used the Logcat interface, provided by the Android Debug Bridge program. This interface provides verbose messages about the state of the system, detailing the exact paths of execution of each method. Listing 4.1 shows some examples of sending log messages to the Logcat output, including messages sent at differing warning levels—the first is a more serious error log, sent

```
int ret = vs.getHaggleHandle().publishDataObject(dObj);
if (ret == -1) {
    Log.e(Vidshare.LOG_TAG, "***!!! Data Object returned error code. !!!***");
}
Log.d(Vidshare.LOG_TAG, "*** Last object published return code: "+ ret +" ***");
```

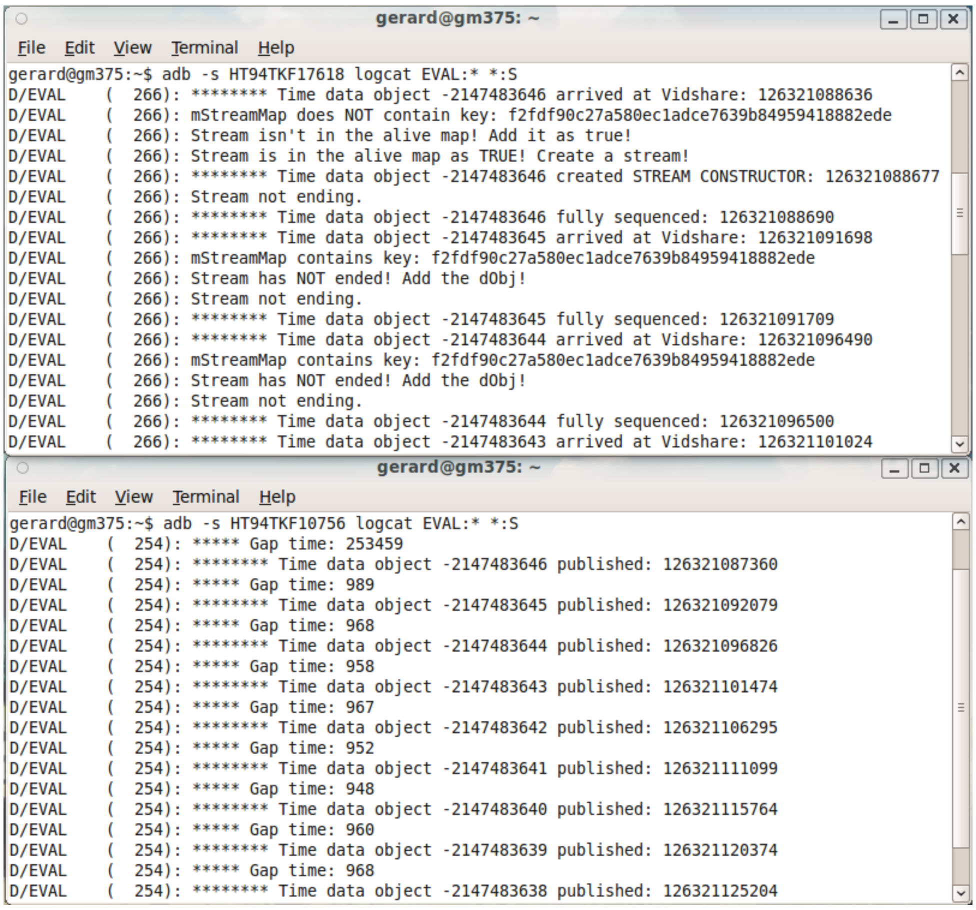
Listing 4.1: Code showing log messages submitted to the Logcat output, with varying degrees of severity

using the Log.e() method, while the second is a less serious debug message, send using the Log.d() method. I have used these liberally throughout my program to ensure I can see the flow of execution and debug any problems on-device this way. Figure 4.1 shows the output of two communicating phones.

I used this approach to test that video chunks were indeed being properly transmitted between phones, to test the rate at which this occured, to test the order in which the video chunks arrived (to ensure the sequencing algorithm worked) and to test whether the video stream exits and deletes itself cleanly when it finishes or times out.

A number of bugs I had were squashed due to a series of load tests I did. I performed five tests like this, where I left a video streaming for as long as possible before battery life ran out, and looked at the effect on any other conected phones receiving the stream. The first bug I found was in regards to the ArrayList of sent data objects I kept to ensure they could be deleted after a video stream finished. After 512 of these were added, the Java Native Interface reference table holding the native references to each data object overflowed and caused a fatal crash of the application. By deleting (and disposing of) old data objects when the ArrayList gets larger than 50 items, I kept the number of native references below the 512 limit. I chose the value 50 as these data objects are minutes behind the live broadcast and would no longer be relevant to a live video streaming application that this is aiming to be.

Another bug I found was that the persistent storage of the phone was being filled with video clips that would no longer be used, and after watching a few video streams in one session a user would find they would have no more memory to view or record any more. To combat this, I ensured that data objects were deleted and unpublished after the video stream concluded (and in the case above, deleted as the stream went on). In the case of errors closing the video stream prematurely, the files are also set to be deleted on exit from the Java virtual machine, so there is no buildup of stale video files.



```
gerard@gm375:~$ adb -s HT94TKF17618 logcat EVAL:* *:S
D/EVAL    (  266): ******** Time data object -2147483646 arrived at Vidshare: 126321088636
D/EVAL    (  266): mStreamMap does NOT contain key: f2fdf90c27a580ec1adce7639b84959418882ede
D/EVAL    (  266): Stream isn't in the alive map! Add it as true!
D/EVAL    (  266): Stream is in the alive map as TRUE! Create a stream!
D/EVAL    (  266): ******** Time data object -2147483646 created STREAM CONSTRUCTOR: 126321088677
D/EVAL    (  266): Stream not ending.
D/EVAL    (  266): ******** Time data object -2147483646 fully sequenced: 126321088690
D/EVAL    (  266): ******** Time data object -2147483645 arrived at Vidshare: 126321091698
D/EVAL    (  266): mStreamMap contains key: f2fdf90c27a580ec1adce7639b84959418882ede
D/EVAL    (  266): Stream has NOT ended! Add the dObj!
D/EVAL    (  266): Stream not ending.
D/EVAL    (  266): ******** Time data object -2147483645 fully sequenced: 126321091709
D/EVAL    (  266): ******** Time data object -2147483644 arrived at Vidshare: 126321096490
D/EVAL    (  266): mStreamMap contains key: f2fdf90c27a580ec1adce7639b84959418882ede
D/EVAL    (  266): Stream has NOT ended! Add the dObj!
D/EVAL    (  266): Stream not ending.
D/EVAL    (  266): ******** Time data object -2147483644 fully sequenced: 126321096500
D/EVAL    (  266): ******** Time data object -2147483643 arrived at Vidshare: 126321101024
```

```
gerard@gm375:~$ adb -s HT94TKF10756 logcat EVAL:* *:S
D/EVAL    (  254): ***** Gap time: 253459
D/EVAL    (  254): ******** Time data object -2147483646 published: 126321087360
D/EVAL    (  254): ***** Gap time: 989
D/EVAL    (  254): ******** Time data object -2147483645 published: 126321092079
D/EVAL    (  254): ***** Gap time: 968
D/EVAL    (  254): ******** Time data object -2147483644 published: 126321096826
D/EVAL    (  254): ***** Gap time: 958
D/EVAL    (  254): ******** Time data object -2147483643 published: 126321101474
D/EVAL    (  254): ***** Gap time: 967
D/EVAL    (  254): ******** Time data object -2147483642 published: 126321106295
D/EVAL    (  254): ***** Gap time: 952
D/EVAL    (  254): ******** Time data object -2147483641 published: 126321111099
D/EVAL    (  254): ***** Gap time: 948
D/EVAL    (  254): ******** Time data object -2147483640 published: 126321115764
D/EVAL    (  254): ***** Gap time: 960
D/EVAL    (  254): ******** Time data object -2147483639 published: 126321120374
D/EVAL    (  254): ***** Gap time: 968
D/EVAL    (  254): ******** Time data object -2147483638 published: 126321125204
```

Figure 4.1: Image of the test output from Logcat, with all but the “EVAL” tag filtered. The bottom window shows log output from the publishing phone, while the top window shows a subscriber receiving the published files. Different severity levels of log can also be shown/hidden

	Serial model	Parallel model
Mean gap time (ms)	1067.57	988.91
Standard error	± 4.39	± 4.14

Table 4.1: Table showing the average gap time after 95 runs

4.3 Empirical evaluation

4.3.1 Gap problem between video files

As I mentioned in section 3.7, there were some issues with gaps in between video chunks recorded, and a solution to ameliorate these problems. To ensure there was benefit to making the code in VideoStream multi-threaded, I decided to measure the time of this gap, between stopping the first video file and starting to record the next.

Table 4.1 shows the average gap time after 96 video chunks were recorded. It clearly shows that on average, the threaded model moved on to record the next video chunk (78.66 ± 8.53)ms faster using the threaded model. Despite being faster, this shows that even though most components of video recording have been parallelised, the largest issue in the process is still the prepare() method that needs to be run serially before recording, taking up the majority of the time. This time may be improved if we re-wrote the application to run on a later version of Android, which provides better facilities to share resources between MediaRecorders.

4.3.2 Peer-to-peer transfer times

We also need to ensure the time taken to transfer video files between phones is low, as some of the use cases (described in section 1.1) of the application would require broadcast video to be as close to the original broadcast as possible. For this reason it was important that time taken to broadcast video was measured and evaluated. I also wanted to change variables associated with each video file to ensure I was transmitting video in an efficient way.

I decided to measure time taken to transmit a video file over the two methods available—ad-hoc Wi-Fi and Bluetooth. I also performed the same tests with the length of each video file at different values. The results are shown in Figure 4.2. To synchronise time on both devices, I reset their clocks in parallel from a host computer, however there could be small timing errors with clock drift and the time difference between the clock being reset on each phone.

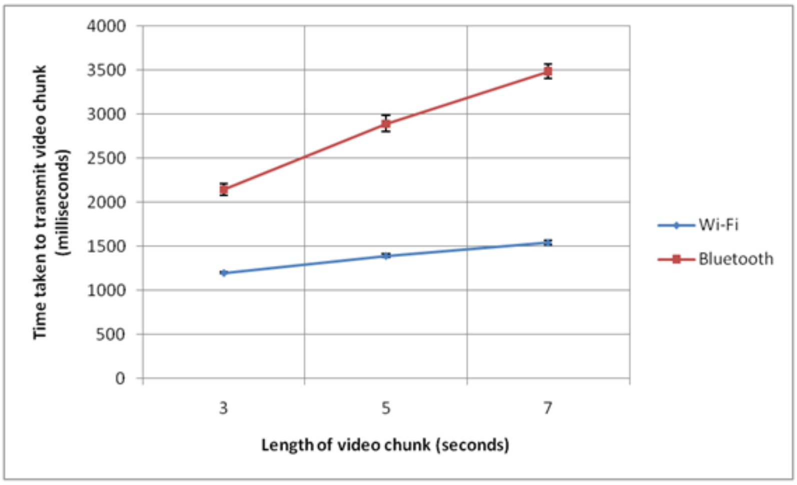


Figure 4.2: Graph showing the time taken to transfer video files recorded at 15 frames per second, with error bars showing standard error of the mean

For these results, I expected Bluetooth to take longer than Wi-Fi to transfer each video clip, but I was surprised by how variable the average time taken over Bluetooth was when the length of the video chunk was increased. It's clear that if the length of each video clip was set at 7 seconds and Bluetooth was being used, the delay in receiving the video would be very noticeable and inconvenient to the user. The relatively unchanging time taken over Wi-Fi as the video length changed was pleasantly surprising.

The previous test was carried out with video clips taken at 15 frames per second, so for the next test I decided to change this to the maximum currently possible through the Android software—24 frames per second. I expected this to have a noticeable detrimental effect on transfer times, albeit with smoother video quality. The results of this test are in Figure 4.3.

While we see a similar story happening in this test, what surprised me were how similar the times between the 15fps and 24fps tests were. While the extra information affected the longer Bluetooth times more, the Wi-Fi times stayed remarkably similar across the tests. For instance, the times taken to send a 15fps and 24fps 5 second video over Wi-Fi were (1393.08 ± 21.93)ms and (1425.45 ± 34.36)ms respectively—within standard error of each other. This shows that the video codec's method of taking advantage of inter-frame redundancy really does bring transfer times down and even makes it possible to always use 24fps when sending video. It does have a noticeable effect on the Bluetooth transfer times however, with the transfer times at 3 seconds being particularly variable.

Overall, this shows positive evidence towards using video lengths of three seconds, as the transfer times are the lowest, meaning that the broadcast video is as up to date as possible when transferring video files of this length.

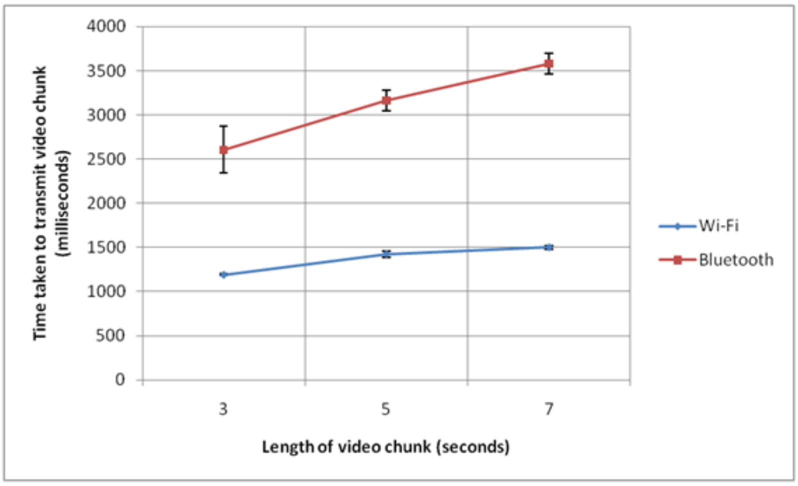


Figure 4.3: Graph showing the time taken to transfer video files recorded at 24 frames per second, with error bars showing standard error of the mean

	Battery life
Wi-Fi	2 hours, 19 minutes
Bluetooth	3 hours, 27 minutes

Table 4.2: Table showing the battery life of the application using different transfer methods

These times are likely to be volatile when in use by real users, where signal could be blocked by various objects, interfere with other signals or could suffer from the multi-path problem in a dense area. I have tested the program at normal distances when sending someone a file over Bluetooth, but the application could be put into use in harsher environments than that.

4.3.3 Battery life

In a mobile application, battery life is very important, as to fulfill the use cases the phone does need to be able to stay on for at least a few hours. The application needs to be able to run for the length of time it may be required to by a user.

I have made some recordings of battery life based on the device I have been using to test with—the HTC Magic. The results are shown in Table 4.2. The actual figures are only indicative as the health of batteries can deteriorate over time and the battery I am using isn't operating at its initial capacity.

The figures show that when sending using Bluetooth alone, battery life is preserved for much longer than simply using Wi-Fi. Using these figures, it is possible to imagine a possible extension to the application with a low-power mode using only Bluetooth, and a possible automatic switch when battery life gets too low.

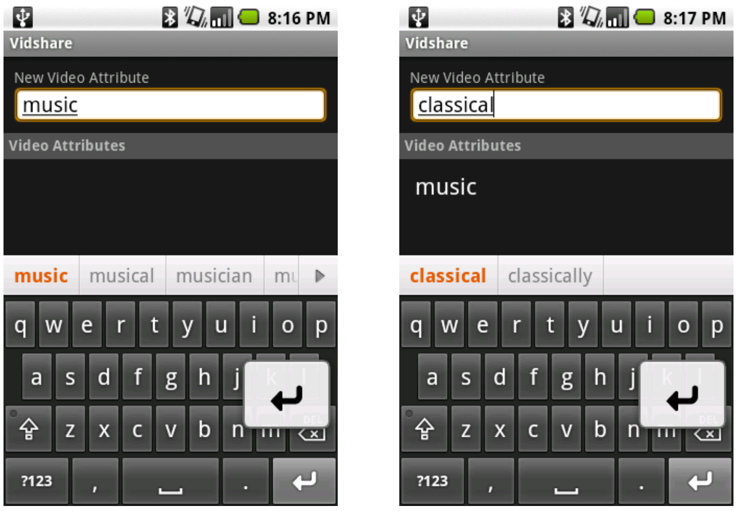
4.4 Cognitive walkthrough of user interface

I have decided to evaluate the user interface design of my application using a cognitive walkthrough, design to simulate what a non-expert user would do at each stage according to their behavioural model. For this we need some sort of model of a typical user according to the use cases, along with their typical knowledge.

We could say that a typical user has no knowledge of programming, but is not a complete technophobe either—they do own a smartphone, and download apps on it for business and pleasure. According to the use cases, they like to go to events like gigs and sports matches, and want to share what they watch with others in a community-like spirit.

While there are other tasks available in the application, the main representative task of the application is to record video and stream it to others in the local area. This has a list of correct actions that go along with it. For each action, we need to consider what the notional user's goal would be, and evaluate how accessible the the action is, how it matches up with the current goal, and the feedback given at that stage.

- Press the 'menu' button.
  - Current goal: To start recording a video.
  - Accessibility of control: To a user who knows about Android menu paradigms, this would be the first thing to do to look for extra options not displayed on screen. Figure 2.1a shows this screen.
  - Match with goal: As the goal is not listed on screen, the user will assume it is behind the extra options menu.
  - Feedback: The feedback to this would be a menu with the option "Stream Video" appearing.
- Press "Stream Video".
  - Current goal: To start recording a video.
  - Accessibility of control: Very accessible. Touchable button with a textual label that matches goal. Figure 2.1b shows this screen.
  - Match with goal: Exact match.



(a) Adding a tag (b) Adding another tag

Figure 4.4: Images of adding tags to a video stream

- Feedback: The next activity is immediately launched.

- Type in each tag you want the video to have, pressing enter after each one.
  - Current goal: To start recording a video.
  - Accessibility of control: Not very accessible. A keyboard and input box appears with "New video attribute" above it, but pressing 'enter' to add a tag is not made clear, and the actual reason for all this is not explained to the user either. Figure 4.4 shows this screen.
  - Match with goal: Complete mismatch. The user wanted to record a video, but is being presented with an input box for adding tags to the video they are about to record.
  - Feedback: Once a tag has been added, it is immediately added to the list on screen, giving feedback that the user's input has made a difference.

- Press the back button to switch to the camera preview.
  - Current goal: To finally start recording a video after adding the necessary tags.
  - Accessibility of control: Pressing 'back' to get to the next stage is not clear and wouldn't make sense to a notional user.
  - Match with goal: Again, needing to press the 'back' button to get to the next stage is not clear and doesn't match up with what a user would expect.
  - Feedback: The camera preview is immediately launched.

- Press the camera button on screen to begin recording.
  - Current goal: To find the control that begins video recording.
  - Accessibility of control: This is very accessible, as a button is displayed on screen, with a helpful camera image to let the user know what it does. Figure 3.3 shows this screen.
  - Match with goal: Matches well in that the icon matches up with the current goal. However, it is not made clear that video recording has not already started.
  - Feedback: The camera button changes colour to show it has been pressed, but no other feedback otherwise.
- Either press the camera button again, or press ‘back’, to stop recording.
  - Current goal: To stop the streaming of video.
  - Accessibility of control: This is just as accessible as the previous section, with the same clear icon used to stop the stream, or the ‘back’ button, which makes sense to stop the stream.
  - Match with goal: Matches well with the same button being used in a kind of easy-to-understand on-off switch.
  - Feedback: The video recording stops and returns to the main menu.

This cognitive walkthrough shows that there are some issues with a normal user’s operation with the application that could be addressed with the application, namely making it clearer what each action is doing and providing the user more straightforward controls. The user interface makes use of the Gestalt laws of perception[12], including the law of similarity and the law of continuity with the list of video streams, showing good graphic design for the user interface.

Chapter 5

Conclusions

I believe that the project has been very successful in providing a way of broadcasting video to others, utilising the local peer-to-peer network. As Haggle only exists as a research architecture at the moment, its installation process is relatively complex for a notional user, though I do believe it outlines the great possibilities of rich content creation and sharing between friends and strangers alike through their smartphones.

I have also tremendously enjoyed learning about many new facets of computer science, both in the theories implemented regarding ad-hoc networks and in new programmatic techniques concerning the Android platform and in data transmission protocols. Building on a research project was both rewarding and challenging, thanks to the novel and interesting idea for the architecture and the mostly undocumented API for that architecture.

5.1 Lessons learnt

If I could start the project all over again, there would be a certain number of changes I would make to make the process much smoother. I still think Android was a good choice for a project such as this, but I would use a more feature-rich and stable version rather than opting for an earlier release.

While I have tested the application, I would always try to cover more ground in this testing. This is because in a networked application such as this, there are many, many things that could go wrong that one wouldn't even have thought of before they do.

I think trying the application out with real users would be important to do if I could start the project over again, especially from a very early stage of the project with prototypes. This would also ensure a smooth experience for the user by getting feedback on problem areas in the graphical user interface and in the general layout of the application.

5.2 Future work

While the application does fulfill the success criteria, there are further extensions that could be made to extend its functionality.

Geolocation through cellular networks or GPS could be used to find content in the local area, directional data using internal compasses could be used to show direction in tandem with location data, and integration with social networks like Facebook or Google Latitude would allow users to find content from their friends and offer more details about the content creators from their social network profile.

If an efficient program to merge video clips together could be created or brought in, this could be used to save selected video streams to the Android video gallery. Also, as the Haggle service runs in the background, a custom persistent service would show a notification if any new video streams matched interests from anywhere in the operating system and without needing the application itself to be open.

Bibliography

[1] 3GPP. *Mandatory Speech Codec speech processing functions; Adaptive Multi-Rate (AMR) speech codec; Transcoding functions*, June 1999.

[2] 3GPP. *Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; International Mobile Equipment Identities (IMEI)*, June 2002.

[3] B. Cohen. Incentives build robustness in BitTorrent, May 2003.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[5] C. P. Fry and M. K. Reiter. Really truly trackerless BitTorrent. Technical Report CMU-CS-06-148, School of Computer Science, Carnegie Mellon University, Aug. 2006.

[6] IEEE Computer Society. *IEEE802: Standard for Local and Metropolitan Area Networks: Overview and Architecture*, Feb. 2002.

[7] ITU-T. *Recommendation H.263: Video coding for low bit rate communication*, Jan. 2005.

[8] A. Kaheel, M. El-Saban, M. Refaat, and M. Ezz. Mobicast: a system for collaborative event casting using mobile phones. In *MUM '09: Proceedings of the 8th International Conference on Mobile and Ubiquitous Multimedia*, pages 1–8. ACM, 2009.

[9] E. Nordström, P. Gunningberg, and C. Rohner. A search-based network architecture for mobile devices. Technical Report 2009-003, Department of Information Technology, Uppsala University, Jan. 2009.

[10] NPD Group. *Android Shakes Up U.S. Smartphone Market*, May 2010.

[11] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks (1976)*, 7(6):395–405, Dec. 1983.

[12] R. J. Sternberg. *Cognitive Psychology.* Wadsworth Publishing Co. Inc., 3rd edition, Sept. 2002.

Appendix A

Original Project Proposal

P2P User Contributed Live Video

Everyone has heard of Youtube, but it takes a lot of resources to upload something to it. Most smartphones have a decent camera (e.g. Android HTC G2 phones have 5MP video capable). What if we could stream live video, in an ad-hoc mobile net (Wi-Fi, Bluetooth) rather than via the cellular net—people could subscribe by tags or by human source (e.g. friend on Facebook or Latitude).

Essentially, this would be an application-layer multicast overlay, that would use intermediate phones (one in contact via whatever wireless local hop was available and that are part of the system) to cache copies until the next "hop" towards a subscriber is availabile. Subscriptions would percolate through the net (indeed, subscriptions might be distributed using the cellular net since they will be small items) whereas video chunks or files will percolate "towards" subscribers, from "providers" in a dynamically changing mesh. The idea of doing it this way is to avoid the large cost of uploading (Indeed, the 3G nets might not be capable of dealing with the load), whereas much of the interest may in fact be relatively local (think sports arena, rock festival, traffic jam etc).

Could be useful for giving live view of traffic ahead on a road or of unfamiliar scene, or for giving multiple crowd views of a sports event.

The Haggle Project has some nice infrastructure software for this sort of application, including coping with disconnections (cache the content on a device for later pick up, or send it into the cloud).

Appendix B

Main activity layout XML

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TabWidget
            android:id="@android:id/tabs"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" />
        <FrameLayout
            android:id="@android:id/tabcontent"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent">
            <ListView
                android:id="@+id/stream_list"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" />
            <ListView
                android:id="@+id/neighbor_list"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" />
            <ListView
                android:id="@+id/settings_list"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" />
        </FrameLayout>
    </LinearLayout>
</TabHost>
```

Appendix C

Data object sequencing algorithm

```
// expected is a HashSet of sequence numbers that haven't arrived but should have.
// toFire is an ordered TreeSet of sequence numbers that have arrived before expected
// ones and are waiting to be sent to the StreamViewer.
// lastFiredNumber is an Integer, initialised to (Counter.INITIAL_NUMBER − 1).
// It is the last sequence number sent to the StreamViewer.

private void sequenceChunks(Integer seqNumber, String filepath) {

    if (seqNumber == (lastFiredNumber + 1)) {
        expected.remove(seqNumber);
        if (isBeingViewed) {
            fireDataObject(seqNumber, filepath);
        }
        int current = seqNumber;
        for (Iterator<Integer> it = toFire.iterator(); it.hasNext(); ) {
            Integer toFireNumber = it.next();
            if (toFireNumber == (current + 1)) {
                if (isBeingViewed) {
                    fireDataObject(toFireNumber, chunks.get(toFireNumber));
                }
                it.remove();
                current++;
            } else {
                break;
            }
        }
        lastFiredNumber = current;
    } else if (seqNumber > (lastFiredNumber + 1)) {
        expected.remove(seqNumber);
        toFire.add(seqNumber);
        // Need to work out expected packets that haven't arrived.
        for (int i = (lastFiredNumber + 1); i < seqNumber; i++) {
            if (!toFire.contains(i) && !expected.contains(i)) {
                expected.add(i);
            }
        }
        while (toFire.size() > MAX_LENGTH_OF_TOFIRE_LIST) {
            Integer firstDObj = toFire.first();
            toFire.remove(firstDObj);
            if (isBeingViewed) {
                fireDataObject(firstDObj, chunks.get(firstDObj));
            }
            int current = firstDObj;
            for (Iterator<Integer> it = toFire.iterator(); it.hasNext(); ) {
                Integer toFireNumber = it.next();
                if (toFireNumber == (current + 1)) {
                    if (isBeingViewed) {
                        fireDataObject(toFireNumber, chunks.get(toFireNumber));
                    }
                    it.remove();
                    current++;
                } else {
                    break;
                }
            }
            lastFiredNumber = current;
        }
    } else {
        // seqNumber < (lastFiredNumber + 1)
        // We can get into the situation where we receive sequence numbers less
        // than the last fired number when the toFire tree gets too big and we
        // send some, only to receive the earlier ones after this. In this case,
        // we can only drop the packet and continue.
        expected.remove(seqNumber);
    }
}
```

Appendix D

Project Proposal

Introduction and Description of the Work

Recently, video as a means of sharing and collaborating has become much more prevalent—many people use video sharing websites like YouTube to share content. Further to this, the majority of modern mobile phones have relatively high-resolution cameras that are capable of shooting video regardless of location.

However, the user is still tethered to uploading this to a computer, then uploading the video file. 3G networks can be used, but they have limited capacity and can still be financially expensive. Infrastructure mode Wi-Fi can be used, but this model requires a user to be within range of an access point and is at odds with the basic notion of a mobile phone—it's not very mobile!

My application utilises the Haggle Project[1], which is a network architecture that forwards data from publishers to subscribers based on the subscriber's set of interests. This can occur over Bluetooth and ad-hoc wireless between phones. In this way, a publisher can record a video and add tags that describe it, and subscribers who have declared an interest in one of the videos' tags will receive the video over peer-to-peer connections from other mobile phones.

This could be useful in a number of situations. For instance, alternate views of a sports event could be offered by members of the audience over the P2P network, or a group of friends on a night out could automatically share recorded videos without needing to do it manually. Extensions to the project include using geolocation to find content in the local area, or being able to find content from social networks like Facebook or Google Latitude.

[1]`http://www.haggleproject.org/`

Resources Required

- My own personal laptop.
  Spec: 1.6GHz CPU, 2GB RAM, 160GB HDD, Linux OS

  I will be using my own computer to develop the project, as the Android SDK is large is size and will restrict my PWF filespace. For back-up purposes, I will use a version control system (Subversion), which will be stored on my local machine. I will also store this in a Dropbox folder, which automatically backs up all files in the repository to the cloud. I will schedule daily incremental backups and weekly full backups to both my SRCF account space, and my PWF account space.

  This is available.

- Three Android mobile phones.

  I will need three Android mobiles to test communication with both forms of P2P communication available and to test the delegate forwarding aspect of the Haggle system.

  These are being made available by my project supervisor to share between another Part II project and myself. They will be available from 16th November 2009 (at the latest), and each will be available to test on in week-long blocks. I will schedule my on-device testing so that it does not massively conflict with the other project's use of the devices.

Starting Point

I have some experience working with the Google Android platform from non-professional coding over the last few months. I know about the basic structures that make up an Android application. Android uses Java as its main development programming language, which I have experience in through the last two years of Tripos courses.

Substance and Structure of the Project

My project will build an Android application which is based on the Haggle architecture for Google Android. This will involve defining the exact features the application will have into a menu structure, then designing the user interface for the application. These will include video sending, defining tags to receive video from, video selection when tagged videos do appear and defining Facebook integration for getting video from socially networked phones in range.

The UI will first of all need to conform to the look and feel of the Android OS itself, so that users do not feel a disconnect when they access the application. The landing page of the application should be clutter-free, and should have large, easily identifiable icons for particular features of the application.

My application will need to find an efficient way to send video files over Haggle, which will involve finding suitable video compression algorithms to encode the video in, and breaking up the video files in sufficient ways that playing the video while it is still being downloaded is possible. The application should then be able to decode and play this video with full control on volume and position of the video, depending on what has been downloaded already. It will also need to utilise the metadata Haggle uses to spread the video properly between nodes, with regards to a video's tags (or Facebook handle and/or location depending on extensions).

Success Criterion

- To be able to send a video file from one phone to another using the system.
- To be able to play this video file while part of it is still being received.
- To be able to tag video with relevant interests for others to receive.
- To be able to select between received videos if more than one match at any given time.

Timetable and Milestones

From Monday 26th October 2009

Preparatory work. Research the structure of Android applications and Android's SDK in relation to video streaming and UI design. Investigate the difference in Android phones with regards to accepted video parameters.
Milestones: Understand Android's video capabilities and application design. Find out optimal video parameters for good mobile cross-compatibility.

From Monday 9th November 2009

Prepatory work. Research Haggle's architecture and how Haggle interoperates with Android. Define exact features and menu structure of application and start

designing user interface for these features, with regards to user interface mentioned in *Substance and Structure of the Project*.
Milestones: Understand Haggle's architecture on Android. Complete design of user interface.

From Monday 23rd November 2009

Start development of Android application using the Android emulator. Ensure application is fully compliant with Android events like incoming calls, text messages, orientation of phone etc. Test application on actual Android phone to check that it works in the real world.
Milestones: Have majority of main application without features developed and tested. Test application on real Android phone.

Friday 4th December 2009 to Sunday 10th January 2010 (Christmas vacation)

Implement stored video file sharing/streaming over Haggle to other Android phones. Will need to use emulator to do limited testing while not able to access hardware. Get feedback on designed user interface from potential users using a mocked up interface.
Milestones: Have video sharing feature implemented and tested using the emulator. Have some user feedback on UI for possible revision.

From Monday 11th January 2010

Test and debug code implemented over Christmas vacation on actual Android mobiles, including data transfer over one and two hops. Start preparations for project report.
Milestones: Have implemented code tested on actual Android devices. Have started work on project report.

From Monday 25th January 2010

Continue with preparation for project report. Investigate and implement live (not recorded) video streaming over Haggle to other Android mobiles.

Milestones: Complete and present finished project report. Investigated and part-implemented live video streaming.

From Monday 8th February 2010

Test and debug code implemented concerning live video streaming. Begin concentrating on writing dissertation.
Milestones: Complete implementation and testing of live video streaming. Have started first draft of dissertation.

From Monday 22nd February 2010

Continue writing dissertation, especially implementation section. Attempt to implement and test feature extensions to project.
Milestones: Continued writing dissertation. Implemented and tested at least some of the project extensions.

Friday 12th March 2010 to Sunday 18th April 2010 (Easter vacation)

Test and evaluate speed and reliability of application for video sharing, both functional (effectiveness of user interface) and non-functional (speed/reliability testing and battery life). Continue writing dissertation, especially evaluation section.
Milestones: Evaluated application design and benchmarks. Continued writing dissertation.

From Monday 19th April 2010

Continue writing dissertation, especially the conclusion. Submit dissertation before deadline.
Milestones: Finish writing the dissertation and submit before deadline.