I. Description:

Our project, called PicShare, consists of an iOS photo sharing service. This service allows a user to share a photo album using pictures from his/her device, and invite his friends to view them. These friends will also be able to upload their own photos to the album, as well as comment and “like” them. In essence, our app is a kind of “lightweight” Facebook or Instagram used exclusively for sharing photos. When a user creates an album, they choose pictures to put in this album and choose friends to share the album with. Then, the user’s friends will receive a notification that a new album has been shared with them, and they will download the album and other metadata from the server. If a user’s friend does not have the app already, the plan is for he/she to receive an email inviting him/her to join the service and download the app. Then, the album will be downloaded from the server, and this new user will be able to view the pictures and share his/her own.

Although many photo-sharing services already exist, we believe there is still a market for our app. When using Facebook or Instagram, a user must create an entire profile; since everyone can view your profile, users feel the need to invest a lot of time personalizing their profile and making it look good. Further, a Facebook or Instagram user does not have complete control over who can see their photos. Since users of these services often have hundreds, if not thousands, of friends, it is almost impossible to restrict permissions for certain photos to just a select few. In PicShare, however, a user needs only an email and a password (and an iOS device) to use the app. This allows for users to join PicShare without investing too much time with other functions. Further, PicShare users have complete control over the privacy of their photos and albums, as they will only be shared with others when the user adds friends to an album. PicShare also contrasts with Apple’s new Photo Stream service within the photos app, which also allows users to share pictures with friends. However, Photo Steam does not allow for others to also contribute to these albums, while PicShare lets other users upload their own photos, allowing for collaboration. For example, if Bob travels across Europe with Jim and Jenny, and all three take pictures on their iOS devices, they can use PicShare to combine all their pictures into a single album, which the three of them (and only the three of them) can see. We believe PicShare is a very convenient service, and the typical use case would be similar to what Bob, Jim and Jenny experienced.
II. Design:

A. UI (iOS) Architecture

1. Introduction

The iOS front-end app leverages several iOS technologies and design patterns in order to achieve the desired behavior of our app. These technologies include Core Data and NSUserDefaults to manage the local persistent app data and preferences, table views and collection views to organize onscreen content and recycle views for scrolling performance across large screens of thumbnails, the AssetsLibrary framework to access the device's media assets, and Apple Push Notification Service (APNS). Additionally, the app puts these all together using an MVC architecture to maintain database consistency, adapt to changes, and streamline the app’s memory usage as well as the 3rd party library ASIHTTPRequest, which simplifies the sending of HTTP POST and GET requests. Below, we briefly outline the aforementioned components.

2. Core Data

The app uses Apple’s Core Data framework to manage the internal database tables and relations of the user’s data. Core Data adds the ability to create Objective-C classes that are “windows” into the database (which is stored as a SQLite database by default). There are several important classes in the Core Data framework, but the most important for our purposes are the NSManagedObjectContext, NSManagedObject, and NSFetchedResultsController. The managed object context class can be thought of as a “scratch pad” to which our app writes database changes and stores them in volatile memory for write speed before committing them at a later time (for example at the end of a batch update, like downloading several large images, or when the app is sent to the background). The managed object is a class that represents a row in a Core Data table and is subclassed to create more specific tables. Instances of our manage objects represent individual rows in the table. The managed object context is asked for existing rows or to create new ones and we can work with them as object instances. Finally, the most important class is probably the fetched results controller. This is a class that responds to updates to the database. It can be set to maintain a collection of database entries that satisfy a given predicate and send notifications to the view controllers when this collection is updated. This allows the view controllers to update the data that is displayed to the user in real time as it is updated. In addition to core data, NSUserDefaults is used to create a persistent dictionary of user and device information.
3. **ViewControllers**

The app contains several basic view controllers for simple static screens as well as the more complicated UITableViewController and UICollectionViewController for screens that present organized and dynamic data. The app's main screen is a table view with a fetched results controller to maintain its data. The table view controller then updates when the fetched results controller receives changes. The views that allow for viewing of photos in an album are collection views. These also have their own fetched results controllers with predicates to manage only rows of image metadata that belong to the album. These views define table and collection cells that are recycled as they are scrolled off screen to streamline performance. In addition, the local database is organized in such a way that it separates image data from metadata, so the view controllers only need to maintain metadata in memory to maintain good scrolling instead of requesting full images from the database to update the user's view of potentially hundreds of photos. The interface is enhanced in some cases by subtle UIView animations and the use of scroll views to zoom and view full-screen images.

4. **AssetsLibrary**

The AssetsLibrary framework is accessed by our app to get access to the device's local media assets. To use the assets library, the app must maintain a single instance of the library throughout the app's lifespan. It achieves this by using a singleton class to access and make requests to the library. The library is used to create custom pickers for choosing photos to add to our albums. The pickers also utilize the collection view recycling above so that scroll performance isn't hindered and memory usage is minimized even with a library of over 1,500 photos in one of our test devices.

5. **Handling Online Data**

To interact with data on our webserver, the app uses the 3rd party ASIHTTPRequest library, which is a wrapper for the more complicated built-in url request classes and which allows requests to be easily backgrounded using either block or delegate design. The app also uses grand central dispatch (GCD) to manage background threads. One of the most difficult design decisions was how to maintain database consistency and failure cases when interacting with the server and waiting indeterminate periods for data requests to process. During large updates, we block the UI with a popup activity indicator to let the user know that photos are being downloaded, but during POST requests to the server, we simply perform the requests in the background after updating the local
database without blocking the UI. The app is designed to receive a complete XML skeleton of the user-specific data in the server’s “master” database and then parse this skeleton locally to determine whether it needs to delete images or download new images or update settings. This design allows the app to work with or without push notifications, which are unreliable and only trigger updates that will eventually occur anyway. From the app’s perspective, all that push notifications do is tell the app to request and parse the main XML skeleton.

B. Server-Side Architecture

PicShare follows the Client-Server model. Each client is an iOS device that has downloaded the app. The data that is created in the client is shared with a central server. In our case, we used a PHP scripted server and a MySQL database, both locally set up on Daniel’s MacBook.

The server receives update POSTs from the clients using variables and values in the body. There is a “command” variable that explains the type of POST it is (such as create_album, add_pic, delete_buddy, etc.). Using this, the server runs the appropriate script, queries the database as needed, and sends a PUSH notification to other buddies involved in the corresponding album. Once completed, the server responds to the client with a GET response. If there was an error in the script, the server responds with an error message. Otherwise, the server responds with an acknowledgement along with information of the update (such as server generated picture_ID/album_ID). When the client sends an “update command”, the server generates an XML skeleton GET response containing all the metadata that the client should have. As such, the server is regarded as the primary database and trumps the client. The central database holds a logical layer and a picture storage directory. The logical layer contains several tables: user_info, album_info, pic_info, shared_album_info (users connected to each album). The database holds a global database with all the information across all the clients.

C. Major Design Points

1. Client-Server Model

We use a centralized server with a “master copy” of all data and metadata. Prior to our implementation, we debated a peer-to-peer design versus a client-server design before making a decision. A client-server scheme just seemed much easier to implement, and we felt that a peer-to-peer design didn’t offer enough advantages to offset its increased complexity. Our backend server and database are written in PHP and MySQL, and the iOS app communicates with the server using HTTP POST and GET requests through our PHP API.
2. **Separation of Control and Data**

One design point that was central to our implementation was the separation of control and data. When dealing with data as large as photos (and especially when dealing with many, many photos), it is not always possible to simply send across all data that the user needs. So, we often use metadata rather than the actual data, in order to limit the data being sent to only what is absolutely necessary. One key way we implement this design point is through the server sending an XML metadata skeleton to the app, and then the app can cross-check this metadata with its own data, and request what it needs. Further, the local database on the app separates metadata from the actual image data, another way that the control and data are separated to improve efficiency.

3. **Apple Push Notification Service (APNS)**

Although we have not implemented Apple’s push notification service yet, this is a feature we plan on adding in the future. Using APNS, the user would receive a push notification upon a change to an album that has been shared with the user. This would allow for users to be notified quickly and conveniently when a change has been made that requires their attention. However, APNS is not entirely reliable, so our app functions with or without push notifications through our update command using the XML metadata skeleton from the server.

4. **Database Consistency**

Since our app deals with large and potentially many photos, we have low bandwidth available relative to the size of the data. As a result, maintaining database consistency becomes extremely important, since it is possible for the local database on the user end and the server database to get out of sync. We combat this issue by always assuming that the server has the “master copy” of all data. Thus, when an app calls the update function (which occurs when the app starts up, every time the main page is accessed, and following a push notification) the server sends an XML metadata skeleton to the app of all data this user should have in its local database. If they are out of sync, the app can request whatever resources it needs, allowing us to maintain database consistency within PicShare.

III. **Future Work**

We would like to first set up a dedicated remote server (likely Heroku) and cloud database (likely Amazon S3) to maintain the central operations. We would add more robustness in the code and add more security to the data transfer (encryption on passwords and pictures). We would like to abstract the picture in the table such that only one copy of the picture needs to be stored when multiple albums refer to it. Since POSTs to the server happen in groups (adding many
pictures in an album), to avoid having the server send push notifications for each update POST the iOS device would send a "commit command" POST, indicating it is finished editing the album. Only then the server would send a single PUSH notifications to each buddy to sync up. We would like add more functionality as well. We would like to build the ability to comment/like on pictures and more preference options to allow album administrators to have more control of how its buddies can be involved in the album. Lastly, we would like to have a buddy list of all buddies the user has connected with instead of having to connect with an email address.