University of Cambridge

Computer Science Tripos, Part IB

Group Project Team Echo

Multi-touch Conference Progress report

Authors:
Mona Niknafs (mn407)
Yojan Patel (yp242)
Alexandru Tache (at628)
Philip Thomson (prt28)
Petar Veličković (pv273)

Client:
Catherine White, BT

12 February 2014

Contents

1 Introduction

1.1 Purpose

This document is intended to record the progress made by Team Echo on the Multi-touch Conference group project.

The system being developed will support the participation of multiple users in a particular notion of a conference, both local and remote to a multiple-touch and multiple-user supporting screen. In particular this notion is a group of conversations where each conversation can be overheard by any user of the system.

The document begins with a summary of what has been gained from our first client meeting on 29 January 2014. In the implementation and testing section which follows, the creation process for each module is explained in detail, along with the associated testing strategies that have been applied.

1.2 First client meeting notes

On 29 January 2014, the group met with Sue Hessey from BT to discuss the plans for the Multi-touch conference system as required by the original client brief. Numerous useful conclusions and system extensions have been discussed; they are given here in the form of bullet points.

- First of all, it was found that when presenting the system, the distinction between the touch-client and screen-client is not as clear as was originally imagined.
- A few limitations of the system were identified, distinct from those already recognised, along with prospective solutions.
- With regards to the data analyst module of the server, it was pointed out that interests and skills of users of the system will not be completely defined by job title. To define users as fully as possible, the system will need to also collect information the user personally provides, previous searches and previous messages. Additionally, due to the less personal nature of a computerised system, users may be members in a conference but might not participate in the traditional sense, through contributing to one or more conversations. Further motivation to participate could be introduced for users by integrating a leaderboard of highest contributors in each conference.
- Once the system functionality was analysed, it was found there existed **three** main potential system extensions that should be considered during system design and implementation. Originally, the touch system was intended for use on a tabletop or bracketed to the wall. It was found that these two environments lead to different ranges of possible user interactions. It would be logical to distinguish these into two uses:
  - The first application would stay true to the original brief, within a business meeting environment where local users have access to conversation windows and global statistics, around a tabletop integrated screen.
  - The wall bracketed screen environment was found to be more suited to an informal conference environment which, upon consideration, was found to be a very desirable environment. This informal setup may lead to more overall participation in a conference, for example if the screen was up in a "coffee/staff room", users are likely to read conversations on their own accord and perhaps participate in said conversations.
  - A final application of the system was mentioned as 'hot-houses', which was said to be very compatible with current development practices in BT. A 'hot-house', with regards to the systems definition of a conference, would be a conference instantiated to solve a particular problem through bringing together people of the appropriate disciplines.
  - Further to these extensions, the environment of any of these uses could be one of many. As opposed to the original business collaboration domain, the system could be used in education, call centres, retail and so on.

- Finally, a final system testing strategy was given. It was recommended a group of users, external to the project group, should test the system functionality as it would be used in its intended environment.

1.3 Deliverables reached

The deliverables set to be completed at this point in the development process were defined by the first prototype, $'\delta'$, in the *Functional specification* document: a stable system should have been built, that contains one server and one client, where useful output is displayed on the touchscreen client.

The status of the system as defined by $'\delta'$ has been reached, for both a touchscreen and Android client.

Furthermore, the requirements of the second prototype, $'\lambda'$, have also been reached: a smartphone application has been developed to a full functionality, and the model for server analytics is complete for basic analysis.

The next steps for the overall system are clearly defined at this stage by the third and final prototype, $'\xi'$. The system components should be integrated fully, to give a fully implemented touchscreen interface. Moreover, the touchscreen client, Android application and data analysis modules should be built upon to provide a useful and intuitive data presentation to users of both clients.

2 Module implementation and testing notes

2.1 Environments and libraries used

This section briefly outlines all the technologies used throughout the implementation stage of the project so far, in the form of bullet points.

- Software developed entirely using the **Java** programming language.
- **IDEs** used:
  - *IntelliJ Idea 13.0.2*
  - *Eclipse 4.3.1*
  - *NetBeans 7.4*
- **Platforms** supported by the software components:
  - *Android 4.0+*
  - *Mac OS X Mavericks 10.9.1*
  - *Ubuntu Linux 12.04 LTS*
  - *Windows 8.1*
- **Libraries** used:
  - *Android SDK*
  - *JavaFX*
  - *zxing*
  - *BeanUtils*
  - *Jersey*
  - *Grizzly*
  - *Jackson JSON Processor*
  - *Hunspell-BridJ*
  - *Porter Stemmer*
  - *Hibernate*
  - *H2 Database Engine*
  - *PostgreSQL*
  - *JUnit*
  - *Simple Logging Façade for Java (SLF4J)*
- **Build automation tools** used:
  - *Maven*
  - *Gradle*
- Additional **version control/testing/hosting** resources used:
  - *GitHub*
  - *Travis CI*
  - *Heroku*

2.2 Server

2.2.1 Core server API

Introduction

The core server module must fulfil the following roles:

- **Aggregate** data from different connected clients and **store** it in a **database** so that it can be **accessed** later (f.ex. for displaying a conversation history) or **recovered** in case of a server crash;
- Provide an **API** for the clients to access the data;
- Provide **notifications** about particular changes in the data (f.ex. when a new message is added to a conversation).

In the next section the implementation details for each of the roles is described.

Implementation notes

∼ **Database technology** In order to be able to work with different databases, we chose to use the Hibernate framework. It allows mapping database tables to annotated Java classes. The framework offers a very intuitive API to generate database queries without having to write SQL that also protects the database against the common forms of security attacks related to that (f.ex. SQL injection). It also automatically handles serialisation of data to and from the database.

The Hibernate framework can be easily be configured to use almost any kind of SQL database as a back-end (H2, HSQL, PostgreSQL, MySQL, Oracle, etc). As such, our server doesn't depend on any particular database technology. We currently are use a single file database (Apache H2) when running the server in development mode and a Postgres database when running the server on Heroku.

∼ **API** When we designed the server's outline, we wanted to make our API as simple to use as possible, base it on an already well established protocol and, most importantly, we wanted it to be language-independent in order for it to be easily accessible from multiple platforms (Windows, Android, iOS, Chrome, etc). Thus we have decided to provide our API as an **HTTP REST API** that provides data encoded in **JSON** which is a text-based, language independent, encoding.

In accordance to the REST conventions we define the following HTTP requests on the server for each data entity, as exemplified for the Conversation resource:

HTTP method	Server path	Description
GET	`/conversations/{id}`	Returns the conversation with the specified ID, encoded as JSON.
POST	`/conversations/`	Receives conversation attributes encoded as JSON, creates a new conversation using them and then returns the conversation as JSON.
PUT	`/conversations/`	Receives conversation attributes encoded as JSON (its ID must be present). It then updates the fields of the conversation with the specified ID.
DELETE	`/conversations/{id}`	Deletes the conversation with the specified ID.

We also define additional requests for accessing data particular to each entity. For example, for the Conversation resource we also have the following:

HTTP method	Server path	Description
GET	`/conversations/{id}/messages`	Returns the messages in the conversation with the specified ID.
GET	`/conversations/{id}/tags`	Returns the tags associated with the conversation with the specified ID.

To achieve this API implementation we used the Jersey framework, which is able to abstract away from the HTTP protocol and is also capable of distributing incoming requests across multiple workers automatically. In order to serialise and deserialise data to and from JSON we use the Jackson framework.

~ Server side notifications Another feature our server has to fulfil is the ability to efficiently notify connected clients about specific changes to the data set (f.ex. we must notify the clients when a new message is added to a conversation). This is slightly tricky to implement efficiently since the naïve approach of querying the server at predefined time intervals will cause too much unnecessary load on the server. Instead, we use a technique called HTTP Streaming, where HTTP connections that are querying the server for notifications are put to sleep until we actually have notifications for them. Fortunately, Jersey already provides an API for using HTTP Streaming.

Testing the server

The main technologies used for testing are:

- JUnit
- Hibernate Fixtures
- Jersey Testing Framework

The testing pattern for the server is as follows:

1. We organise and run our tests with JUnit, a general purpose Java unit testing framework.
2. For each test, we set up an in-memory database populated with pre-defined data, using Hibernate's support for fixtures.
3. We use the Jersey testing framework to simulate HTTP requests to the server.
4. Finally, we inspect the output of the server and the consistency of the database after the requests.

2.2.2 Data analyst

Introduction

As outlined in the functional specification, one of the key distinctive features of this project is emulation of a physical conference environment in a virtual setting (f.ex. on an Android smartphone). The main thing that separates a conference from an ordinary set of conversations is the "connection" that the conversations within the set have. A conference attendee, upon leaving a conversation, can easily listen in on other conversations to find something that interests him; also, it's not necessary for him/her to detach from the conversation to overhear something he's more interested in. We decided to simulate this by processing the data stored within a conference and using that to provide useful feedback to all the users of our service in real-time. The back-end behind all of this is in the **data analysis** module, which is documented in this section, while the individual clients are responsible for handling the interpretation of the received raw data.

Implementation notes

All the forms of output that might be required from the data analysis module have been listed in an interface that can be implemented in many ways, depending on the metrics the conference administrator would like to use to determine how to extract and rank the results; one sensible metric implementation has been provided for the purposes of this project. A single 64-bit integer identifier representing the parent `Conference` is the only parameter needed to construct a Data Analyst; this ID is being used when querying the database for snapshots from the conference. The methods primarily return lists of `Conversation` or `User` objects that correspond to the most relevant hits (the clients can specify the upper bound on the number of solutions they want returned). The methods include:

- **searching conversations** by **name**, **tags** and/or **keywords** within the messages;
- **most active conversations** in terms of **user count**;
- **most recently** (over a configurable interval) **active conversations** in terms of **message activity**;
- **most active users** in terms of **message activity**;
- **recommended conversations** for a specific user;
- etc...

The bulk of the aforementioned methods consist of simple queries to the server's database to extract the relevant data, and then using a priority queue with an appropriate comparator to filter the most appropriate results. The keyword searching and recommendation methods, however, required a more sophisticated algorithm and usage of a few external libraries in order to correctly extract keywords from messages and process those keywords against our query word. The remainder of this section will be a description of the **two algorithms** used to achieve this, **and their dependencies**.

$\sim$ **Algorithm 1 - Keyword extraction** The main problem tackled by this algorithm is: given a string representing a single message, produce a (possibly empty) list of base keywords within that message. For example, given the string

"The current financial crisis is the worst the world has seen since the Great Depression of the 1930s."

the algorithm should produce a list of strings similar to

`["current", "finance", "crisis", "worst", "world", "see", "great", "depression"]`.

As this is an open problem in NLP, there is no "correct" answer and our algorithm should not be 100% accurate; we should aim for something that works in linear time (as we might have a potentially big number of messages to process) and has a reasonable level of accuracy – over a large set of messages the keyword frequencies should converge to something that reasonably describes the conversation they belong to.

We start off by preparing the string for analysis and generating the initial list of token words; this implies converting the string to lower case, removing any punctuation and non-letter characters (including digits), and finally splitting the string with whitespace as a delimiter.

Once we have our tokens, we need to make sure that they are properly spelled before going any further; the Hunspell library was our choice for the job, as its API is very simple to use and it's used as the spell-checking tool in applications such as *LibreOffice, OpenOffice.org, Mozilla Firefox & Thunderbird, Google Chrome* etc. More precisely, as the original library is written in C, we used the Hunspell-BridJ project to interface to it in Java via BridJ.

After obtaining correctly spelled tokens, it is a good idea to remove stop-words; words that are commonly used in communication and don't really contribute to describing a conversation (f. ex. "a", "an", "whatever", "some"...). We have a plain text file containing the list of stop-words for this purpose.

In order to further enhance our results, we wouldn't want f.ex. "interface", "interfaces", "interfacing" etc. to be counted as different words, as they semantically represent the same base word ("interface"). To take care of this problem we have used a stemmer, which is the less accurate but also less expensive method of bringing a word to its base form; it essentially cuts the commonly used word appendices in the English language and does some basic character conversions; we have used a well-known implementation by Martin Porter (the "Porter Stemmer"). Finally, to compensate for possible errors made by the stemmer, we do another spell-checking pass before returning the final list of base words.

The entire algorithm is summarised below:

Algorithm 1 Keyword extraction (by `pv273`)

```
1:  function ExtractKW(S : String) : List<String>          ▷ Extract keywords contained in S
2:      S ← ToLowerCase(S)                                  ▷ Convert S to lower case
3:      S ← RemoveNonLetters(S)                             ▷ Convert non-letters within S to whitespace
4:      List<String> ret ← Split(S, '␣')                    ▷ Split S with whitespace as a delimiter
5:      for all String w ∈ ret do
6:          w ← SpellCheck(w)                               ▷ Fix any misspelling of the token
7:          if w ∈ stopWords then Remove(ret, w)            ▷ Remove stop-words
8:          else
9:              w ← Stem(w)                                 ▷ Stem the word
10:             w ← SpellCheck(w)                           ▷ Fix any faults caused by the stemmer
11:         end if
12:     end for
13:     return ret
14: end function
```

$\sim$ **Algorithm 2 - Keyword processing in a query** The problem tackled by this algorithm can be stated as: given a (previously extracted) list of keyword strings K and a query string Q, determine a "score" gained by querying Q on K.

As we require our whole algorithm to scale well to a large amount of messages, we have decided to use the Knuth-Morris-Pratt (KMP) string matching algorithm, which has asymptotic time complexity $O(n+m)$, where n and m are the lengths of the string in which we are looking for matches and of the matching pattern, respectively. Another interesting feature that KMP has and we found useful is the ability to keep track of the **longest matched prefix length** throughout its execution; this given us a possible **scoring function** that eventually got implemented. That function is as follows:

$$f(K,Q) = \sum_{i=1}^{|K|} \frac{\mathrm{KMP}(K_i, Q)}{|K| * |Q|}$$

where K_i corresponds to the i-th keyword in K, $|K|$ represents the amount of keywords in the list, $|Q|$ represents the length of the query string, and $\mathrm{KMP}(K_i,\ Q)$ corresponds to the length of the longest matched prefix of Q in K_i as found by the KMP algorithm.

Testing notes

The two algorithms listed above have been unit tested locally using JUnit on messages and queries of varying length to verify that the output keywords represent the message properly, and that the scoring function is calculated properly.

Testing of the efficiency and quality of the remainder of the analysis module has been performed on the server that we had set up on Heroku, both by issuing HTTP queries directly via browsers, and through all of the already implemented clients (Android, touchscreen, computer). The test results were highly satisfactory with respect to both criteria.

Issues

Nothing of significance to report - the development of the data analysis module hasn't encountered any significant setbacks.

2.3 Client

Introduction

The **client** library is a Java library that handles communication to the server API described in subsection 2.2.1. Its role is to provide a highly friendly API that abstracts away from most of the communications to the server.

In order to guarantee consistency between the server and client APIs we have defined several interfaces that describe the data sent between the server and client as well as the HTTP requests that the server accepts. Details about the implementation can be found below.

Implementation notes

In the `uk.ac.cam.echo.data` package of the project, several interfaces define the entities stored in the database alongside with the methods used for information retrieval. For example, consider the `Message` interface:

```
1 public interface Message extends Base
2 {
3   public long getId();
4   public long getTimeStamp();
5   public User getSender();
6   public Conversation getConversation();
7   public String getContents();
8   public void setContents(String contents);
9 }
```

In the `uk.ac.cam.echo.data.resources` package, several annotated interfaces define the HTTP requests that the server has to support. For example, consider a part of the `ConversationResource` interface:

```
@Path("/conversations")
@Produces("application/json")
public interface ConversationResource extends RestResource<Conversation> {
    @GET
    @Path("/{id}")
    public Conversation get(@PathParam("id") long id);
    ...
}
```

The Jersey client framework is a very useful tool that allows us to automatically implement the needed HTTP requests based on the resource interfaces defined above. It is able to generate, at runtime, an instance of the interface where each method call automatically does an appropriate HTTP request to the server. It also handles the serialisation and deseralisation of data to / from JSON automatically using the Jackson framework.

Testing details

The testing pattern for the client library is very similar to the one used for testing the server:

1. We organise and run our tests with JUnit, a general purpose Java unit testing framework.
2. For each test, we set up a server in the current process.
3. The server uses an in-memory database populated with predefined data, using Hibernate's support for fixtures.
4. Finally, we use the client library to issue requests to the server and inspect the output.

2.3.1 Touchscreen client

Introduction

At this stage in the project, the **touchscreen client** is expected to have the ability to connect to the conference server, and the screen itself providing a basic user interface and with associated gesture recognition. These goals have been largely achieved.

Current system state

The connection to the server has been established through the ServerConnection class, which provides connections between classes and stores information about the conference the user is in.

The user interface has been implemented to give the components on the touch screen enough distinction to allow for basic user interaction. The user interface has been implemented using JavaFX libraries. This user interaction has so far been provided twofold: through gesture recognition on the multi-touch screen and connection to a server conversation thread from an android application user.

Gestures on the touch screen have been provided using action listeners based on the JavaFX libraries which allow the program to recognise the touch gestures generated by the windows touch drivers and the touch screen.

Android application users can connect to a conversation of interest displayed on the screen through the scanning of a QR code generated on-screen. The zxing barcode image processing library was used to provide this functionality in combination with the JavaFX image display library.

Problems encountered

Through the development process a few complications were encountered. Concerning the gesture recognition, it was not completely clear how the RotateEvent class and Node class provided by the library functioned and how it interacted with the other touch gestures.

In addition, there have been, and still exist some concurrency issues between the server and GUI; for example, when conversations are switched and previous messages are displayed at the same time as new incoming messages.

System testing

The components of the system have been so far tested to a reasonable level. Although it is an accepted limitation that the touch screen can support a limited number (10) of concurrent touches, the system behaviour was tested when under this pressure. An exception was thrown by the underlying libraries, which caused the system to lose the ability to recognise gestures. This gesture will be appropriately caught to result in desirable action.

The few gestures so far implemented in the system have been tested through natural user interaction with the screen. The drag function works as expected. The rotate function originally was unresponsive and jumpy, this was due to an issue with the drag gesture overriding the rotation gesture. This problem has since been resolved.

System development

There exist a few main steps that should be taken for the development of the touchscreen client.

Firstly, a working replacement policy should be implemented for the conversation windows displayed on the screen. This policy should be determined through careful consideration of system and the associated user interactions. Furthermore, the user interface should be developed to a more user friendly and well designed standard. This includes the addition of useful tool icons and user avatars.

Finally, through integration of the touch client and data modules, the statistics, both global to the conference and local to the conversation should be displayed and interpreted in a logical and understandable manner on the screen interface. The understandability of the statistical representations will be assessed through appropriate user testing.

2.3.2 Android client

Introduction

The **Android application** to be used as the main form of input by the attendees of the conference should at this stage have been in a prototype stage where Activities and the associated user interface components have been set up. These goals have been achieved and all problems regarding build tools and compatibility of Java libraries with the server have been handled.

Implementation notes

Currently, the app consists of several activities (individual user screens) that allow for user login, viewing the available conversations in the conference and a messaging activity where the attendee would send and receive messages. The login activity currently consists of simple validation and is largely used as a placeholder for when a proper user authentication system has been implemented.

Viewing the individual conversations is the main activity in the app where the user is likely to spend the most time other than the individual messaging activities. It consists of a list-view to see some basic information about each conversation such as the usernames and the amount of people in the conversation, associated tags and the name. Touching any of the single conversation elements opens up a dialog where the user would be able to see an extended overview of the conversation and have the option to join the conversation if desired. The motivation for this functionality was to allow the user to preview the conversation before committing a join. The conversation list activity will also interact with the large touchscreen interface which generates QR codes for the conversations. A button in the ActionBar causes an implicit intent to an external app to allow for QR code scanning and thus joining the conversation. The user can also open up another dialog to create a new conversation in the conference. Search functionality using the data-analysis component of the server has also been implemented to allow the user to search for conversations.

Finally, the messaging activity employs the familiar messaging system with chat bubbles, but maintains the main difference from existing systems. i.e. a group in one office speaking to a group in another office.

Testing

Testing for the system has taken place by running the app on multiple Android devices (phones of different sizes and a tablet). The user interface adapts well to different sizes and processing power variation of the different devices. QR code testing has also taken place using QR code generators. Testing the listview was also carried out to ensure edge case situations were handled accordingly (no conversations in conference and a large amount of conversations simulated with dummy data). Search functionality on the app was carried out by comparing achieved results with results from the unit testing of the search function from the data analysis component of the server. The creation of conversations and adding new messages to various conversations was also carried out and checked against the database state.

Problems encountered

Various libraries in the server and client library implementation as well as other dependencies are not provided by Android. Using the standard build tools provided by Android Developer Tools (Ant) to build the integrated app also proved difficult and therefore the application build configuration had to be converted to use Gradle. The list view for the conversations was also initially slow with the standard ArrayAdapter and therefore, overriding it with optimizations implemented (ViewHolder pattern, recycling views etc.) had to be carried out.

Next steps

The current system consists of several activities with their user interfaces implemented. While an effort has been made to ensure an intuitive interface, they are likely to change as additional functionality is incorporated in the system. An ever-lasting service has to be implemented which would provide notifications to the user on any activity taking place in the conference. As more components are implemented in the server, they will be integrated with the Android app.

2.3.3 Computer client

In order to be able to manually test the system as well as to provide a simple and reliable way to participate in a conversation from any computer we also have implemented a small text-based **desktop client**. At the moment the text based client allows you to create or join conversations and then write and receive messages on that conversation.

No tests have been written for this client, it was tested only manually.