Final Project - Milestone 2

Christopher Roman — **cr469**
Mushaffar Khan — **mk2248**
Mubeen Sheeraz — **ms2689**

Due December 07, 2017

Camlator

1 System Description

1.1 Summary

We have created a messenge application similar to WhatsApp/Facebook Messenger. However, *Camlator* is different in the sense that it utilizes Google's Translate API to translate messages between people who may converse in different languages.

1.2 Core Vision

With more than 6000+ languages in the world, it is sometimes difficult to get ones thoughts across to someone that does not speak the same language. That is why our main goal for the final project in 3110 was to build a chat box that is similar in vein to WhatsApp/Facebook Messenger such that the users can communicate with each other inspite of the language barrier. The application asks each participant for his/her language preference and they can send messages to any other participant in their preferred language without any sort of barrier. When the other person replies to the current user, that person may reply in any language of their choice and it will be translated for the current user to their laguage so that they can understand and reply. A good example is if person A's language preference is English and person B's language preference is Spanish, they can converse with each other in their preferred language and *Camlator* will translate A's (English) messages to B's language preference (Spanish) and vice-versa for messages sent by B to A.

1.3 Key Features

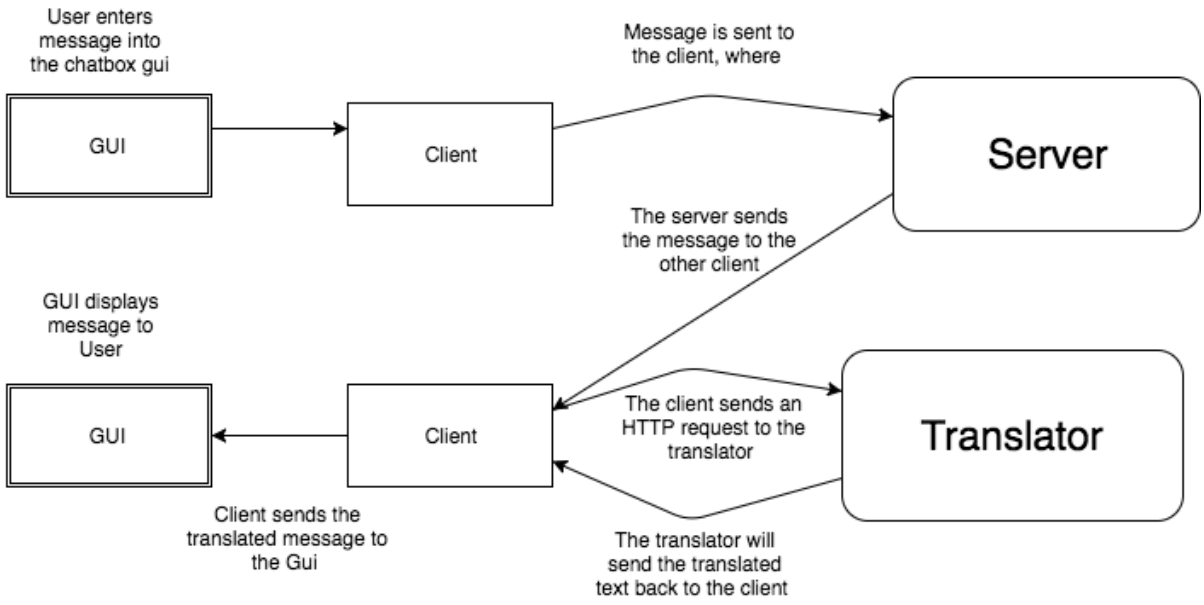
Some of the key features of *Camlator* include:

- It supports general messaging between two users
- Message text is translated in such a way that each user sees it printed onto their screen in a language of their choice, regardless of what language the other users prefer.
- Will support all UTF-8 characters rather than being limited to ASCII characters

- A GUI built completely using OCAML that has all the basic features of a message application such as a close button, box that displays the messages, with the addition of a drop down menu to select the language of user's preference.
- The application also supports the main emoji's but since our implementation of the project is in OCAML, we had to utilize aschii emojis, as the GUI library (lablgtk) for OCAML is outdated and cannot display the current emojis in the text field.

1.4 Top Level

We initially planned on using Bootstrap for as our front end of *Camlator* but we decided to make GUI in OCAML, utilizing the lablgtk library, as that was a lot more sensible with respect to rest of the project. We used a HTTP server to host the clients, which are the Inputs and Outputs in *a*, where they are connected via sockets. As a client sends a message to the server, the server will receives the information (as a HTTP request) and then passes it to the other client via sockets. Previously we had decided that we wanted to handle the language translation on the server side, however, later we found it to be a more intuitive and sensible decision to handle the translation on the client side. Therefore, once the server sends the message to the client, it is the client's job to send a HTTP request to the Google Translate API, where the API returns a JSON with the translated text. This way, the user can specify the language they want to receive the messages in and the since the translation is being done locally, translating text becomes easier. This way the server does not need to handle, which language it needs to translate from and to, who sent which message in a certain language, etc. Figure *a* is a rough top level view of how we implemented *Camlator*.



(a) Top Level of Camlator

2 System Design

The following discusses what modules are found in our system:

1. **Chatbox** — This module is the GUI for our application, using lablgtk, and is the main entry-point for a "client". On startup, a thread is ran for the GUI which displays the chat history and most importantly allows the user to change their preferred language (with default as English), so any messages received will be translated into that language. We provide some functionality to make the chat more exciting to use, such as name changing and preloaded text emojis. In parallel to this thread, we connect to the server through `Lwt`'s wrapper for sockets, and communicate through them.

2. **Chatclient** — This module is used for the underlying connection made between the "client" and the "server". Here is where the actual sending and receiving of messages happens. We use `Lwt` sockets because when doing any I/O operations, they are non-blocking. On the receipt of a message, it will be translated based on the language preference set in the GUI. The translated message is then shown on the GUI.

3. **Httpclient** — This module is meant to translate messages (based on some preferred language) using the Google Translate API — this is done simply with HTTP Requests. One caveat is that we didn't have the necessary tokens to use the Google Translate API. We saw there was a package in `Node.js` that could do this for us, so we have a small script written in javascript that will do this for us. The Httpclient uses this to get the token, and uses this token for the HTTP Request.

4. **Stringliterals** — This module holds our pre-loaded data for the GUI, such as the supported languages and the text emojis.

5. **Chatserver** — A Chatserver is a singleton meant help clients communicate with each other. It's job is to receive all client's messages and broadcast to all other clients. A Chatserver keeps a log of every client's message history. A Chatservers must also deal with UTF-8 characters due to the plethora of supported languages.

3 Data

3.1 Storage

Most of our data pertaining to this project will be the chat history amongst people, which will be stored simply as a mutable string in the Chatserver. The string will contain all the chat messages and the person that sent them as UTF-8 characters. Note that this is just stored in-memory, so if the Chatserver goes down, the message log is lost. This decision is for simplicity sake of not having to write to some stable storage such as a file or a database. The client's information, i.e. their name and preferred language, is stored locally by them in-memory. Clients also end up keeping a copy of the chat history.

3.2 Communication

Communication is done between the clients and servers via `Lwt` sockets. Clients send messages to the central server. The server broadcasts these messages to all other clients. It is important to note that two different `Lwt` threads are used — one for sending and one for receiving. This way, we can still send messages without waiting to receive a message, and vice-versa.

3.3 Data Structures

The server makes use of `Map` to keep track of outgoing connections so when it needs to broadcast something, it can do so easily. We use a `Map` to be able to efficiently insert and remove clients as connections open and close — this way we can handle a high volume of clients.

4 External Dependencies

Some of the modules we are using:

- **YoJson:** To parse HTTP Responses from the Google Translate API
- **Cohttp:** To form HTTP Requests to interact with the Google Translate API
- **Lwt:** To run the GUI in parallel with communication via sockets (which is also run in parallel, namely between sends and receives)
- **Lablgtk:** To create the GUI so users of our application have an intuitive front-end

Some external dependencies outside of OCaml itself:

- **Node.js** This is extremely important for generating tokens.

5 Testing Plan

We tested each function heavily as we were progressing along in the implementation. Also, as we added more functionality to *Camlator* we tested to see if our old functions that we implemented were not affected by the changes. Moreover, once we completed *Camlator* we will tested to see how the overall functionality of *Camlator*, and how accurate it was in having a conversation with various languages. To make sure that we committed to our test plan as a team, not only did we test the functions we implement ourselves, we also rigorously tested each other's implementation to catch any nuances that one may have missed. The bulk of our testing came from seeing how other users interacted with the *Camlator* messenger interface. To do so, we had plenty of our non-3110 friends test it out with each other and took their feedback to update *Camlator* accordingly. This way we were able to catch any bugs that came up as students were messaging each other via our computers, particularly any concurrency issues that came up and user interface issues with the GUI.

6 Division of Labor

Mush: Built the front end of the messaging app in OCaml using the `Lablgtk` library. Made the chat room where you can enter the Users name and language preference that they would like to message in. Also, implemented the features so that the front end is responsive to inputs and also implemented emojis. Time spent: about 30 hours.

Mubeen: Went back and forth between Mush and Chris where he helped Chris implement the actual translate functionality in OCaml that sent the Google Translate API a HTTP request with the phrase and language it wants to translate it to. Mubeen also helped Mush with the front end by helping implementing the User's language preference and emoji selection in the chatbox. Time spent: about 30 hours. message.

Chris: Implemented the backend client-server architecture, learning `Lwt` along the way. Implemented broadcasting system to send messages to every client. Helped work on itegrating the GUI with the communication backend to have a fully functional chat application. Time spent: about 30 hours.