Fluid Photo Browser with NFC-based Payment and Printed Photo Pickup

David Thomas (dtho@itu.dk)
Egil Hansen (ekri@itu.dk)
Jonas Rune Jensen (jruj@itu.dk)

April 20, 2012

1 Introduction

The task for this assignment was to develop a tabletop app for a photo print shop, featuring transfer of photos between the tabletop and customers' smartphones of as well as payment for printed photos using Near-Field Communication (NFC). The tabletop app should allow editing of the photos before printing them and sending them back to the customer.

2 Background

2.1 Tabletop computing

Tabletop computing generally covers the use of large, horizontally-mounted touch screens. As described in [2], this encourages usage which differs from the white board-inspired use of wall-mounted displays. Tabletop computers are often surrounded by people interacting from different angles.

3 The System

The system we have implemented consists of the parts shown in figure 1: the tabletop app itself, an Android app for the customers, a service on Google App Engine to facilitate communication between the tabletop app and the Android app, and finally a helper program to connect the tabletop to an NFC reader.

To summarize the scenario, the idea is that a customer enters a print shop and uploads one or more pictures to the shop's tabletop computer from an Android phone. The pictures are edited on the tabletop and sent back to the phone, and printouts of the pictures are paid for using an NFC-based payment system.

Each part is described in details in the following sections. The source code is available at `https://github.com/jeffton/Pervasive-Assignment-2`.

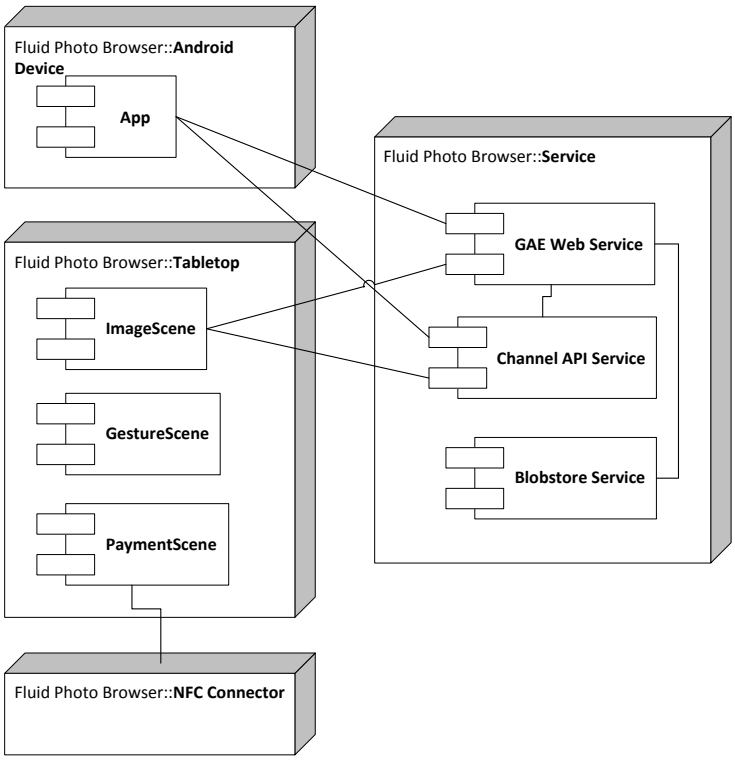

Figure 1: Deployment diagram for the system.

4 Tabletop App

MT4J [1] is a Java-based framework for multitouch applications. MT4J apps consist of a number of scenes, of which ours has three. `ImageScene`, shown in figure 2, displays images as they are uploaded from the phone and provides options for editing. Editing is kept within the same scene for simplicity and ease of use – the last-tapped image is highlighted and will be the target of any applied effects. For our prototype, we have implemented a single effect: cropping.

After editing, the images can be sent back to the phone – either by applying a custom gesture, or by tapping the upload button. The custom gesture can be defined by tapping 'New gesture', which leads to the `GestureScene`. Here, the user can define a gesture by recording a movement as a list of coordinates.

Finally, the `PaymentScene` handles the payment flow. When an NFC card is connected, the balance is shown along with the number of pictures corresponding to the ID stored on the card. When accepting the payment, the balance on the card is reduced, and the user is returned to the image scene where the purchased

[1] `http://mt4j.org`

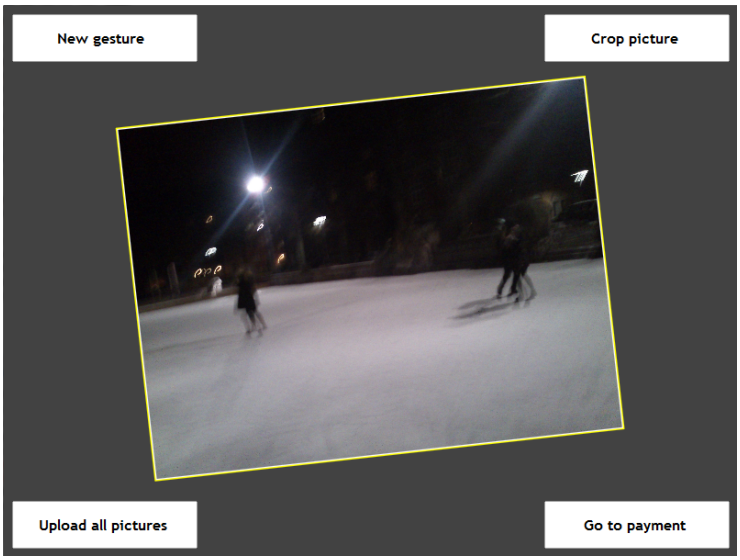

Figure 2: `ImageScene` in the Tabletop App.

images have been removed.

4.1 Image Transfer

All transfers of images between the tabletop app and the Android app happen through the Google App Engine service described in section 6. Since the tabletop app subscribes to notifications from the service, images appear on the tabletop almost instantly after being uploaded from the Android app.

5 Android App

The android application has two functions, upload selected pictures to the tabletop and downloading the edited pictures.

For selecting pictures we are using the Androids built in intent to select data on the device. In our case it is specified to be pictures. By using the built in intent, there is the disadvantage that we can only select one picture to upload at a time, which means that the user has to select upload button again for each picture. For a better user experience, we could build a custom intent where the user is able to select more than one picture.

When the user has selected a picture, it is uploaded to our service on Google App Engine, and then downloaded by the tabletop application.

When the tabletop is done editing the picture and uploads it to the service on Google App Engine, the application receives a push notification with id of

the picture, and starts downloading it. We save the downloaded picture in a folder called PicPush so we don't delete older pictures.

6 Service on Google App Engine

The photo relay service[2] we built is hosted on Google App Engine[3] and gives us a few advantages over a hardcoded static IP during development, at the cost of a little more work, and it gave us a chance to build a system that is a little closer in design to a system one would build in the wild.

The main benefit is the loose coupling between the Android and tabletop app. They do not need to know about each other explicitly, and this allows the system to scale very easily, e.g. support more than one tabletop. Loose coupling also enables us to develop each component mostly independent of each other since each implementation just has to adhere to the agreed upon web service interface.

On the protocol level, using Google App Engine also means using HTTP(S) for transport of the photographs between it, the Android app, and the Tabletop app. Normally, HTTP only supports pulling for resources, but luckily Google has a solution for this as well called The Channel API[4]. The Channel API allows us to push notification from the photo relay service to the Android and Tabletop app when something of interest happens, such as a new photo being uploaded. With push notifications we are able to get very seamless transfer of photos between the apps.

The biggest downside of using the photo relay service is that the transfer time is increased, but we feel that the added transfer time is a fair tradeoff considering the benefits mentioned above.

6.1 Photo Relay API

Uploading a photograph To upload a photograph, a client must first calls `/getUploadUrl`. A unique upload URL is required by the Google App Engine cloud storage system. With the new upload URL the client uses regular `HTTP POST` to upload photo(s) along with an associated NFC ID and the source of the upload, ie. Android or tabletop.

Downloading The service provides two ways of querying for photographs, `/getPhotoUrls` and `getPhotoUrlsByNfcId?nfcid=<NFC ID>`. The latter allows clients to filter photographs by associated NFC ID. Querying either will result in a JSON array like to the example below:

```
[
  {
```

[2] `http://fluid-photos-at-itu.appspot.com/`
[3] `https://appengine.google.com/`
[4] `https://developers.google.com/appengine/docs/java/channel/`

```
    "id":"AMIfv...",
    "source":"android",
    "filename":"DSC_0072.JPG",
    "nfcId":"123",
    "uploadedOn":1334853268514
  },
  {
    "id":"4VYNz...",
    "source":"android",
    "filename":"DSC_0054.JPG",
    "nfcId":"123",
    "uploadedOn":1334853971989
  }
]
```

With the ID of each photo, the client can now call `/getPhotoById?id=<photo id>` to get the raw photo data.

Deleting a photograph It is also possible to delete a photo stored in the cloud service. This is done via a call to `/deletePhotoById?id=<photo id>` with the ID of the photo.

Push notifications The clients must acquire a security token before they can subscribe to receive push notifications from the cloud service. This is done through a `HTTP GET` call to the `/token` service call. Once a client has a token, it can use a Google App Engine Channel Client framework to do the heavy lifting of keeping the communication channel open. In our case, we used the Tom Parker's Java AppEngine Channel Client framework[5].

7 NFC Payment

NFC technology is used for identification with the tabletop app as well as for payment. In our proof of concept, the payment takes place directly in the tabletop app. As we are lacking an NFC-enabled Android phone, a MiFare Classic NFC-card is used. We store information on blocks 1 and 2 on the card: block 1 contains a header and an ID, and the block 2 contains the amount of available coins.

Communication with the card is implemented in a C# helper application using the C# API in the PC/SC SDK from SpringCard [6]. The tabletop app starts this helper application in a separate process and communicates through standard input and output streams.

[5] `http://masl.cis.gvsu.edu/2012/01/31/java-client-for-appengine-channels/`
[6] `http://www.springcard.com/download/sdks.html`

8 Limitations

8.1 Lack of Security in the Payment System

No attempts have been made to secure the payment system, so cloning a card as well as changing the amount stored on one is straightforward. Since the built-in encryption on MiFare Classic cards is insecure [1], we should encrypt our data before storing it on the card. This would prevent tampering but not cloning. With an online system such as this one, security could be increased by storing the amount on a server rather than on the card itself. While this would not prevent cloning, it would prevent spending the same coins twice. To avoid the cloning problem completely, we could use one of the more secure - but more expensive - successors to the MiFare Classic card.

8.2 Push Notifications on Android

Instead of using the Channel API to get push notification from the Google App Engine, we could have used the Android Cloud to Device Messaging Framework[7]. This framework pushes messages to the device using its connection to Google services. The most useful part, is that our app would not have to be open, as the device would automatically open the app and a give it the message to process. But we are not able to use the system with the tabletop, so we chose the Channel API as it works on both systems, and potential other new systems.

8.3 Choice of NFC SDK

The license of the SpringCard SDK restricts distribution to applications that use card readers purchased through SpringCard. Also, the C# helper application limits the platform independence of our tabletop app. Nonetheless, the SDK works for our proof of concept, and the NFC commands would be the same for any SDK.

9 Conclusion

For this assignment, we have implemented a tabletop app for a photo print shop along with an Android app. The use of an intermediate Google App Engine service helped us by mitigating connectivity issues and by making it easier to develop the remaining parts of the system independently.

References

[1] N. T. Courtois. The dark side of security by obscurity and cloning mifare classic rail and building passes anywhere, anytime. Cryptology ePrint Archive, Report 2009/137, 2009. `http://eprint.iacr.org/`.

[7] `https://developers.google.com/android/c2dm/`

[2] C. Mller-Tomfelde, A. Kunz, and M. Fjeld. From tablesystem to tabletop: Integrating technology into interactive surfaces. In C. Mller-Tomfelde, editor, *Tabletops - Horizontal Interactive Displays*, HumanComputer Interaction Series, pages 51–69. Springer London, 2010. 10.1007/978-1-84996-113-4_3.