INFO401 Group Project

Iteration 3 Documentation

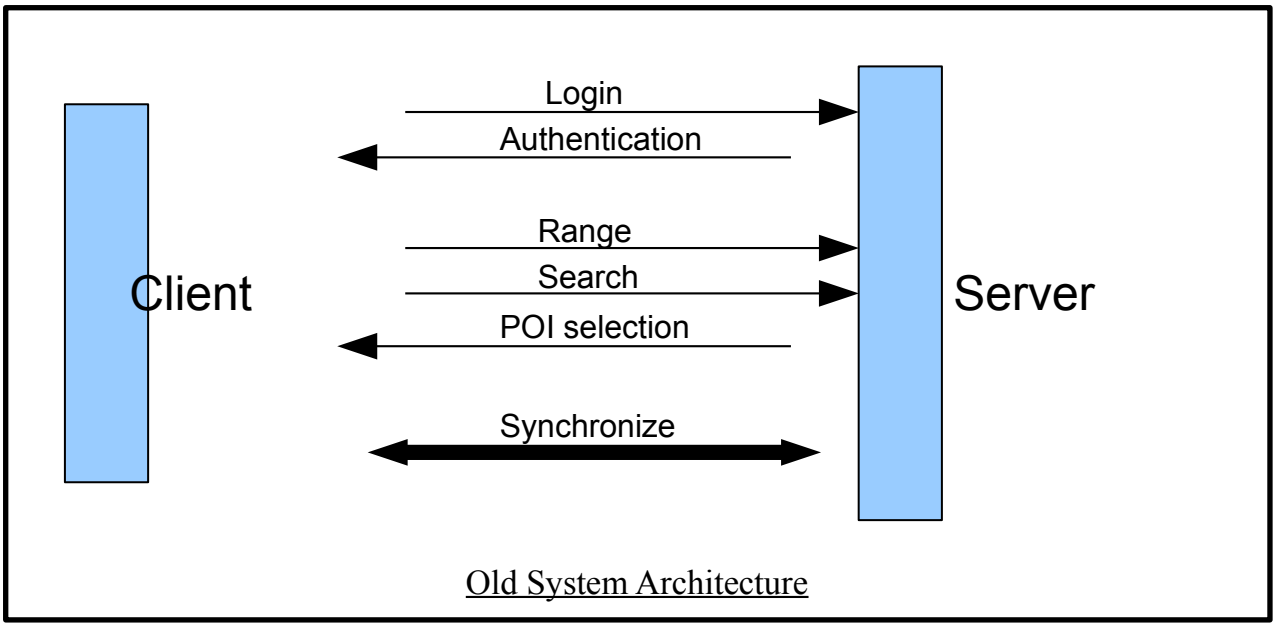
System Overview

The application is a mobile location based system which allows the users to visualise points of interest overlaid on a map. Users can register, edit and delete points of interest, as well as additional features such as add comments, tags, pictures and ratings. The system maintains a subset of all POI data on the phone, depending on the user's location. This database is updated periodically, depending on access, from a full dataset, held on a remote server.

General System Architecture

The system is a fusion of the last 2 iterations of the project, with the Point of Interest client side communicating and storing POIs on a server (iteration 2), while maintaining a local client database (iteration 1) of nearby POIs.

This project introduces the use of a single communal server for synchronizing POIs for multiple clients, the ability to search through current POIs, the ability to add photos, tags, and ratings to each POI, and ownership of POIs.

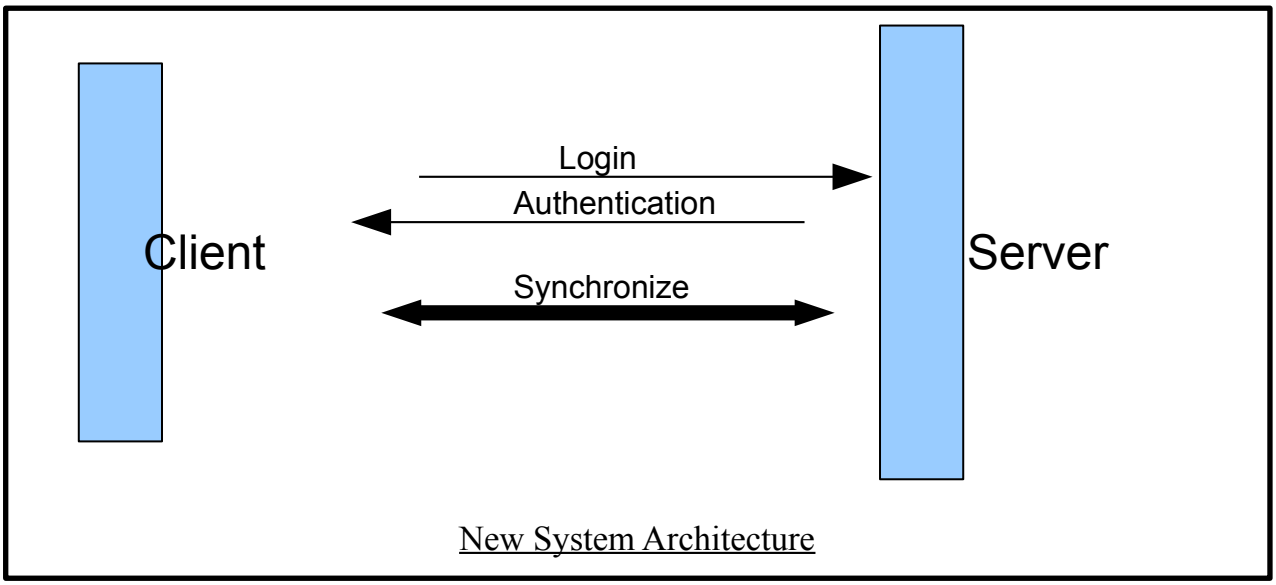


Old System Architecture

The original design for the project was to have a system where the client, and server contained similar functionality (as the client must be able to work offline). The POIs stored in the local database were determined by the POIs requested from server. In general, the client would request to sync POIs within a certain range (i.e. 100km) of the client.

When a search or range filter was applied, the client would send the request to the server, and the server would return the POIs that match the filtered request. If offline, the client would query the local database. The problem with this model is that too much information needed to pass between the client and the server, complicating the communication needed.

Furthermore, this model also further complicated the control of the local database (what to keep, and when to purge, etc).



New System Architecture

Therefore the system was modified to simplify the communications needs, and functionality of the client/server. The server is only responsible for login authentication, acting as a repository for all POIs, and synchronizing clients' databases.

Synchronization occurs by the client pushing all new POIs to the server, and then informing the server of its current position. The server then returns all POIs within a set range from the client. The range may be modified by the user from the main screen, from the context menu. There is a maximum of 100km for the range.

Searching and range filters are performed on the client side, and are only applied to the local database, while the local database is updated 'behind the scenes'. This architecture means that the client/server are successfully decoupled with (after initial login) the only connections between the two systems being the sync. Therefore the client can operate with full functionality independently of the server.

Ownership

Basic rules regarding the ownership of points of interest(POI) are as follows:
•Before logging into the system, both registered and unregistered users are only able to view the POIs and its contents (including pictures).
•Registered users may login to the system, which allows them to add comments and tags or rate the POIs. They are also able to edit or delete POIs which they have created.
•POIs may only be edited or deleted by their owner.
Unregistered users may create an account from the context menu.

Technical Login details

The process of logging into the system is:
•Users may select to login from the main screen context menu. This calls the onOptionsItemSelected method in POIMain.java, which will in this case create a UserUtilizes object to handle the login information.

•This class contains a method called Login, which creates a servlet to verify the user's details with the server.
•New users may register themselves using the register button from the same menu. The user will enter a username and password, which is handled by the register.java class. This checks with the server to see if the username is available, asking the user to retry if necessary, otherwise a new user is registered with the system.

After a user has successfully logged in, the user's state is saved in the preferences of the system. The details which will be saved are username and user id. Once the user logs out (using the same context menu used for login), the states of the user are cleared from the system.

Tagging

The tagging functionality is used to add common keyword tags to POIs as to attribute commonalities between POIs, allowing those POIs to be found when searching for that tag.

Tags can be added by any user when creating/editing a POI. When adding a tag, the user inputs a keyword in the input bar on the edit/create POI activity. All current tag keywords are used for auto-text complete for the input box, thus allowing the user to add a predefined keyword as a tag, or to make a new tag.

When the user clicks the 'Add' button, the system adds the words to an arraylist of tags for the POI. Upon POI creation/updating, the system looks at each word, seeing if the keyword is already in the system. If so, it joins the POI id and keyword id on the TAGS table. If the keyword is new, the word is first added to the KEY table, and then the POI id and keyword id is joined.

Searching

With the decoupling of the server and client, searching is only performed client side (the original system designs preformed searching server side, or on the client side if the server was unavailable). The search is performed on the local database, and then filtered for range before displaying the resulting point of interests.

One of the main concerns with searching the POI database are the resources (namely time) needed to index a database for a full-text-search (FTS). On the server side, which could potentially hold thousands of POIs, indexing the database, and maintaining this index can be resource intensive. On the client side, however, the local database represents only a small subset of all the server's POIs. Therefore, indexing and searching the database is less intensive than on the server.

FTS with SQLite is actually a relatively simple procedure. The main POI table was changed to a 'virtual' table which allows it to be FTSed. Every time a POI is created/deleted, the table's index is rebuilt, keeping the table index current.

To initiate a search, the 'search' home key is pressed, bringing up the search dialog. The user starts inputting a word. All tag keywords are used for auto-text complete for the input box,

thus allowing the user to use a predefined tag for searching.

Once the user has entered the a term, and starts the search, the system performs two searches on the database. First it finds if the term entered is a tag keyword, and retrieves all POIs associated with that tag. Then the system performs a FTS of the POI table of the database, in particular looking for the term in title and description of the POIs. The search is of a wild-card type, therefore searching for 'cat' returns POIs with 'cats', 'catgut', etc. All returned POIs are then displayed on the map. The POIs can then be viewed individually as normal.

A new search can be imitated at any time, and the range function can be applied to all search results (from the menu bar). While searching, POIs cannot be created, edited, or deleted.

To stop searching, the user presses the menu key, and pressed the 'stop searching button'. This returns the system to the previous state, showing all POIs within the local database, filtered by whichever range has been previously chosen.

Adding POI by address

This is a function which allows users which are logged in to add a POI by typing an accurate address. The process is:
1.The users open the context menu, and choose 'add item'.
2.They then enter the address in the pop up window.
3.The system then attempts to verify the address with the Google API. If results are found the list is returned to the user.
4.The user will then select the right choice, which will call IntentA.java to create the POI.
Associated functions are InputAddress(), SelectAddress(), FoundAddress(String).

Display Current Location

The users location is shown on the map. This is read as a GeoPoint from the GPS provider of the device, and then passed to the getPOIsFromCursor function, which draws it on the map. The class which handles tap events on the map, myItemizedOverlay.java, then has to check when a POI is tapped whether it is a POI or the users location. If it is the users location the system will show a message saying 'This is the current location”.

Change Filter Range

This function allows users to define how close POI needs to be before they are interested in them. Any POI outside the range will not be downloaded to the client from the server.
•Users will select a range from the context menu.
•This selection changes the value of a variable maintained in POIMain.java.

Picture Functionality

There are three main classes used for picture implementation. These are the GalleryActivity class, the Image-Adapter class and the Base46 class.

When viewing the contents of a POI, the user is able to access a gallery view, allowing all pictures associated with a POI to be viewed. Users can upload new, and delete irrelevant photos using the gallery view. A maximum of 10 pictures per POI has been implemented. Images are stored as Strings in the database. The gallery view is handled by the GalleryActivity class. When this activity begins, it opens the database and checks whether there are pictures associated with this POI already. If there is it will:
1. Create a new folder on the SD card to hold the images.
2. Read the entire image Strings from the database.
3. Convert these into images.
4. Populate the gallery view with the results.

To add a photo the GalleryActivity class calls the startcameraActivity method. The user is then able to take a new photo using the built in camera. When a picture is taken onActivityResult method will be called. This method will call the onPhotoTaken method to:
1. Resize the image down to 20kb.
2. Convert the image to a String.
3. Add the picture to the database.
4. Reload the gallery view with updated photo added.

Base46 class is used to convert a picture saved in SD card to string or other way around string from the local database to a picture. Android does not support base46 conversion; this functionality has been implemented manually to the system.

Synchronisation

The client/server architecture we used for this project requires a syncing operation to be carried out, to maintain that the two separate databases (client and server) remain consistent with one another. It is also needed as changes users make only occur on the local database until the syncing operation is carried out.

Synchronisation between the server and the client database is an ongoing process which must be managed at all times. For this reason a SyncAdapter is added to the system. Having a SyncAdapter component within the application allows some of the work involved in synchronisation to be handled by the Android OS. This is because once a SyncAdapter is in place it can be interacted with by the SyncManager. This is the same part of the Android operating system which is responsible for handling synchronisation between mail contacts, and so on. Even when a sync operation is explicitly called, the SyncManager determines when the applications syncing operation should be carried out.

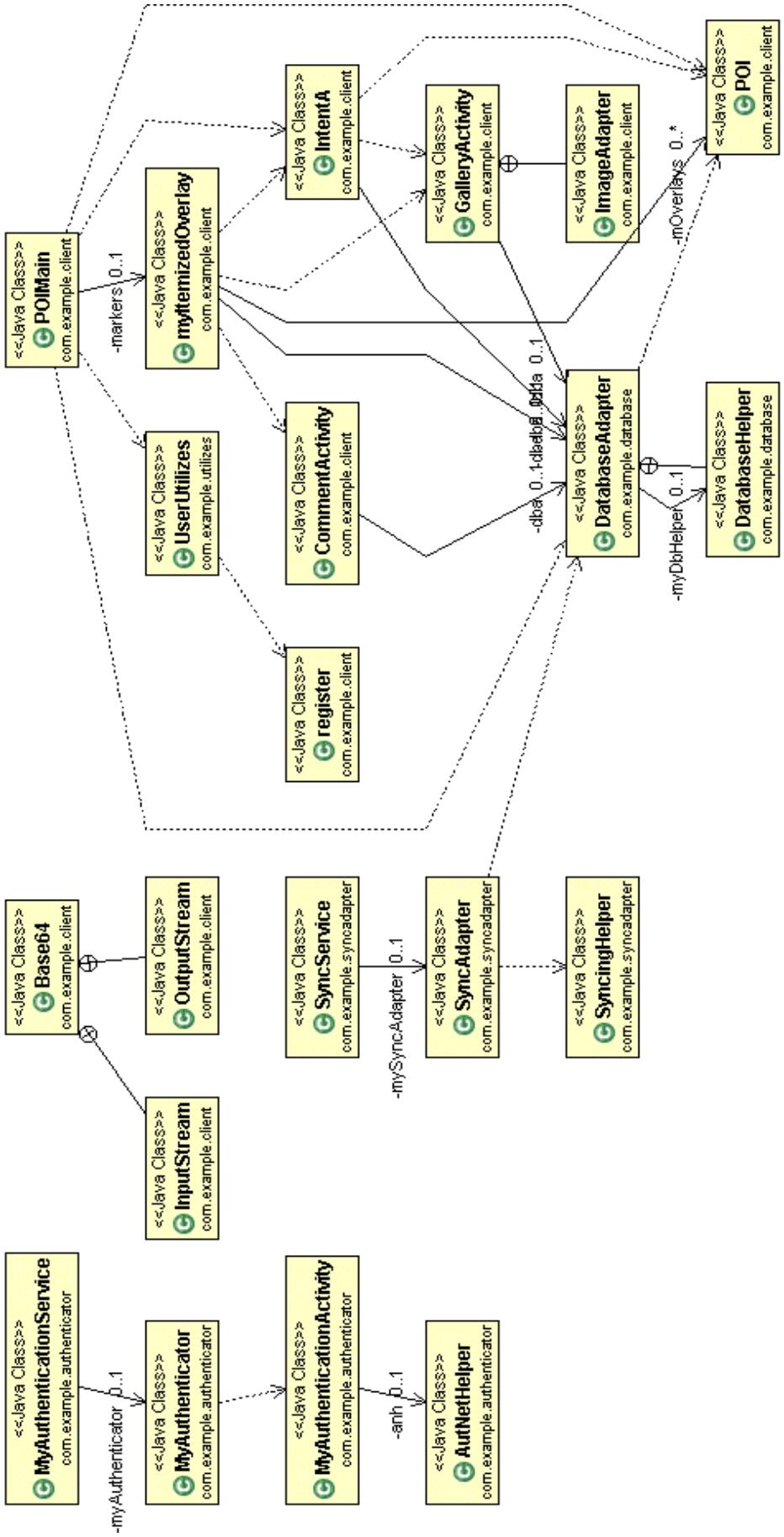
When it is time to sync it calls the onPerformSync(...) method in the SyncAdapter class. This method handles all the operations needed for syncing, completing the syncing operations on each table at a time. The operations needed to sync the databases are carried out by the SyncingHelper.java and DatabaseAdapter.java classes. The SyncingHelper first updates the server database of all changes made to the table in the local database. This is to ensure that a user's operations they have been performing on their local data will not be lost. The SyncingHelper class then fetches the POIs(or whatever table is currently being synced) from the server, as one long JSON string. This String is then handled by the

DatabaseAdapter class, which parses the String, removing any details it needs and inserting them into the local database. After this set of operations has taken place for all tables within the database, the database has been synced.

Diagrams

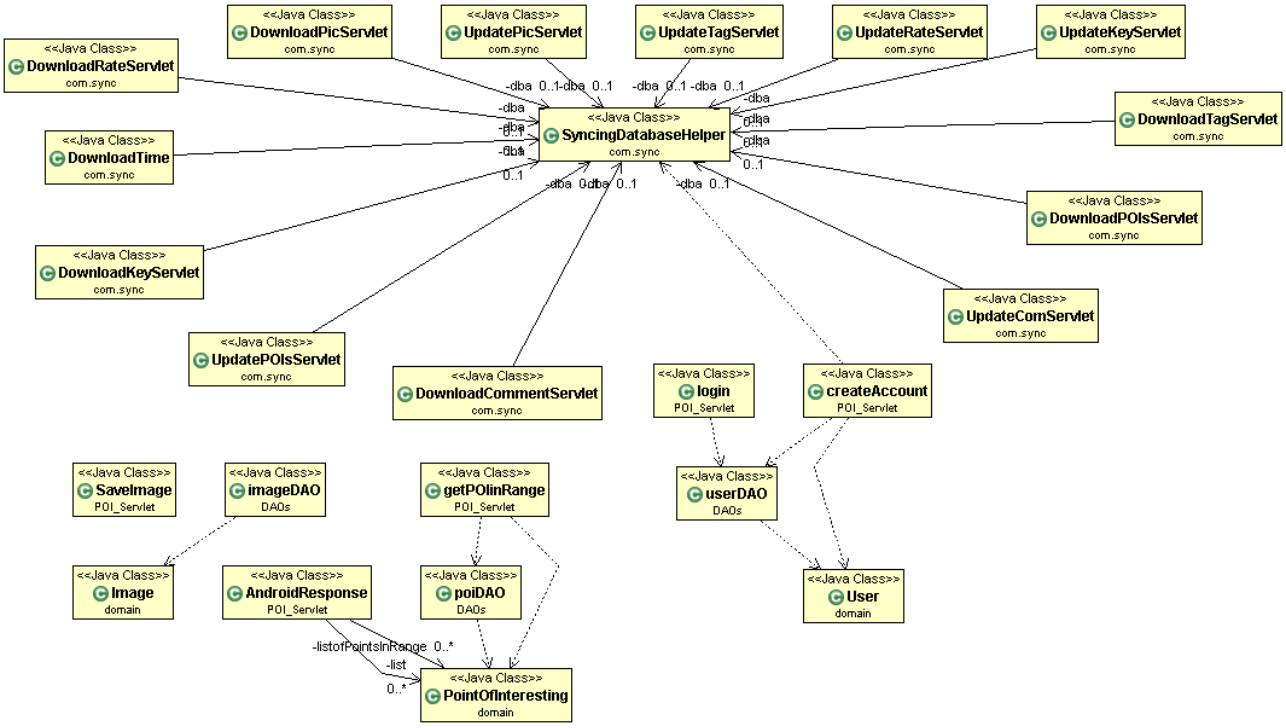
Reduced Client Class Diagram

For full diagram please see http://info401deki.otago.ac.nz/Project_Requirements/Documentation_Page



Reduced Server Class Diagram

For full diagram see http://info401deki.otago.ac.nz/Project_Requirements/Documentation_Page



Synchronisation Sequence Diagram