Translateet

NET COMPUTING 2016
A high quality real time translator system

Mijland, Stijn Thomas (s2589443)
`s.t.mijland@student.rug.nl`

Jimenez Hernandez, Jose Antonio (s2528223)
`j.a.jimenez.hernandez@student.rug.nl`

March 22, 2016

1 Introduction

Before the quality of automatic translators matches that of human translators, a lot remains to be done. Besides, we are still suffering the consequences of the economic crash in 2008: higher unemployment and tougher competition. **Translateet** comes to offer a (partial) solution: an application that allows users to request translations on small text fragments, translated by real translators. Users can rate these translations, allowing serious translators to build a CV.

2 System Analysis

The idea of this application is to have translators provide translations for real-life people. This system is to be presented as a service, rather than an API, which means users will expect a website.

We realise that many phrases are common requests to translation services. Therefore many previous translations can be stored in a database. Only requests that are not in that database need to reach the translators.

We realised that translators are people too, so they are not always available. As a result we have a potential timing mismatch between users and translators we need to address.

3 System Architecture

To present the system as a service, aswell as to improve website compatibility, the obvious choice for a first entry point is a REST service. This service would be responsible for receiving and answering all interaction with the user.

As a database we use MongoDB. MongoDB is highly compatible with web technologies, provides high performance and good integration with our system.

Addressing the timing mismatch between users and translators is easily fixed using RabbitMQ. RabbitMQ provides a convenient interface that allows us to solve multiple intricate issues.

Apart from the timing issues, by using the proper naming, routing a message based on relevant languages is a trivial affair. If a translator can translate i.e. English to Spanish, by sending a request for Spanish to an English exchange, that message is easily enqueued for the translator.

Although not a common scenario, for this project we were required to use multiple technologies. We explored two options: The use of RMI to simulate an AI translator at a distance, and the addition of a support chat using TCP/IP.

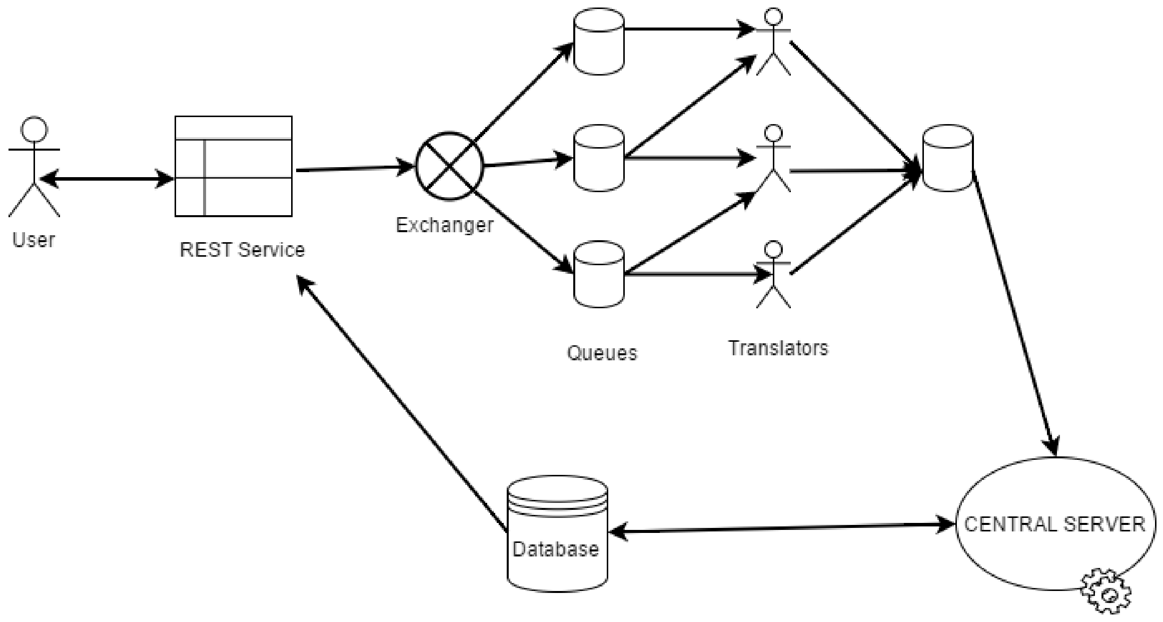

Figure 1: *Physical view.* Architectural model that shows the subsystems.

3.1 REST

The REST interface is responsible for the front-end. A request for a translation is first sent to the database, and if it doesn't have anything, forwarded through the rabbitmq architecture to real translators.

3.2 RabbitMQ

The queue-based subsystem treats a translation as a transition from one Language to another. A Language can be asked to translate a message to another language. That other language becomes the key that allows Translators to only receive languages that they know. Each Translator deposits their translation in a DBQueue, a queue dedicated to updating the database. There are ofcourse multiple Languages and multiple Translators.

3.3 RMI

We use RMI to simulate an AI on some distant server. It performs a heavy translation calculation, and returns a robot translation. It is used as an alternative for the rabbitMQ network.

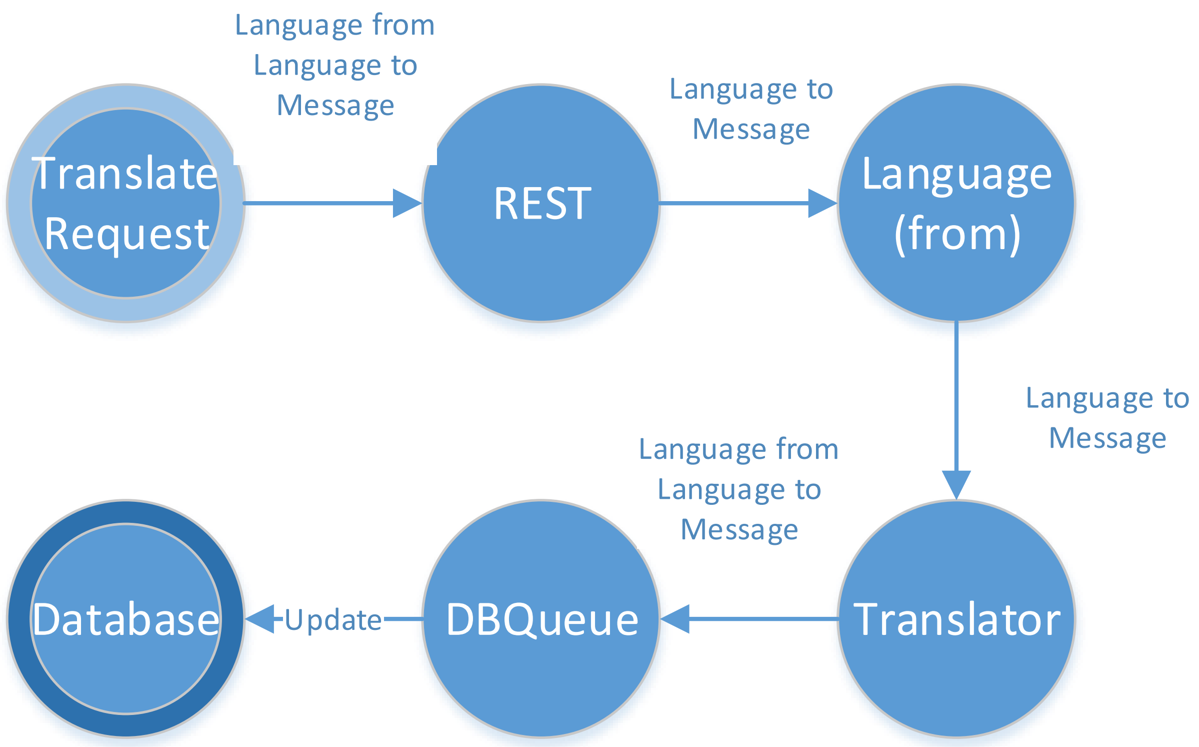

Figure 2: *Data Flow in RabbitMQ Section*

3.4 TCP Chat

The TCP chat simply sends messages back and forth between two people, using the basic structure presented by the Java tutorials.

4 Logical view

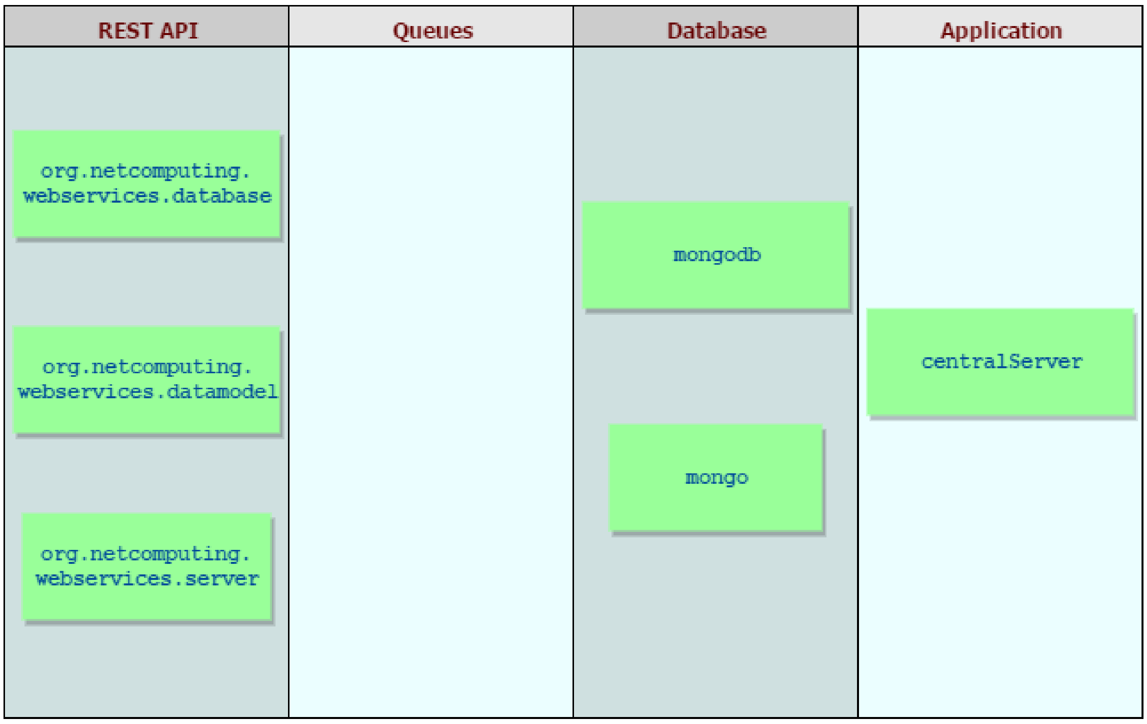

Figure 3: *Logic view.* The project split up in logical independent parts.

The `REST API` is divided in three folders (tests, resources and main). The one in the diagram is the `main` one, which contains three packages. The `datamodel` package stores both the classes that represent data (`Texts`,`Translations`, `Users`) and the data access layer to link this data with the database. The `database` package implements the hard-coded communication with the database. The `server` package implements the initialization of the `REST API` as well as the `ConfigLoader` class for initialisation the mongo DB. The database could be in an independent machine, here for the scope of the university project, we run both the REST API and the database on the same machine.

The `resources` folder contains a unique file, `mongodb.properties` and stores the database configuration (name, port, address) that is easily changed and loaded on initialization of the server. The `tests` folder contains mostly database tests.

The (Mongo) `Database` is composed of the executables `mongodb` and `mongo`. The first runs the handling of the database connections on port 27017. The second is a command line interface that lets the administrator do changes on real time on the databases (e.g. `db.users.find()` will show all users stored in the database). Although it was not coded by us, we considered it important enough to be commented in the present document.

The `Application` is another way of referring to the `CentralServer` class, which receives translators from a `Queue` and writes them in the MongoDB.

There are also packages for RabbitMQ (`org.netcomputing.webservices.queues.q4user`) and for TCP chat (`sockets`)

5 Physical view

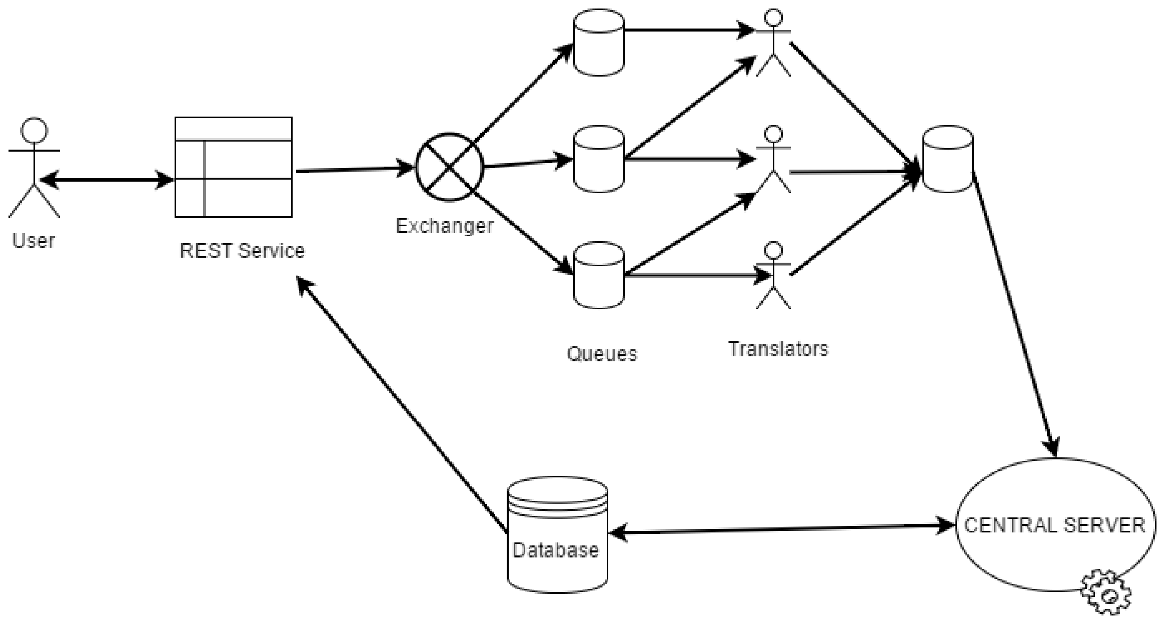

Figure 4: *Physical view.* Architectural model that shows the subsystems.

An architectural model shows an overall system split up in smaller subsystems. Those subsystems are meant to gather subprocesses that are related each other (eg. use the same data representation, programming language, are placed in the same machine, or aim to solve a particular subproblem).

We find a `User` that requires a translation and interacts with the `REST API`. The `REST API` aims to solve the general purpose `GET` and `POST` services (e.g. `GET` an user with a specific UID). The `REST API` should aim to deal with many requests, thus being stored in an individual server would be desireable.

Another subsystem would be the `translators'` one. This is a mix of the technology `RabbitMQ` and special `Users` that handle the requests. The latter are beyond the control of the system, but the `RabbitMQ` is implemented using a central filter for languages, that splits the requests into different `Queues`, and then a central `Queue` (let it be called "Answers Queue") handles the responses from the translators.

This filter and the queues are initially stored in the same machine; however, having them split in several queues facilitates scalating the system too. Adding a new language could be a matter of adding a new queue, and making the system able to handle many more requests would be a matter of moving a queue to other machine, a process that is supposed to be straight-forward.

All this distribution of subsystems into different machines makes it easier to **scalate** the whole system.

6 General remarks

The experience working with **networking technologies** was enjoyable. We learned how to structure a program in several layers, as well as physical machines and subsystems. We got dirty hands on teamworking, getting a deeper use of **Git** and repository systems. Such a repository can be found in the following link: https://bitbucket.org/translateet/translateet/overview

When working on the project, we considered some tasks to be extra: the web design and a proper `user subsystem`. The web design was still acceptable, using some previous work from other personal projects. We did not implement a real user subsystem.

As we got deeper into the project, we realized that it was larger than expected, and hardly manageable. Many features that conceptually were close, when it came to be implemented, did not seem as straight away as expected; for example, the score system for messages and users by means of REST API and the MongoDB.

We had to fake the translators activity as well, as creating dialogues with users would have been (still) more extra work. At the same time, we were still implementing new features, because we were still missing some of the necessary technologies. This ultimately resulted in this rushed report, and many features that are not yet integrated.

Thus, the previous document represents the (whole) conceptual idea, but is not completely implemented.