CS807 - Final Project

André E. dos Santos
Jhonatan S. Oliveira

April, 2016

University of Regina

Department of Computer Science

Resource Constrained Computing

CS807 - Final Project

André E. dos Santos
Jhonatan S. Oliveira

1. Reviewer Dr. David Gerhard
Department of Computer Science
University of Regina

2. Reviewer Mr. Trevor Tomesh
Department of Computer Science
University of Regina

April, 2016

André E. dos Santos
Jhonatan S. Oliveira
CS807 - Final Project
Resource Constrained Computing, April, 2016
Reviewers: Dr. David Gerhard and Mr. Trevor Tomesh

University of Regina
Department of Computer Science
3737 Wascana Pkwy
Regina, S4S 0A2

Abstract

An interactive kiosk is a computer terminal that provides access to information and applications for communication. It contains specialized hardware and software designed within a public exhibit. Thus, an interactive kiosk must address resourceful solutions to minimize its possible constrains. In this project report we propose an interactive kiosk with speech recognition for academic environments. We utilized *Wit.ai*, a *cloud service* that turns speech or text into actionable data. Wit works on the project as a accessibility tool, an interactive feature. The server, data modeling to the client and user interface was built with *Meteor.js*, a full-stack javascript framework. The hardware of choice was Raspberry Pi 2, a low cost single-board computer with salient features to support efficiency and reliability for the project interactive kiosk.

In this project report we discuss the process and constrains to build an interactive kiosk with speech recognition. We describe the implementation and functionality of the project to use clouding platforms to overcome hardware platform limitations. We utilized a system implemented in Meteor.js containing a server and a client. The server, a more powerful computer, is used as a central processing unit and only forward responses for the client's processing requests. Finally, we discuss the challenges of localization and map apps that impose barriers efforts for simple implementations.

Contents

1 Introduction

Accessibility is a key assistive accommodation to enable people overcome physical, technological, or informational barriers [1]. As companies seek for cost reductions, revenue opportunities, but yet with good quality of customer service, interactive kiosks has been a leading medium for communication trend to enable their customers with self-service options [2]. A *interactive kiosk* is a computation system terminal with spcialized hardware and software designed for self-service options, such as information, communication, commerce, entertainment, and education [27]. As a computation system, a interactive kiosk systems has specific constrains associated with its components. For instance, power constrains applies since the terminal usually is implemented in low power devices. In this report we present the proposal and implementation of a low cost interactive kiosk with voice recognition feature.

In our project, we sought for tools to enhance the experience and quality on a interactive kiosk systems. *Wit.ai* [38] is a cloud service that allows a simple implementation of natural language processing for developers. With only a few lines of its code, Wit.ai let developers build a speech recognition and voice control console.

As for an efficient system to manage the data and improve the information retrieval, the implementation was executed with the open-source web platform called Meteor.js. With *Meteor.js* [18], or Meteor for short, developers can create applications using the Web 2.0 paradigms. The platform provide a reactive approach by focusing on the data flow. This is done by creating and managing events in the application. Also, data management is done with a non-SQL database called *MongoDB* [19]. Meteor simplify the application development by providing an unique programming language, javascript, throughout the whole stack process. Moreover, the platform makes available a set of common tools for business logic and data management. Finally, Meteor deploys the application in desktop and mobile without needing to change the source code.

The cost of the project is manly dictated by the hardware component that holds the system. Some platforms often utilized for low cost system are BeagleBone Black, Arduino, Phidgets, Udoo, and Electric Imp [15, 25]. In our project, the hardware chosen *Raspberry Pi* [25], a low cost single-board computer that runs on the various distribution of Linux operating system and can be programmed as needed. One of

the Raspberry Pi advantages is it capability for a remote communication, making it a suitable choice for applications in the *Internet of Things* (IoT) concept.

With the implementation some issues were faced. For instance, the initial plan for our final project included a service inside the app for indoor navigation. This feature is commonly seen in big centres such malls, hospitals or airports. However, indoor navigation itself is a complex and well known task problem in computer science and engineering. Although some systems are available, such as *OpenStreetMap* and *Anyplace* [6], they are not at low level for simple systems and not suitable for small projects. In this way, we wanted to show how resource constrain devices can still be used to provide such service by using cloud computing. After facing those constrains (better described in Chapter 4), we then turned our attentions for the software design and search engine of the interactive kiosk. We implemented basic methods for searching graphs that perform efficiently search on a tree structure: *depth first search* and the *breadth first search* [12].

The project presents many salient features. First, it can be voice activated and has state-of-the art speech recognition throughout the usage of the Wit.ai. Second, it performs an solid search on the tree structure implemented on the web platform Meteor.js. Features as uncertainty decision and incomplete information are well handled by the system. Third, the hardware of implementation was on a Raspberry Pi, which is well suited for the project since it is (i) low cost system, (ii) customizable and (iii) programmable. The result is an interactive kiosk that "understands" voice requests of informations, answering them with a variety of possibilities responses, all editable by the administrators.

The reminder is as follows. Chapter 2 shows a quick overview on the history and background informations of interactive kiosk systems, reactive programming, and graph data structures. The project implementation along with related works, platforms used, and results are given in Chapter 3. Chapter 4 shows the discussion on constrains of developing an the app for indoor navigation. Conclusions are given in Chapter 5. Additional and complementary coding are provided in Appendix Chapter 6.

2 Background

In this chapter we give an overview on interactive kiosk systems, reactive programming, and graph data structures.

2.1 Interactive Kiosk

As defined by [27], "an interactive kiosk is a computer terminal featuring specialized hardware and software designed within a public exhibit that provides access to information and applications for communication, commerce, entertainment, and education." By integrating emerging technologies, kiosks can perform a wide range of functions, evolving into self-service kiosks. The main types of self-service kiosks of telephony, financial services, photo, internet access, ticketing and information kiosk. According to [14], "historically electronic kiosks are standalone enclosures which accept user input, integrate many devices, include a software GUI application and remote monitoring and are deployed widely across all industry verticals."

The reduction of costs, increasing revenue opportunities, and improvement customer service are the main goals of companies in a range of industries [26]. Interactive kiosks has been used as an additional channel to enable their customers with self-service options, matching in this way the primarily goals those companies. According to [2], by 2016 the number of interactive kiosks in operation will rise from approximately 1.6 million deployed in 2011 to nearly 3 million deployed globally. While the interactive kiosk market is expected to grow strongly, there remain challenges to address. Reliability is an important consideration. Many specialised kiosk software applications have been developed for the industry [14].

2.2 Reactive Programming

In 2001, after the the bursting of the dot-com com bubble, the Web 2.0 phenomenon appeared and a survival way to the recent collapse [23]. New applications aimed to involve the user on the content creation process. One way of engaging the user is to offer a familiar environment, a similar to the one used in a desktop computer. Web platforms and frameworks made use of *reactive programming* [5] to achieve a desktop-like experience on the web. A consequence of turning the content creation

responsibility to the user is the massive amount of data created by them, which originated another phenomenon called *big data* [30]. In order to manage all these data, new paradigms emerged. *Non-relational* or *No-SQL* [32] is a database paradigm that manage the data storage and retrieval without using a relational table. Novel web platforms and frameworks had to incorporate these new aspects in order to model and maintain the complexity requirement of the new applications.

2.3 Graph Data Structures

Graphs are a fundamental data structure in the world of programming. Knowing the correct data structures to use with graph problems is critical [33]. Graphs can represent many different types of systems. A graph consists of two components: nodes and edges. A node (or vertex) is a discrete position in the graph. An edge (or connection) is a link between two vertices that can be either directed or undirected and may have a cost associated with it. An undirected edge means that there is no restriction on the direction you can travel along the edge. Formally, a *graph* $G = V, E$ is defined as a set of vertices, V, and a collection of edges (which is not necessarily a set), E [4]. An edge can then be defined as (u, v) where u and v are elements of V.

2.3.1 Trees

A tree is an undirected graph in which any two vertices are connected by exactly one path. Thus, a acyclic connected graph is also a tree.

2.3.2 Basic methods for searching graphs

There are two methods for searching graphs that are prevalent: *Depth First Search* and the *Breadth First Search*. In this section, all steps are drawn from [12]. For more details, please refer back to the original source.

Breadth-first search

Breadth-first search (BFS) is a strategy for running through the vertices of a graph. It was presented by Moore [20] in 1959 within the context of traversing mazes. Lee [17] independently discovered the same algorithm in 1961 in his work on routing wires on circuit boards.

The basic BFS algorithm can be described as follows. Starting from a given vertex v of a graph G, we first explore the neighbourhood of v by visiting all vertices that are adjacent to v. We then apply the same strategy to each of the neighbours of v. The strategy of exploring the neighbourhood of a vertex is applied to all vertices of G. The result is a tree rooted at v and this tree is a subgraph of G. Algorithm 1 presents a general template for the BFS strategy. The tree resulting from the BFS algorithm is called a *breadth-first search tree*.

Algorithm 1 A general breadth-first search template.

Input: A directed or undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.

Output: A list D of distances of all vertices from s. A tree T rooted at s.

```
 1: Q ← [s]                                   ▷ queue of nodes to visit
 2: D ← [∞, ∞, ..., ∞]                        ▷ n copies of ∞
 3: D[s] ← 0
 4: T ← [ ]
 5: while length(Q) > 0 do
 6:     v ← dequeue(Q)
 7:     for each w ∈ adj(v) do
 8:         if D[w] = ∞ then
 9:             D[w] ← D[v] + 1
10:             enqueue(Q, w)
11:             append(T, vw)
12: return (D, T)
```

The breadth-first search algorithm makes use of a special type of list called a *queue*. Formally, a queue Q is a list of elements. At any time, we only have access to the first element of Q, known as the *front* or *start* of the queue. We insert a new element into Q by appending the new element to the *rear* or *end* of the queue. The operation of removing the front of Q is referred to as *dequeue*, while the operation of appending to the rear of Q is called *enqueue*. That is, a queue implements a first-in first-out (FIFO) protocol for adding and moving elements. As with lists, the *length* of a queue is its total number of elements.

Depth-first search

A depth-first search (DFS) is a graph traversal strategy similar to breadth-first search. Both BFS and DFS differ in how they explore each vertex. Whereas BFS explores the neighbourhood of a vertex v before moving on to explore the neighbourhoods of the neighbours, DFS explores as deep as possible a path starting at v. One can think of BFS as exploring the immediate surrounding, while DFS prefers to see what is on the other side of the hill. In the 19th century, Lucas and Tarry investigated DFS as a

strategy for traversing mazes. Fundamental properties of DFS were discovered in the early 1970s by Hopcroft and Tarjan [28].

Algorithm 2 formalizes the above description of depth-first search. The tree resulting from applying DFS on a graph is called a *depth-first search tree*. The general structure of this algorithm bears close resemblance to Algorithm 1. A significant difference is that instead of using a queue to structure and organize vertices to be visited, DFS uses another special type of list called a *stack*. A list $L = [a_1, a_2, \ldots, a_k]$ of k elements is a stack when we impose the same rules for element insertion and removal. The top and bottom of the stack are $L[k]$ and $L[1]$, respectively. The operation of removing the top element of the stack is referred to as *popping* the element off the stack. Inserting an element into the stack is called *pushing* the element onto the stack. In other words, a stack implements a last-in first-out (LIFO) protocol for element insertion and removal, in contrast to the FIFO policy of a queue. We also use the term *length* to refer to the number of elements in the stack.

Algorithm 2 A general depth-first search template.

Input: A directed or undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.

Output: A list D of distances of all vertices from s. A tree T rooted at s.

```
 1: S ← [s]                                   ▷ stack of nodes to visit
 2: D ← [∞, ∞, ..., ∞]                        ▷ n copies of ∞
 3: D[s] ← 0
 4: T ← [ ]
 5: while length(S) > 0 do
 6:     v ← pop(S)
 7:     for each w ∈ adj(v) do
 8:         if D[w] = ∞ then
 9:             D[w] ← D[v] + 1
10:             push(S, w)
11:             append(T, vw)
12: return (D, T)
```

3 The Project

> *I feel much better now, I really do.*
>
> **— HAL 9000**
> 2001: A Space Odyssey

In this chapter we describe in more details the platforms utilized and show the steps of making the project alive. We developed a self-service kiosk that can be voice activated, performs an solid search on the tree structure, all in a low cost single-board computer. Our focus with the project is to use clouding platforms to overcome hardware platform limitations. The result is an interactive kiosk that "understands" voice requests of informations, answering them with a variety of possibilities responses, all editable by the administrators.

3.1 Related Works

Related works on tree structure search can be drawn from the first attempt at building chatbots: ELIZA. *ELIZA* was created in the 60's by Joseph Weizenbaum with the objective to emulate a psychotherapist in clinical treatment [36, 37]. Another example used in this field is ALICE. *ALICE* [29], which was adapted from ELIZA program, was built to entertain users and talk to them as a real person. Both chatbots used a tree structure representation (knolewdge tree) which contained all patterns and templates. The pattern matching algorithm used was depth first search technique, as presented in Algorithm 2

3.2 The Platforms

Now we describe the platforms for utilized for the project, namely, Wit.ai, Metore.js, and Raspberry Pi.

3.2.1 Wit.ai

Wit.ai is *cloud service* that turns speech or text into actionable data. Wit is supported on every major platform and can be implemented in mobile apps, home automation,

wearable devices, robots, and messenger agents. Thus, it is a interesting *natural language processing* tool for developers.

History and Background

Natural language processing is the technology for dealing with a variety of human language in thousands of languages and varieties [13]. Since the last decade, every year we seem more of successful natural language processing applications on our day-by-day. Spelling and grammar correction in word processors, machine translation on the web, and email spam detection are some of the many applications of natural language processing that are present on our most common daily routine. Observing this potential, the online social networking service *Facebook* acquired Wit [38], a startup founded to create an API for building voice-activated interfaces.

Getting Started

Note that, in order to sign up on Wit.ai and create the voice application, Wit only allow access through a *GitHub* [8] account. All steps are drawn from [38] Quick Start Guide. For more details, please refer back to the original source.

The Console

Once with access to Wit.ai, one is already able to access to what is called the Wit Console. The Console is where we can manage the Wit.ai-powered apps, configure a voice app and improve it. Thus, on the Wit Console App is where the voice applications logic abides. For instance, the Wit Console with App $terminal - student$ is shown in Figure 3.2. Note that if a user signs in for the first time, Wit.ai creates the first app and the user will land on its page. We can create new apps from the top-right menu.

Determining and Creating User Intent

Given that the user has created a new app, it is time to determine the its intents. An intent is something that the end-user wants to perform. For example, “What’s Professor Xavier’s phone number?”, “show me Professor Xavier’s room number”, and “hello robot”. It is common to focus on a finite list of possible intents. Hence, each intent corresponds to one action in the app. For our application, the intent is to “change the colour of the object” and also“greetings.”

Once with a raw speech or text input, Wit.ai will determine what is the user intent.

For instance, all the following expressions should be mapped to the same intent:

```
1 ‘‘What’s Professor Xavier’s phone number?’’
2 ‘‘Professor Xavier’s phone number’’
3 ‘‘Show me the phone number of Professor Xavier’’
```

There are many different ways to express the same intents. Thus, it is the job of Wit.ai to map these expressions to actual intents.

The user can browse existing intents from the community of Wit online. When typing examples for the intents, Wit will suggest these existing intents. If one fits the intent, clicking on “GET” gets a copy of the existing intent in the Wit app. If there not exist an intent that fits, it is necessary to create a new intent.

To create a new intent, the user must type a name for your new intent in the “Name your intent” field. Usually, intent names try to match the app functions or methods. At least three expressions, even with synonymous way to say the same command, must be added. Figure 3.1 shows the intents $contacts$ and $event$ for the terminal-student App. Some of the expressions for $contact$ were previously mentioned.

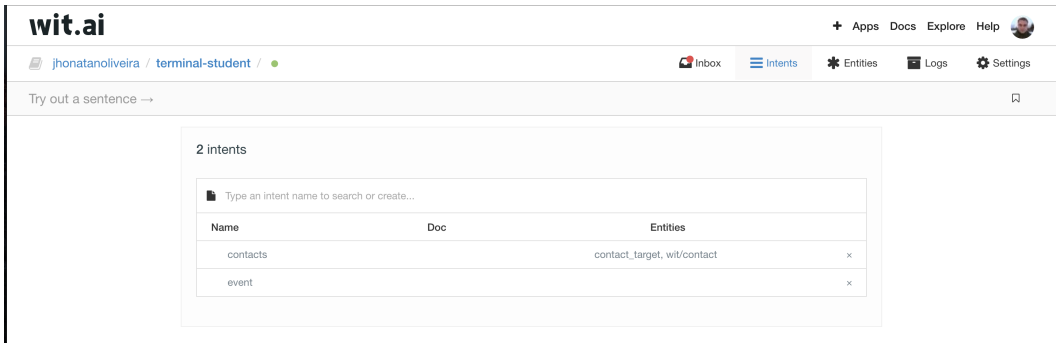

Fig. 3.1: The intents for the terminal-student App.

Training

It is time to query the voice app. At this point, we can already request the voice app via the Wit.ai API.

Notice that, before training the app, it is necessary to add the *Client Access Token* to the website. We will show more details about this step in Section 3.2.1.

In the Inbox of the App we can see the examples said when inputed trought web-site.

At Inbox we can validate the the correct sentences recorded by Wit. Once trained, Wit improves its speech recognitions an the *Confidence* ratio for the expression intents

tends to raise. To train the voice is very simple. For each expression it reproduces the audio recorded. If it is correct, we can validate it. If not, we type the correct sentence and then validate it.

At Inbox we can validate matches with expression and intents that were correctly captured by Wit. For instance, Figure 3.2 shows two expressions captured by Wit, the first one was validated as intent *contacts*, the second was *Archived* since was not any of the existing intents, and the third one was validated as intent *contacts*. In our example, we also want to capture the targeted phone number. This is called an *Entity*. We can create our own one or select a common one from the dropdown list. For instance, the entity *phone* was created for the intent *contacts*, as depicted in Figure 3.2.

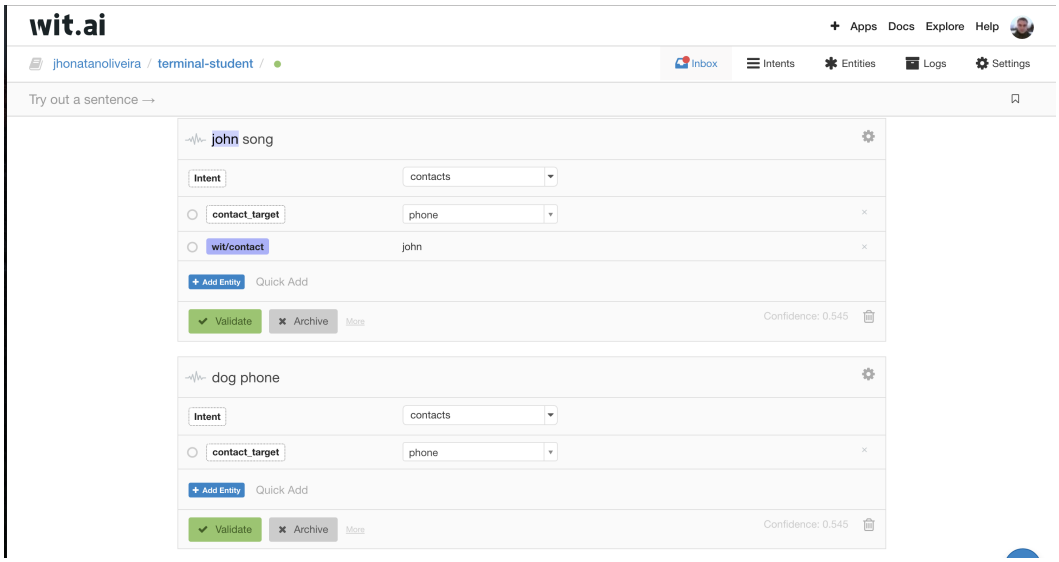

Fig. 3.2: Expressions captured by Wit.ai.

Web application

Now we show how to add Wit to a web app. As a prerequisites, the browser must support [35], which is the case of Chrome, Firefox and Opera.

To integrate the voice app with the web app we need to create a new folder for the app, download the *Web SDK* (the microphone app) and extract the archive:

```
1 mkdir myapp
2 cd myapp
3 curl -L <https://github.com/wit-ai/microphone/releases/download/0.7.0/
    microphone-0.7.0.tar.gz | tar xvzf -
4 mv microphone-* microphone
```

In the myapp folder, we must create a file $index.html$ containing the html provided by Wit on `https://wit.ai/docs/web/0.7.0`.

In the Setting page of the voice app we generate a *Client Access Token*, as illustrated in Figure 3.3 A client access token is unique and authorizes the domain to access the voice app. To use the client access token we must replace $CLIENT\_TOKEN$ on the $index.html$ file. For example, replace

```
mic.connect("CLIENT_TOKEN");
```

for the client access token of our running example, given

```
mic.connect("AWRBY6WLIPAQP7PGCGMQQKKO45LELCWO");
```

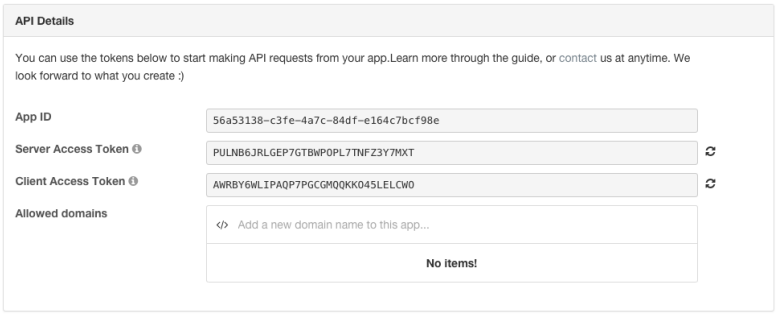

Fig. 3.3: A Client Access Token.

In Action

To see the web application in action we serve the app with a webserver. For instance, using Python:

```
python –m SimpleHTTPServer
```

Then, to load the page on the browser

```
http://localhost:8000
```

After allowing your microphone, we will be able to click on the microphone icon and say a command. The command will be streamed to the voice app.

Resource Constrains

3.2.2 Meteor.js

Meteor.js [18], or Meteor for short, is an open-source web platform to create applications using the Web 2.0 paradigmas. The platform provide a reactive approach

by focusing on the data flow. This is done by creating and managing events in the application. Also, data management is done with a non-SQL database called *MongoDB* [19]. Meteor simplify the application development by providing an unique programming language, javascript, throughout the whole stack process. Moreover, the platform makes available a set of common tools for business logic and data management. Finally, Meteor deploys the application in desktop and mobile without needing to change the source code.

History and Background

The Meteor platform was created by a company called Skybreak in 2011 [31]. Later, in 2012, the company changed its name to Meteor. The startup was incubated by YCombinator [11] and after receiving an investment of $11.2 M, the platform development increased considerably. The platform left beta in October 26th, 2015, with a version that currently provides multiplatform support.

Regarding its internal structure, Meteor is built on top of *Node.js* [21]. That means that Meteor is driven by events, in order to create an asynchronous model. This feature is implemented using callback functions: when an event happens a specific function is called to execute a portion of code. The server-side and client-side are both implemented using javascript. In the client-side, templates are used to design user interfaces. Here, a simple markup language defines the design and the events handled by the application. Meteor has support for more than one template language, being *Blaze* [18] the official one. In the server-side, Meteor handles data management using collections. The oficial non-SQL database supported is MongoDB, but new ones are being incorporated in future versions [16].

Getting Started

Now, we will follow a common flow for an app creation process. During this section, we will use a single running example which is an application for todo lists. This todo example can also be found on the official getting started tutorial in [18]. We will try to keep the same code conventions and names always that possible so the reader can use this paper as an extended guide for that tutorial. We assume that the Meteor program and required

The following command line is used to create a new application with Meteor:

```
1 meteor create todo
```

This creates a folder structure as shown below:

```
todo.js todo.html todo.css .meteor
```

The *todo.js* file is where the javascript code for the server and client stays. Here is where the code throws and handles events, besides business logic. Templates for user interface is done in the *todo.html* file with pure HTML markup language and a template markup language. For this project Blaze is the template languages used. The *todo.css* file contains styles for the templates. Lastly, the *.meteor* stores settings and the meteor aplication itself in a hidden folder.

Templates

Templates are defined using a special tag called *template*. Once defined, a template can be included within any HTML code. In this way, a template is a interface module that can be reuses wherever it is required. All the HTML code and the templates have access to the data made available in that part of the HTML code. The basic flow is: the javascript code makes available some data to a specific area of the HTML code (or a specific template), so the template language can access that data and print the ones of interest for the user. Bellow, we show a simple todo list interface.

```
<head>
  <title>Todo List</title>
</head>

<body>
  <div class="container">
    <header>
      <h1>Todo List</h1>
      <form class="new-task">
        <input type="text" name="text" placeholder="Type to add new tasks
    " />
      </form>
    </header>

    <ul>
      {{#each tasks}}
        {{> task}}
      {{/each}}
    </ul>
  </div>
</body>

<template name="task">
  <li class="{{#if checked}}checked{{/if}}">
    <button class="delete">&times;</button>
    <input type="checkbox" checked="{{checked}}" class="toggle-checked" /
    >
    <span class="text">{{text}}</span>
  </li>
```

```
28 </template>
```

Here, the HTML code defines in its *body* a title and a list. Notice that Blaze commands are defined within {{ and }}. In the list, the Blaze command *#each* goes through a list of data, as in a loop lace. The data variable *tasks* contains this list of data and was made available to this part of the code by the javascript code. The Blaze command > *name_of_template* prints a template previously defined with name *name_of_template*. In lines 22-28, the template named *task* is defined. This template expect to find available in its scope a data variable called *text*. In line 16, the template is included in that spot of the HTML code and the data variable *text* is available on the loop scope of the *#each* command. Note that *text* is a field within *tasks*.

On the javascript file we define code for the client and server. In order to distinguish between them, Meteors makes available a global variable called *isClient*, which goes to true if the running environment is the browser and goes to false if it is the Node.js one. If a code is called without being under the conditions of a client or server, the code is run in both environments. Follows the javascript code to create a simple database where the *tasks* are saved.

```
1  Tasks = new Mongo.Collection("tasks");
2
3  if (Meteor.isClient) {
4
5    // This code only runs on the client
6    Template.body.helpers({
7      tasks: function () {
8        return Tasks.find({});
9      }
10   });
11
12   Template.body.events({
13     "submit .new-task": function (event) {
14       // Prevent default browser form submit
15       event.preventDefault();
16
17       // Get value from form element
18       var text = event.target.text.value;
19
20       // Insert a task into the collection
21       Tasks.insert({
22         text: text,
23         createdAt: new Date() // current time
24       });
25
26       // Clear form
27       event.target.text.value = "";
28     }
29   });
```

```
30
31  Template.task.events({
32    "click .toggle-checked": function () {
33      // Set the checked property to the opposite of its current value
34      Tasks.update(this._id, {
35        $set: {checked: ! this.checked}
36      });
37    },
38    "click .delete": function () {
39      Tasks.remove(this._id);
40    }
41  });
42
43 }
```

Line 1 runs on the server and on the client, since it is not inside the conditional in line 3. In the server, that command creates a database called *tasks* if it does not exist already. In the client, it creates a cache of the same database where Meteor manages some saved data in order to reuse repeated queries from the user. From line 6 to 9, using the global variable *Template*, we make available to the *body* of the HTML whatever data the not named function in line 7 return. The function returns all the entries in the previously created *tasks* database. These returned data is available to the HTML code on a data variable called *tasks*, as determined in line 7.

Inserting Data

Regarding the data inclusion, this HTML code has a form which will receive the new input data from the user. Lines 9-11 defines a simple form where the user can input new tasks. When pressing enter, an event is thrown from the HTML code and can be handled in the javascript code.

The javascript code watch for an event for the form submission, as shown in line 13. From here, the default reaction of the browser, which is try to submit the form, is stopped. The, the value of the input text is saved in a variable. Next, the cached database variable *Tasks* is used to insert a new entry on the server database with the text saved from the user input. Finally, the text input is cleared.

Notice in the last insertion process that the cached database was used to update the server database. This is possible due to the way Meteor works in client and server. The client has only a cached version of the database. But whenever this cached version is updated, an event goes to the server (whenever there is connection available) making the server database to update. The other way around works in the same manner: if the server database is updated, an event goes to all client spreading the update.

Updating and Removing Data

If a task is done, we want to check it out from the list by updating its entry with a done flag. In the HTML code, line 23, the template *task* has a conditional statement checking for a variable called *checked*. If the variable is true, the body of the condition, which is just a string *checked*, is executed. This variable, as expected, is set in the javascript code. Indeed, in line 32, the javascript watch for an event of a click on the checkbox. If it happens, the cached database *Tasks* is updated by setting a field *checked* to the opposite value that the user entered. Notice the use of a special variable *this* which provides context for HTML access from where the event occurred.

The deletion process is similar to updating but a different function is used on the cached database. The javascript code, line 24, watch for the HTML code, line 24. When the user clicks in the delete button, the javascript code is executed by removing the correspondent id of the entry. Again, note the use of *this* as a context variable for the entry where the event occurred.

Resource Constrains

Meteor can be used to build resource constrain products. In this type of application, the amount of processing and memory are often restricted. Therefore, the use of parallel computing or decentralized tasks are common and necessary. Meteor can be use to avoid heavy load computation in the limited hardware itself. A server can be used to process all the required heavy workload while the client simply show in real time the results.

One way of providing an interface to a Meteor server side is by using the official server-client protocol, called *Distributed Data Protocol* (DDP). The DDP protocol is a simple specification on how other languages can communicate with a Meteor server.

3.2.3 Raspberry Pi

The *Raspberry Pi* a low cost is single-board computer developed in the UK by the Raspberry Pi Foundation with the intention of stimulating the teaching of basic computer science in schools [25]. It is a capable small device that is capable of doing everything a desktop computer does, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games [7].

The Raspberry Pi has a Broadcom BCM2835 system on a chip (SoC), which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and 256 megabytes of RAM. It uses an SD card for booting and long-term storage. The Raspberry Pi 2 Model B is the second generation Raspberry Pi. It replaced the original Raspberry Pi 1 Model B+ in February 2015. Compared to the Raspberry Pi 1 it has: (i) A 900MHz quad-core ARM Cortex-A7 CPU and (ii) 1GB RAM [7]. The Raspberry Pi Model A (left) and Model B (right) board are shown in Figure 3.4.

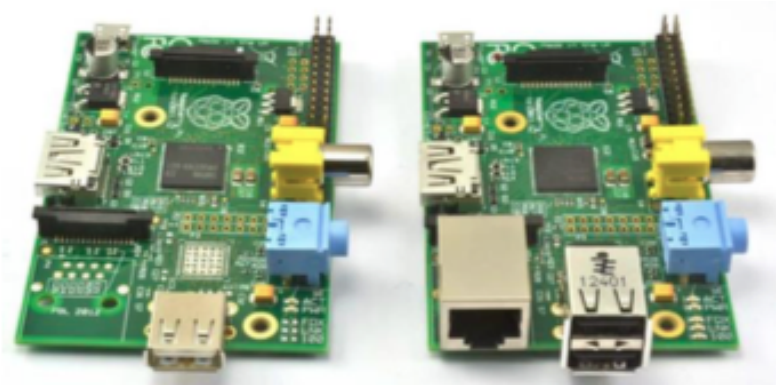

Fig. 3.4: The Raspberry Pi Model A (left) and Model B (right) board [7].

History and Background

In 2006, early concepts of the Raspberry Pi were based on the Atmel ATmega644 microcontroller [39]. The Model A was not the first Raspberry Pi to hit general availability – that distinction went to the Model B. The A was, nevertheless, the truest to the bare-bones ethos of the Pi project, retailing for $25 and initially packing just 128MB of memory. (Later raised to 256MB.)

Getting Started

All steps are drawn from [7] Raspberry Pi 2 Model B product documentation. For more details, please refer back to the original source.

Some components are required in order to have a functioning Raspberry Pi device. It is required: a SD Card (the newer Raspberry Pi Model A+, Raspberry Pi Model B+, Raspberry Pi 2 Model B, Raspberry Pi Zero, and Raspberry Pi 3 Model B require micro SD cards); display and connectivity cables; keyboard and mouse; and power supply. It is not essential, but helpful to have internet connection to update or download software. To connect your Raspberry Pi to the internet, it can be done either via an Ethernet cable or a WiFi adapter.

Plugging in the Raspberry Pi

Before plugging anything into the Raspberry Pi, it is necessary to be made sure that we have all the equipment listed above to hand. Once confirmed, the following instructions apply: (i) Begin by slotting the SD card into the SD card slot on the Raspberry Pi, which will only fit one way. (ii) next, plug the USB keyboard and mouse into the USB slots on the Raspberry Pi. (iii) Make sure that the monitor or TV is turned on, and that the right input (e.g. HDMI 1, DVI, etc.) have been selected; (iv) then, connect the HDMI cable from your Raspberry Pi to the monitor or TV. (v) If its is intended to connect the Raspberry Pi to the internet, plug an Ethernet cable into the Ethernet port next to the USB ports, otherwise skip this step. (vi) When all the required cables and SD card have been plugged in, plug in the micro USB power supply. This action will turn on and boot your Raspberry Pi. (vii) If this is the first time the Raspberry Pi and NOOBS SD card have been used, then its is necessary have to select an operating system and configure it. Follow the NOOBS guide to do this.

For more information on settings and implementations tips, please read the official documentation (https://www.raspberrypi.org).

Resource Constrains

We now list the main disadvantages commonly listed of a Raspberry Pi [34]. The first one is linked to a power constrain. A Raspberry Pi does not have a real-time clock (RTC) with a backup battery. However, this problem can easily work around using a network time server, and most operating systems do this automatically.

Not necessarily a disadvantages, but it can be seen as one, a Raspberry Pi does not have a Basic Input Output System (BIOS) so it always boots from an SD card. Also, it does not support Bluetooth or WiFi out of the box but these supports can be added by USB dongles.

Finally, it has a relatively small number of digital I/O. This problem can be easily addressed by expanding with external logic devices.

3.3 Implementation

3.3.1 Description of design process

The design process of our project is based on three different platforms, including two cloud computing ones, namely Wit.ai and Meteor.js, and one hardware, Raspberry Pi. The overview design is a system implemented in Meteor.js containing a server and a client. Here, the server is a more powerful computer, capable of processing exhaustive algorithms. Thus, the server is used as a central processing unit and only forward responses for the client's processing requests. The client is considered a less powerful computer, which receives orders from the user and send processing requests to the server. This design process is managed by Meteor.js by using its clear division between client and server code.

The main idea in this division is to allow the client to make calls to functions implemented inside the server. These functions, in the other hand, returns plain text in json format to the client. In this way, a client would require only processing power enough to make the request. Our project focus on the Raspberry Pi. Then, whenever a processing algorithm requires too much power, we transfer that to the server side of the code.

The requests from the user is treated by the Wit.ai platform. We use its natural language processing capabilities to understand the user's query and send the proper request to the server. This processing is done in a search within a knowledge tree, also saved on the server.

3.3.2 Discussion of group contributions

On January, the project was discussed by classmates and professor and procedures and deadlines were arranged. Our group consisted of 2 students: André E. dos Santos and Jhonatan S. Oliveira. Our goal was to present the most important information from our project implementation of a self-service kiosk. As a group, we wanted to address key points while relating them to our learning process of the platforms involved.

Our first step in achieving this goal was to meet regurlarly (in our research lab). In those meetings, we first threw out ideas on what we were thinking regarding the project. André bought the Raspberry Pi and was in charge of having Wit.ai running. Jhonatan implemented the administrator system on meteor.js. The implementation precesses, assembly and testing were performed together. The same can be stated for the writing process of this report.

When problems were encountered, the decision were done after a brief discussion, always following guide lines of the instructors Dr. Gerhard and Mr. Tomesh. Those casual meetings with the instructor were biweekly arranged after classes.

Some channels of communications played important roles on the assembly of this project. First, the version control tools Github. Our work were safely administrated and saved in a shared repository. However, other channels like Slack and Google Drive were constantly used.

Our group worked very well together. Each person expressed their ideas with a constant open line of communication. Even though the feature of indoors localization were not able to be applied, we achieved our goal of implementing a self-service kiosk (university level). Overall, the project was a success.

3.4 Results

The system was implemented successfully with only one change in the original idea, which is not crucial to this demonstration. Our focus with the project was to use clouding platforms to overcome hardware platform limitations. In this way, we could make a system that interact with the user by a terminal client using a Raspberry Pi, and replies to the user by processing the query on the server side. The only feature we could not implement was the indoor navigation capability. This failure in our plans is further described in Section 4, as we also discuss the context and alternatives that we tried in more detail. Now, we describe and show our implemented system.

The administrator log in the system using its unique account. For now, this is fixed and set automatically by the system on startup. For test purposes, the current username is *admin* and password *cs8072016*. The welcome page, shown in Figure 3.5, means a successful log in. This page has no useful information, but it was created thinking in a possible dashboard with overview statistics of the whole system.

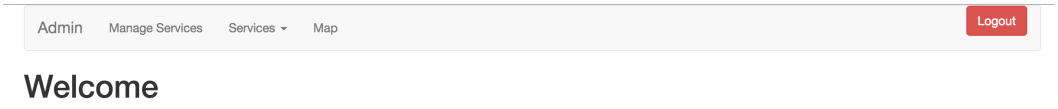

Fig. 3.5: Welcome page at the administrative side of the system. Future work might include a dashboard at this page.

Adding services to the system can be done by using the top menu bar. In “Manage Services”, depicted in Figure 3.6, the administrator can create and delete services that are available for the user. These services are of two types: data services and external services. The data service is internally described by a knowledge tree. All corresponding knowledge trees can be edited in this same administrator screen (as

described next). The external service is configured in a hash map in code. This is a flexible way of adding external APIs within the system. Whenever the administrator wants to make an API available to the user, one more pair of "key-value" needs to be added to that hash map. Figure 3.7 shows a popup used to add both type of services.

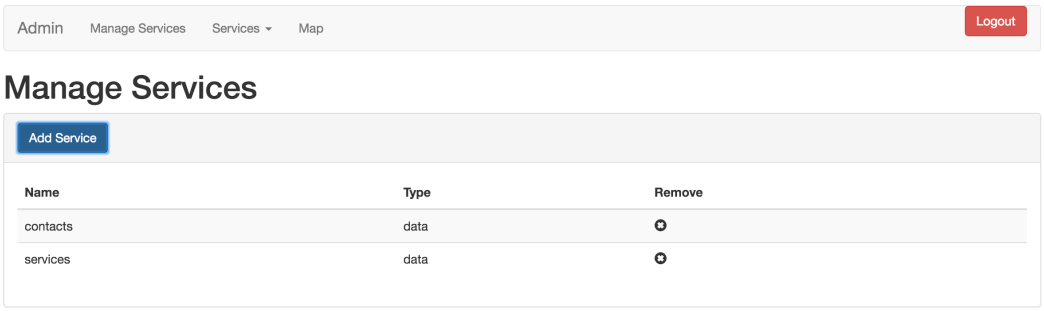

Fig. 3.6: Add and remove services within the System. There are available two types of services: data and external. The former is described by a knowledge tree, while the later is described by an external API.

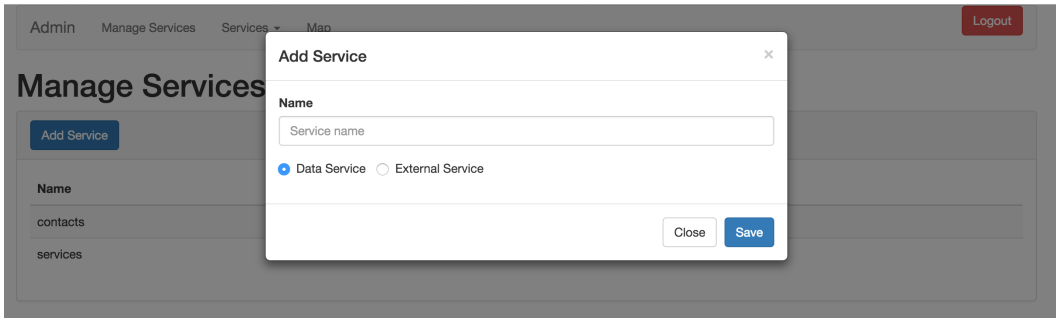

Fig. 3.7: Popup to add a new service.

Now, the data service type can be fully edited inside this administrator screen. As illustrated in Figure 3.9, the administrator has a visual tool to expand/contract the branches of the tree. Using the tools in the right hand side, nodes can be added, removed and its label edited. Notice that removing a node also removes all its descendants. A sandbox for testing user queries is also added within this screen. In this way, the user can test a query on the fly, just like the user would ask. The answer is then depicted in the knowledge tree in the left hand side. Since this knowledge tree can always be edited, these available tools can be powerful for modelling a whole problem domain and testing them at the same time.

Lastly, a setting page for the indoor navigation was also created. As aforementioned, this feature is not available in this version of the system due to technical problems faced during the implementation. (For more details, see Section 4). Anyway, the configuration page allows the administrator to set sizes of the map presented to the user, as well as the accuracy used when answering localization queries.

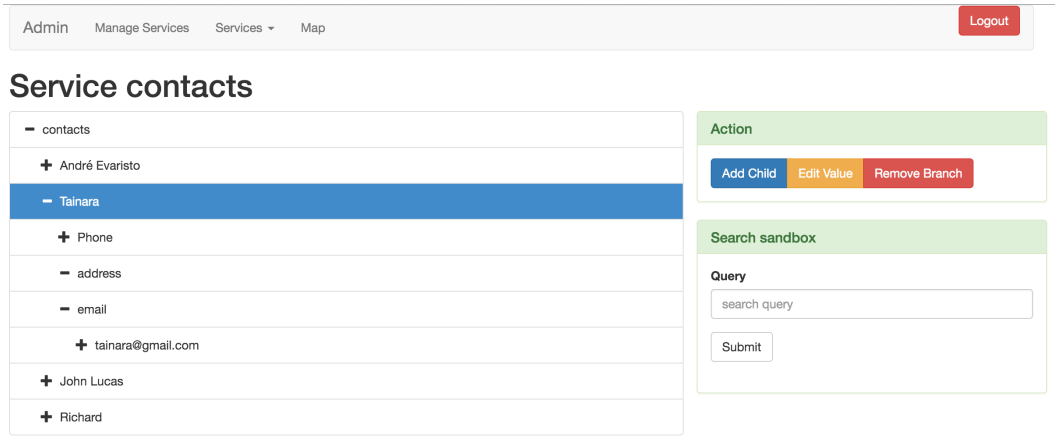

Fig. 3.8: Editing a data service within the administrator screen: the knowledge tree can be fully edited visually and a search query can be tested on the fly.

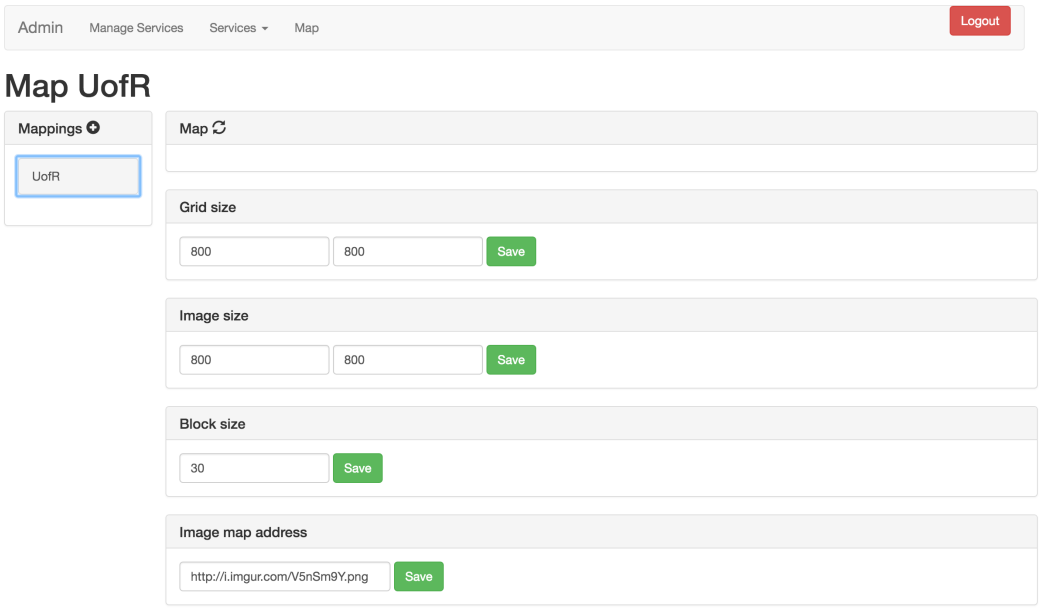

Fig. 3.9: Future works include indoor navigation capabilities allowing the administrator to set up this service from within the administration screen.

4 The Ex-Project

> *Funny, how just when you think life can't possibly get any worse it suddenly does.*
>
> **— Marvin**
> (The Hitchhiker's Guide to the Galaxy)

The initial plan for our final project included a service inside the app for indoor navigation. The idea was to allow the user to ask for a location within the University of Regina (UofR) main campus and the app would trace a route from where the device hosting the app is to where the user requested. Indoor navigation itself is a complex and well known task problem in computer science and engineering. Our primary focus was to use some already available solution for indoor navigation, instead of trying to come up with our own solution. In this way, we wanted to show how resource constrain devices can still be used to provide such service by using cloud computing. Our hardware platform, a Raspberry Pi, would need a GPS chip for self localization. Then, we considered using a simple IC shown in Figure 4.1.

We did a broad research on publicly available solutions for indoor navigation, including paid, free or open source ones. For the commercial options, the most accessible options that we found was *Insoft* [9] and *InDoors* [10]. They both have well documented SDKs and support for multiple platforms, which is crucial for a variety of real world applications. The disadvantage of them is, first, the price which might be expensive for small projects/companies. And second, the use of specialized hardware to operate in full potential. Indoors, for instance, although has support for wireless mapping, recommends the use of beacons for better localization. These small devices are low energy cost bluetooth enabled which can communicate with smartphones. By using this signals, Indoors makes indoor navigation more accurate.

We could find two open source options that we considered viable for a real world implementation. Namely, *OpenStreetMap* [22] and *AnyPlace* [3]. The former has a internal project for indoor navigation[1] which maps the target location. The project even has a client called *OpenLevelUp* [24], which runs an implementation of it. Although the client is open source, the implementation is very specific for using

[1] http://wiki.openstreetmap.org/wiki/Indoor_Mapping

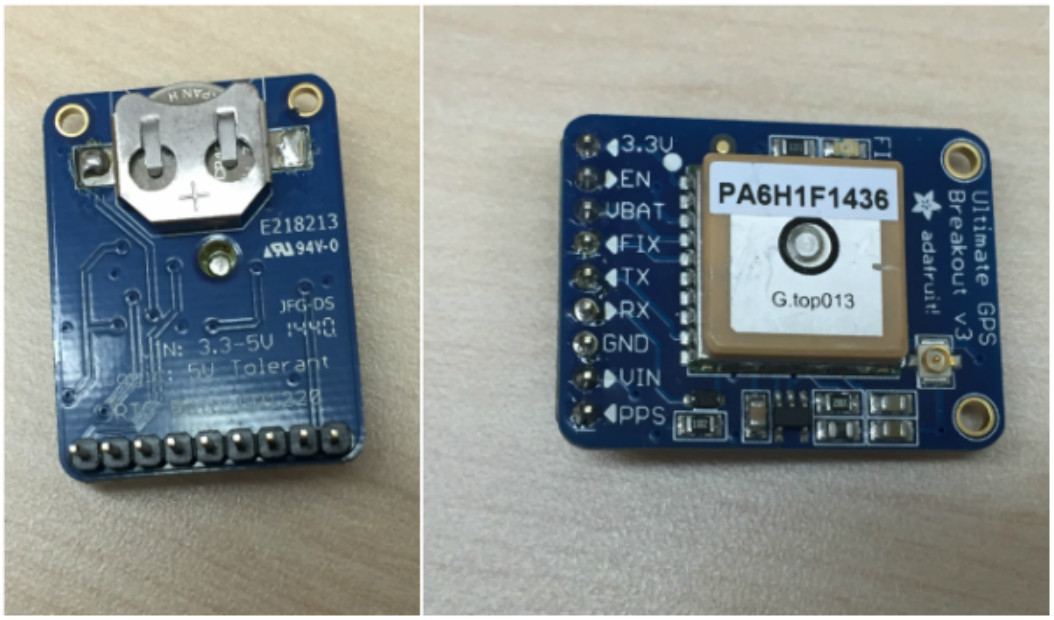

Fig. 4.1: GPS integration for a Raspberry Pi.

the mapped location from OpenStreetMap. That is, it would be very hard to adapt that client in order to run a customized mapping made at the UofR. This required workload may not pay off for this prototype. Then, we decided to give a try on AnyPlace.

Anyplace [3, 6] is an open source alternative for indoor navigation. Their technology have won three awards at conferences in indoor navigation, including the 1st place at Evaluation of RF-based Indoor Localization Solutions for the Future Internet (EVARILOS Open Challenge), European Union. The project is structured into three pieces: the Viewer, the Architect, and the API. All of them work around a common database, saved at the project server. The Viewer is basically a client which can read from the database of mapped locations and exhibit these informations in a user friendly fashion. In order to do that, the Viewer uses a layer over the Google Maps application. Locations and their points of interest (POI) are shown with specific icons and information on this layer. The Architect is a web application designed to facilitate the entrance of information in their database. Using this interface, a user might define a new location in the map (again using Google Maps on the background) and its related POIs. After saving this entry at Architect, the database use by all clients (including the Viewer) are automatically updated. Lastly, the API is a web interface for reading information from the database and sending them though HTTP requests. General apps can be built by only using this API.

We decide to use AnyPlace for the localization service inside our final project application. We used a map from the UofR provided at the official website to upload the

main floor plan in the Architect. With the map online, we define several common POIs around the campus, including Tim Hortons, BOB, Subway, among others. The Architect also allows the creation of path among these POIs, which facilitate the task of finding the best path. The next step is to use a smartphone app to create a mapping of the location entry in the Architect app. This is done by mapping wireless connections and the feedback from the user throughout the paths. The last step would be to create a simple client to Anyplace inside our map. In this way, whenever the user ask a location question, we would be able to show a map and the correspondent path. Thus, we looked at the source code for the Viewer, which is a more robust client, looking for inspiration. Although we could get the beginning of the client running, that is a basic integration with the Anyplace API, we got stuck on the Google Maps integration. The problem was with creating the layer over Google Maps to show all of our required information. This is done within the Viewer and we tried to understand from there, but they used a different framework (angular.js) for implementing this task, while we use Meteor.js. That is, we could not use their solution in our application.

We decided to do not go further with the indoor navigation due to its complexity requirements for implementation. This can be motivation for future works in this project. Indeed, our current application is flexible enough for incorporating any third part services, including indoor navigation.

5 Conclusion

An interactive kiosk contains specialized hardware and software designed within a public exhibit. As a computation system, a interactive kiosk systems has specific constrains associated with its components. Thus, an interactive kiosk must address resourceful solutions to minimize its possible constrains. In this project report we have shown an interactive kiosk with speech recognition for academic environments. We utilized three different platforms, including two cloud computing ones, namely Wit.ai and Meteor.js, and one hardware, Raspberry Pi.

Our focus with the project was to use clouding platforms to overcome hardware platform limitations. For such, spcialized concepts were required as reactive programming and graph data structures. In this way, we could make a system that interact with the user by a terminal client using a Raspberry Pi, and replies to the user by processing the query on the server side. These services are of two types: data services and external services. The data service is internally described by a knowledge tree. All corresponding knowledge trees can be edited in this same administrator screen. Basic methods for searching graphs such as DFS were essential for traversing the tree for proper information retrieval.

The initial plan, however, included a service inside the app for indoor navigation. Indoor navigation itself is a complex and well known task problem in computer science and engineering. We did a broad research on publicly available solutions for indoor navigation, including paid, free or open source ones. After several tests, we decided to do not go further with the indoor navigation due to its complexity requirements for implementation. Even though this application were not able to be applied, we achieved our goal of implementing a self-service kiosk (university level).

6 Appendices

Templates

Follows the template for the service page. This template makes the editing of the knowledge tree more reactive. Due to space limitation, we will not show the other templates.

```
function drawTree() {
    Meteor.call("getDataTree", Session.get("serviceId"), function(error, result){
        if (error) {
            bootbox.alert("Error retrieving the nodes. Please, try again.")
        } else {
            $('#tree').treeview({data: result});
        }
    });
}



Template.serviceTemplate.onRendered(function(){
    var data = []
    drawTree();
});


Template.serviceTemplate.events({
    "click .add-child": function(e) {
        var selectedNodes = $('#tree').treeview('getSelected');
        if (selectedNodes.length > 0 ) {
            var selectedNode = selectedNodes[0];

            bootbox.prompt("What is the value of the node?", function(result) {
                if (result) {
                    Nodes.insert({serviceId: Session.get("serviceId"), parent: selectedNode._id, value: result});
                    drawTree();
                }
            });
        } else {
```

```
            bootbox.alert("Please, select a node first.");
        }
    },

    "click .remove-branch": function(e) {
        var selectedNodes = $('#tree').treeview('getSelected');
        if (selectedNodes.length > 0 ) {
            var selectedNode = selectedNodes[0];

            bootbox.confirm("Are you sure that you want to remove this node and ALL its descendants?", function(result) {
                if (result) {
                    Meteor.call("removeBranch", selectedNode._id, Session.get("serviceId"));
                    drawTree();
                }
            });
        } else {
            bootbox.alert("Please, select a node first.");
        }
    },

    "click .edit-value": function(e) {
        var selectedNodes = $('#tree').treeview('getSelected');
        if (selectedNodes.length > 0 ) {
            var selectedNode = selectedNodes[0];

            bootbox.prompt({
                title: "What is the NEW value of the node?",
                value: selectedNode.text,
                callback: function(result) {
                    if (result) {
                        Nodes.update(selectedNode._id, {$set: {value: result}});
                        drawTree();
                    }
                }
            });
        } else {
            bootbox.alert("Please, select a node first.");
        }
    },

    "submit .search-form": function(e) {
        e.preventDefault();
        if (e.target.query.value.length > 0) {
            Meteor.call("searchQuery", e.target.query.value, function(error, result){
                if (error) {
                    bootbox.alert("Problem while processing query. Please, try again.");
```

```
                } else {
                    $('#search-tree').treeview(
                        {
                            data: result,
                            showBorder: false
                        }
                        );
                }
            });
        } else {
            bootbox.alert("Please, type in a query.");
        }
    }
});
```

Server side methods

These are methods implemented on the server side which can be called from the client. These methods perform graph search and string manipulations and sends back only the final answer to the client.

```
Meteor.methods(
{
  removeAllNodes: function(serviceId) {
    if (this.userId) {
      Nodes.remove({serviceId: serviceId});
      return true;
    } else {
      return false;
    }
  },

  getDataTree: function(serviceId) {
    var data = [];

    if (this.userId) {

        // Breadth-first search
        var queue = [];
        var root = Nodes.findOne({serviceId: serviceId, parent: ""});
        var currentTreeNode = {"text": root.value, _id: root._id, nodes:
    []};
        data.push(currentTreeNode);

        queue.push(currentTreeNode);

        while(queue.length > 0) {
            currentTreeNode = queue.pop();
```

```
            Nodes.find({serviceId: serviceId, parent: currentTreeNode._id
  }).forEach(function(post){
                var childTreeNode = {"text": post.value, _id: post._id,
  nodes: []};
                currentTreeNode.nodes.push(childTreeNode);
                queue.push(childTreeNode);
            });
        }
    }
    return data;
},

removeBranch: function(nodeId, serviceId) {
    if (this.userId) {
        // Breadth-first search
        var queue = [];
        var currentTreeNode = nodeId;

        queue.push(currentTreeNode);

        while(queue.length > 0) {
            currentTreeNode = queue.pop();
            Nodes.remove({_id: currentTreeNode, serviceId: serviceId});
            Nodes.find({serviceId: serviceId, parent: currentTreeNode}).
  forEach(function(post){
                var childTreeNode = post._id;
                queue.push(childTreeNode);
            });
        }
      return true;
    } else {
      return false;
    }
},

searchQuery: function(query, serviceId) {
    var bestNodes = [];
    if (this.userId) {

        // Breadth-first search
        var queue = [];
        var root = Nodes.findOne({parent: ""});
        var currentTreeNode = {_id: root._id, points: 0, confidence: 0};

        queue.push(currentTreeNode);

        var hasNewBestNode = false;
        while(queue.length > 0) {
            currentTreeNode = queue.pop();
            Nodes.find({parent: currentTreeNode._id}).forEach(function(
  post){
```

```
74              var newPoints = 0;
75
76              // Matching function: Using FuzzySet library from http://
    glench.github.io/fuzzyset.js/
77              // This library computes a score based on the Levenshtein
    distance
78              var CONFIDENCE_LEVEL = 0.2;
79
80              var fuzzySet = FuzzySet([post.value]);
81              var foundValue = fuzzySet.get(query);
82              if (foundValue && foundValue[0][0] > CONFIDENCE_LEVEL) {
83                newPoints = currentTreeNode.points + 1;
84
85                hasNewBestNode = true;
86              }
87
88              var childTreeNode = {_id: post._id, points: newPoints,
    confidence: foundValue ? foundValue[0][0] : 0};
89              queue.push(childTreeNode);
90
91              // Update best nodes
92              if(hasNewBestNode) {
93                if ((bestNodes.length == 0) || (newPoints == bestNodes
    [0].points)) {
94                  bestNodes.push(childTreeNode);
95                } else if(newPoints > bestNodes[0].points) {
96                  bestNodes = []
97                  bestNodes.push(childTreeNode);
98                }
99                hasNewBestNode = false;
100             }
101
102           });
103       }
104
105
106       // Breadth-first search to get all descendants of the best nodes
107
108       var data = [];
109
110       var queue = [];
111
112       for (var i = 0; i < bestNodes.length; i++) {
113         var n = Nodes.findOne({_id: bestNodes[i]._id});
114         var currentTreeNode = {_id: n._id, text: n.value, nodes: [],
    confidence: bestNodes[i].confidence};
115         data.push(currentTreeNode);
116
117         queue.push(currentTreeNode);
118       }
119
```

```
120
121         while(queue.length > 0) {
122             currentTreeNode = queue.pop();
123             Nodes.find({parent: currentTreeNode._id}).forEach(function(
    post){
124                 var childTreeNode = {_id: post._id, text: post.value,
    nodes: []};
125                 currentTreeNode.nodes.push(childTreeNode);
126                 queue.push(childTreeNode);
127             });
128         }
129
130     }
131
132     return data;
133   }
134 }
135 );
```

Note

Due to space limitation, only two files were shown in this section. But the full source code for the application and this latex paper are available at the repository of this final project in *https://github.com/jhonatanoliveira/cs807-final-project*.

Bibliography

[1] Federal disability reference guide. Human Resources and Skills Development Canada, Gatineau, Québec (2012)

[2] Allied Business Intelligence, I.: The number of interactive kiosks in operation will nearly double to three million by 2016 (2016), https://www.abiresearch.com/press/the-number-of-interactive-kiosks-in-operation-will/

[3] AnyPlace: Anyplace | indoor information service (2015), http://anyplace.cs.ucy.ac.cy/

[4] Berge, C., Minieka, E.: Graphs and hypergraphs, vol. 7. North-Holland publishing company Amsterdam (1973)

[5] Bonér, J., Farley, D., Kuhn, R., Thompson, M.: The reactive manifesto (September 2014), http://www.reactivemanifesto.org

[6] Demetrios Zeinalipour-Yazti, Christos Laoudias, K.G., Chatzimiloudis, G.: Internet-based indoor navigation services. IEEE Internet Computing

[7] Foundation, R.P.: Raspberry pi 2 model b (2016), https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[8] GitHub: Github · where software is built (2016), https://github.com/

[9] infsoft GmbH: Indoor navigation & indoor positioning by infsoft (2016), http://www.infsoft.com/

[10] indoo.rs GmbH: indoo.rs (2016), http://indoo.rs/

[11] Griffith, E.: First github, now meteor: Andreessen horowitz backs another developer favorite (July 2012), https://pando.com/2012/07/25/first-github-now-meteor-andreessen-horowitz-backs-another-developer-favorites/

[12] Joyner, D., Van Nguyen, M., Cohen, N.: Algorithmic graph theory. Google Code (2010)

[13] Jurafsky, D., Martin, J.H.: Speech and language processing. Pearson (2014)

[14] Kelsen, K.: Unleashing the power of digital signage: content strategies for the 5th screen. CRC Press (2012)

[15] Kubitza, T., Pohl, N., Dingler, T., Schneegass, S., Weichel, C., Schmidt, A.: Ingredients for a new wave of ubicomp products. IEEE Pervasive Computing (3), 5–8 (2013)

[16] Lardinois, F.: Meteor acquires yc alum fathomdb for its development platform (October 2016), http://techcrunch.com/2014/10/07/meteor-acquires-yc-alum-fathomdb-for-its-web-development-platform/

[17] Lee, C.Y.: An algorithm for path connections and its applications. Electronic Computers, IRE Transactions on (3), 346–365 (1961)

[18] Meteor: (2016), https://www.meteor.com

[19] Mongo: (2016), https://www.mongodb.org

[20] Moore, E.F.: The shortest path through a maze. Bell Telephone System. (1959)

[21] Node.js: (2016), https://nodejs.org

[22] OpenStreetMap: Openstreetmap (2016), http://www.openstreetmap.org/

[23] O'Reilly, T.: Design patterns and business models for the next generation of software (September 2005), http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html

[24] PAVIE, A.: Openlevelup! (2015), http://openlevelup.net/

[25] Pi, R.: Raspberry pi. Raspberry Pi 1 HDMI 13 Secure Digital 34 Universal Serial Bus 56 Python (programming language) 84 p. 1 (2012)

[26] Rigby, D., Bilodeau, B.: Management tools & trends 2015. London, Bain & Company (2015)

[27] Satish, K., Viswanadham, Y., Priya, I.L.: Money to atm–fake currency detection

[28] Schrijver, A.: On the history of the shortest path problem. Documenta Mathematica pp. 155–167 (2012)

[29] Schumaker, R.P., Ginsburg, M., Chen, H., Liu, Y.: An evaluation of the chat and knowledge delivery components of a low-level dialog system: The az-alice experiment. Decision Support Systems 42(4), 2236–2246 (2007)

[30] Sharma, S., Tim, U.S., Wong, J.S., Gadia, S.K., Sharma, S.: A brief review on leading big data models. Data Science Journal 13, 138–157 (2014)

[31] Skybreak: (2012), http://info.meteor.com/blog/skybreak-is-now-meteor

[32] Strauch, C.: NoSQL Databases. Stuttgart Media University (2012)

[33] Topcoder: Data science tutorials (2016), https://www.topcoder.com/

[34] Vujovic, V., Maksimovic, M.: Raspberry pi as a wireless sensor node: Performances and constraints. In: 37th International Convention on Information and Communication Technology, Electronics and Microelectronics. pp. 1013–1018. IEEE (2014)

[35] WebRTC: Webrtc home | webrtc (2016), https://webrtc.org/

[36] Weizenbaum, J.: Eliza—a computer program for the study of natural language communication between man and machine. Communications of the ACM 9(1), 36–45 (1966)

[37] Weizenbaum, J.: Contextual understanding by computers. Communications of the ACM 10(8), 474–480 (1967)

[38] Wit.ai: Wit — natural language for developers (2016), https://wit.ai/

[39] Wong, G.: Build your own prototype raspberry pi minicomputer. Ubergizmo, November 2 (2011)