COMP 529 - Software Architecture
Project Report: BusTracker

Amr Mallah
Mohammad Usman Ahmed

Table of Contents

1. Overview of BusTracker
2. The COTS components
3. Strategies to ensure COTS functionality
4. Overview of the final architecture
5. Architecturual trad-offs in the BusTracker application
6. Integration issues
7. Framework
8. Deployment Issues/Decisions
9. Future work

1. An overview of BusTracker

The goal application to be developed is a web based application, which will allow for a real-time visualization/simulation on a map, of the estimated dynamic bus movement according to a bus schedule that is available on the website of the Montreal public transport society (stm.info). The basic motivation behind this project was to provide a more user friendly method of checking the bus schedules;
The stm website provides a poor user interface to check the schedule, requiring the user to navigate through 3-4 pages before getting to the desired route stop, and then conducting another read and comparison to the current time to be able to figure out the current bus location, or the next time it will get to a specific stop.

The overall application functionality is, given a bus number and a direction, the system would check the bus schedule that is posted on the Montreal transport society (stm.info) website and examine all the stops on that route and after processing the info from the website, it would display the locations of buses currently running on that route.

Users would also have the option to subscribe (simulate a dynamic movement) to a route while viewing it so that they don't have to request updates at later time and can see the buses moving from stop to stop, simulating actual bus movements on that route. They would also be able to plot the bus route on the map, and choose a different time to check where the buses will be at any given time.

2. The COTS components

A description of each of the component used and their relevant features for the BusTracker application are the following:

1. Google Maps:

- **Description:** Google maps is web based service software that is provided by google, and provides a wide range of advanced features. It is available for public use through a publicly exposed API that provides, The Google Maps API, by embedding Google Maps in web pages with the use of JavaScript. The API provides a number of utilities for manipulating maps and adding content to it through a variety of services.
- **Features for use :**
    - displaying a map embedded in an HTML page .
    - provide geocoding (latitude/longitude positioning) services for specified addresses.
    - display icons (to represent buses) at a specific location on the map
    - drawing poly-lines to represent the bus route on the map.

2. HTML Parser

- **Description:** "HTML Parser is a Java library used to parse HTML in either a linear or nested fashion. Primarily used for transformation or extraction, it features filters, visitors, custom tags and easy to use Java Beans. It is a fast, robust and well tested package." The parser used is an open source project. It parsed the html page into a tree of different types of nodes with text at its leaves. The extraction of data after parsing required extraction of certain types of nodes from the list of nodes and grabbing the embedded text at its leaf-nodes.
- **Features for use :**
    - Extraction of the data from the website using the features:
        - Node list of different types of tags extractable from the parsed tree.
        - text extraction out of HTML files
        - link extraction out of HTML files

3. Bus schedule website (stm.info)

- **Description:** The Montreal transport society website (stm.info) provides scheduling, maintenance and other important information about the main public transport means in the city

of Montreal, through a static html website that has a non-changing structure.

- **Features for use :**
    - Provides (in HTML web page format)
        - the list of all available buses routes.
        - the list of bus stops for both directions for a given bus route.
        - the schedule of when the buses arrive at a given stop on a bus route.

3. Strategy used to ensure the COTS functionality

Google Maps:

- To establish a decent level of confidence the project has started by implementing a proof of concept for the most essential feature of the system which is displaying the bus icon at specific locations on a map and to update its locations with time, by hard coding values to simulate movements between several positions on a google map.
- The extensive documentation that the google maps api website provides, allows for better validation and testing .
- The Component in this case exposed its API, and was used as a Grey Box
- The simplicity and isolation of the google maps API allowed for an easier validation of functionality, especially with its visually verifiable behaviour in most cases, e.g.
    - Markers placed at the returned addresses could be verified directly on the Map
    - Simulated movement of markers could be verified by checking the time it took from one intersection to the next on the map and comparing that to the time as given on the STM schedule.

HTML Parser:

- The parser provided with easy access to the text contained in a web page but since in our case, the required text was spread all over the web page in rows and columns in multiple tables with only one or two words in one place and thus separated from each other by multiple tags, the exact extraction of the required data turned out to be by trial and error where the extracted children from the Node list of a particular kind of tag (row & column tags - td & tr) had to be individually examined until the right ones were found. This had to be done by making a match between the text extracted from the a given node and the expected text read from the web page and storing the right index values.
- This kind of parser would have best worked for modifying html tags of a web or extracting large pieces of text form a web page, but for the purpose of this project, significant work was required to extract the right nodes. The Node List form of the parsed data as stored by the parser made the search for the right text much easier.
    - To ensure that we could make this work with the other components, we tested a sample schedule by printing the extracted data from the website and retrieval individual intersections by performing search on the Node Lists.
    - The requirements from this COTS was for it to allow us to be able to
        - re-create an address after having extracted the individual street names from a bus-stops web page
        - extract the arrival times for the stops from a schedule page.
    - We were able to ensure this requirement by a thorough print and compare strategy by using various retrieval procedures on the Node Lists created by the parser.

Bus schedules website (stm.info):

- This was the most unchangeable yet important part as the entire information required for the software was based on the available data on this website.
- The website is being produced and maintained by the STM (Montreal transport society), any change in the original schedules is assumed to reflect a change in the schedules appearing on the website.
- The static and hard-coded nature of the data (in html format) on the website meant that we had to use certain integration code specifically required to extract certain parts of the web page.

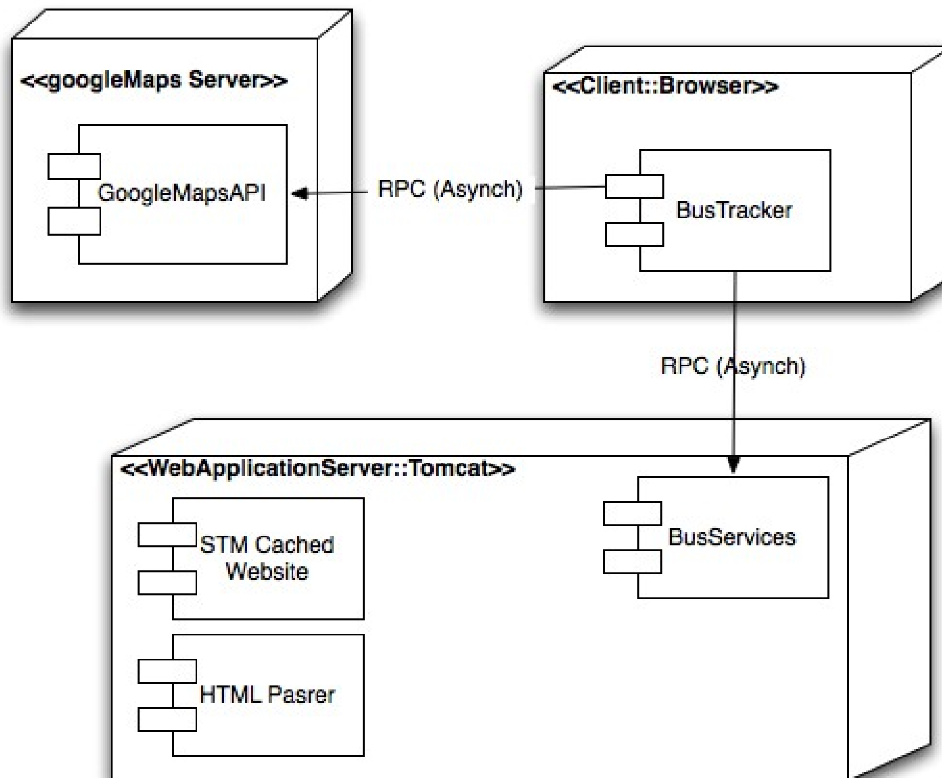
    - The web pages for different routes usually matched each other in their format, e.g. if the first stop intersection was the 2nd and 3rd column of the 3rd row in the 2 table in the HTML then it was almost always so for all the other bus route pages. This allowed for easier extraction of the required data from the different web pages.
    - The web pages for the subset of routes included in our project were cached and stored under the project package to ensure that the data format did not change overtime (i.e. the 2nd row in table 2 always referred to the first intersection for the East direction of the given route) and to enhance performance by reducing calls to the STM website.
- Validation was done by manually checking that a bus actually passed a specific stop at a current time according to the displayed schedule

4. Overview of the final architecture

The final architecture was based on the 3 components discussed above. The GoogleMapsComponent and the HTMLparser were used as a grey box through their available APIs.

The BusServices package essentially adapts the HTML Parser to form a parser specifically for the STM Website which is retrieved from a pre-cached folder. One bus route consists of approximately 100 html files. The adapted parser provides methods to parse these pages for the retrieval of Bus Stop intersections, and time-schedules. The adapted BusScheduldeParser provides a Bus Services API.

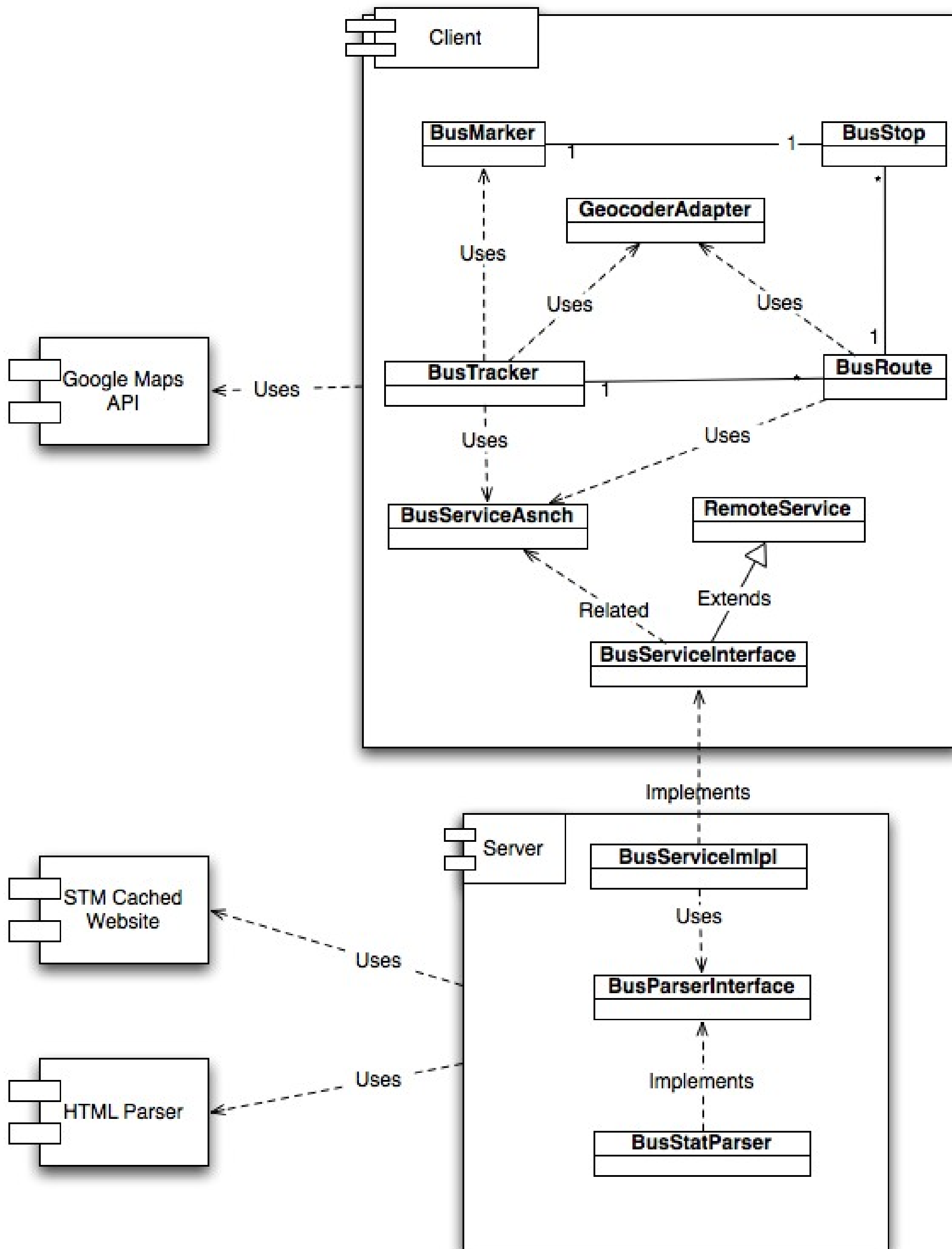
GoogleMapsAPI component is used to create a BusTracker that requests the Bus Services From the adapted Parser. The interaction between the 3 packages consists mainly of Asynchronous RPC(Remote Procedure Calls) which had to be re-synchronized in the BusTracker glue code. The following **Deployment View** of the architecture shows this interaction and demonstrates where each component lives :



(Deployment View)

The different classes that constitute the system can be shown in the following more detailed view of the

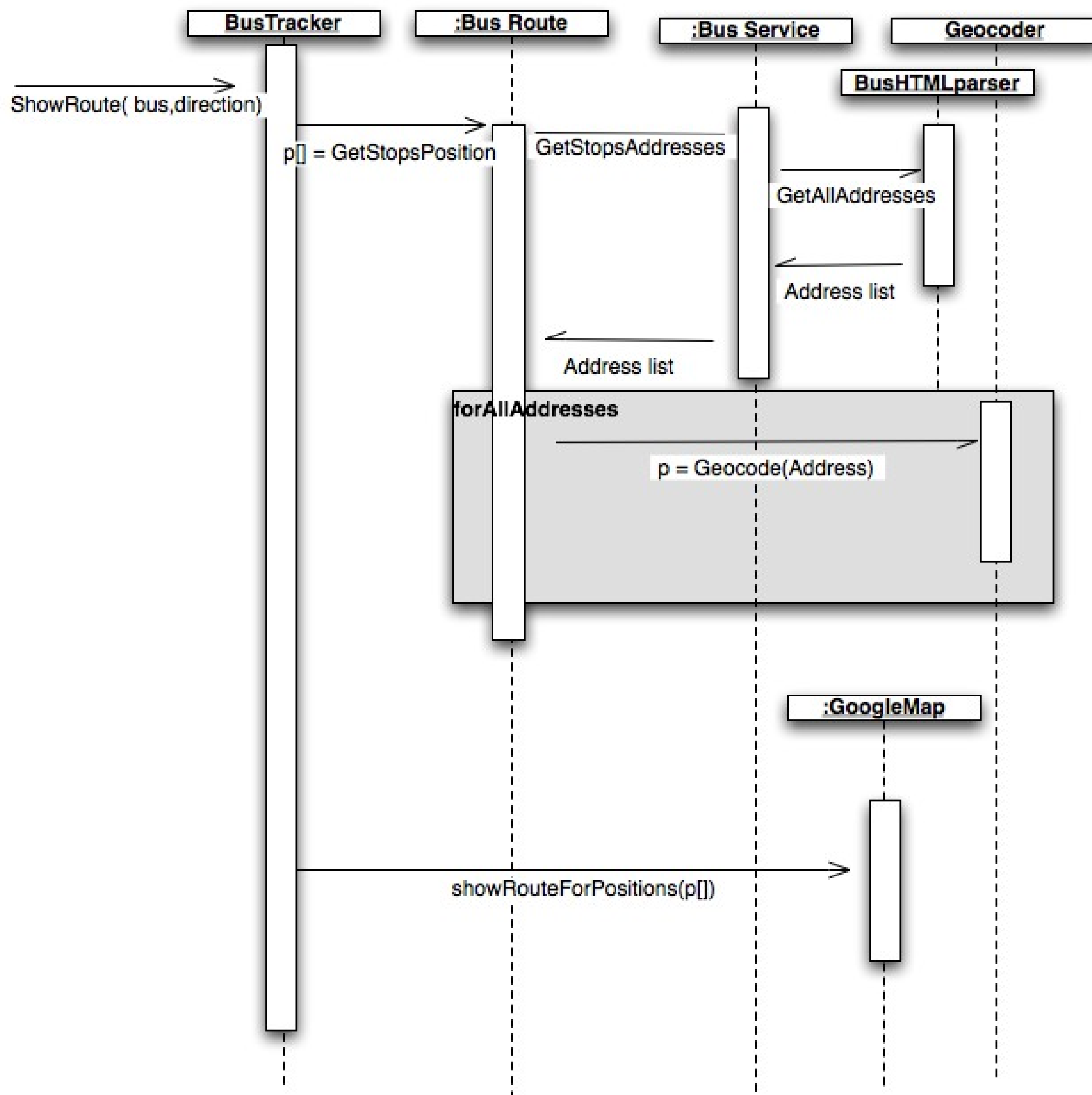
subsystems:



(subsystem view)

The following sequence diagram illustrates a common scenario for the execution of the show route service. it helps demonstrate the interactions sequence that is taking place between the different components.
BusTracker caches results of specific requests to be reused for better performance(like geocoding addresses), in this case that is covered by the diagram a new route is being created. I
It creates a new BusRoute object for the given bus number and direction and requests the BusRoute object for all the bus stop positions (given as an array of Latitude/Longitude geocode positions for a googleMap) on that route. To acquire this list, the BusRoute object then has to make a call into the BusService module, sending it the bus number and direction.
The BusService module uses its BusScheduleParser object to process the html files, and create a list of all bus stops in the form of street intersections,to be returned to the BusRoute object.The BusRoute

object then makes a number of asynchronous calls to the googleMaps component to acquire the geocode(Long/Lat map positioning) for each of the addresses in the busRoute. The main thread waits on the completion of the last of the calls using a timer to synchronize and then returns the assembled stop positions to the BusTracker object.
The BusTracker object is responsible for visualizing the positions for the user by creating visible Markers for these positions and displaying them on the Map using the googleMaps component. All this is characterized in the following **sequence diagram:**



(Sequence Diagram)

5. Architectural trade-offs in the BusTracker application:

During the design and implementation of the application, some trade-offs design decisions had to be made, here is a few characterized by their quality attributes:

Usability:
Averaging neighbouring geocodes when a bus stop address cant be geocoded.
Due to the low accuracy of the addresses provided by the bus schedule website, some of these addresses were non geocodable (e.g. mis-spelled street names), so the application made sure to accommodate for such scenarios by running an algorithm that will fix the route by calculating an average geocode rather than skipping that stop completely. This decision was made as a workaround of

the components mismatches that will be discussed in the following section.
Similarly if the schedule for a certain stop could not be accessed, then utilizing the information of time-differences between other stops on the schedule which on average came to ~2min, the stop would be given an estimated schedule by averaging that of the preceding and following stop, rather than failing with an error. This functionality is further enhanced with better accuracy by the 'subscription' mode of the bus movements as the bus marker moves from the previous stop to the next stop and passes the current stop with the missing schedule at just the right time to be standing at the next stop at its scheduled time.

Performance:
Setting a delay between firing the sequenced geocoding requests.
The chosen design of setting a delay between each subsequent geocoding request made to the server sacrificed some performance for a better chances of achieving higher rate of successful results. Given the fact that firing few subsequent requests at the geocoder usually means less successes. This reduced the performance at the cost of accuracy but still the route gets loaded in ~5-10 seconds.

Synchronizing the main thread with the asynchronous calls to the service and the GWT
Although using asynchronous method calls is intended to decrease the overhead of blocking the caller until the process of a synchronous request finishes, the fact that the BusTracker needed to, in most scenarios, to have the returned result of many asynch calls (Geocode all the route's stops), before finishing a certain functionality(displaying the Route), this added some unnecessary overhead to the synchronizing adapter in terms of blocking for an undetermined amount of time, checking every x milliseconds whether it got all the results back from all the Async calls.

Caching the website locally on the server.
Due to the extreme low preformance and bad response time of the STM website, a decision was taken to cache the website contents on the server. The size of the website that is relevant to the BusTracker application is relatively small and could be easily stored. This increased the performance dramatically, and didn't need extra modifications on the other component parts. The only drawback of this decision was that the schedule that is used might not remain up-to-date with the website, while on the other hand it ensured that if the website got redesigned, it would not have an adverse affect on our software. (This problem could have been avoided as we describe in the mismatches section, if the website used some sort of RSS feeds service). A possible solution to this issue could be to write a script that will run periodically (every 3 days) which copies the relevant pages and replace the old ones if it passed certain tests for ensuring matching design. This script is evaluated to be straightforward to manage but is beyond the scope of the current version of the project.

Testability:
The GWT framework that was chosen eases up testability through its support of JUnit and also provides the ability to use a debugger on Java code before it gets compiled into javascript as compared to programming it directly in javascript.
However, since many of the calls made to retrieve data are actually asynchronous, using the Java debugger did not help at all as pausing at a breakpoint would pile up a number of asynchronous call responses from the other threads which would barge in at the resumption of code execution thereby making a step by step evaluation of the program state during the code execution impossible.

Modifiability:
The individual components allow for easy modifications and enhancement due to the separation of modules, for example, if someone wishes to use the BusTracker module with a different city's Transit System schedule then they just need to develop the BusService module to provide a similar interface as our BusService which would provide a list of addresses for the selected route. The BusTracker would not need to be modified at all and would smoothly integrate with the new BusService module as long as they fulfill its 'requires' specification.

6. Integration issues :

(1) The google maps API as being part of the integrated Google AJAX API loader, which creates a common framework for loading and using multiple Google AJAX APIs, has conformed to providing the

map services under Asynchronous calls. This assumption about the pattern of interaction and the connector protocol proved to be useful in certain scenarios, in terms of better performance on the client side, by executing the call back function that is provided by the caller component (java script embedded in an HTML page in this case) when the google maps API component is ready to provide the service (like displaying a map for example). The performance gain of this scenario would be letting the code do other things meanwhile it is waiting for another service. This assumption of not providing synchronous calls to any service of the google maps API as a type of connector with other component proved to be problematic in using the Geocoding service of the Google Maps API. In BusTracker Application, when the user subscribes to a certain bus route, the client needs to geocode the whole list of addresses of the individual stops (upto 50 at times) that constitute the selected bus route, and since the Google Maps API provides a single address geocoding service at a time (in an Asynchronous fashion), a component that needs to geocode a bunch of addresses at the same time couldn't do that easily. The workaround that we were forced to use for this was to create a number of Timer with repetitions scheduled on the thread and then waiting on return conditions which are checked every few milliseconds. This assumption made the geocoding services hard to use in such a scenario, especially in cases like dealing with non geocodable addresses, and it forced us to have separate sanity checks for the collected data and re-arranging of the bus-stop positions in the correct order after all geocodes are received. The solution to this mismatch was achieved by building an adapter class that provided the required synchronous response to the geocoding request and dealt with all these issues using the Timers.

(2) Data Model - Mismatch between the html data format in the website and the required format for schedule calculations:
The bus schedule component of the system was provided in html webpages. The format of the designed website was purely for human interface where each street name was provided in a separate cell of a table in the webpage. For this reason, in some cases the street names were mis-spelled and other cases had names missing entirely in some cells where the other name provided an obvious location for people who know the region. This also explains the fact that in some cases (ex: bus number 144, or 28) when there is a stop which doesn't have an intersecting street specified, the schedule displays the number of the stop which was in the form of (NO ####). Some intersections are listed as 'Exit Only' which have no link to schedule for the stop. All these assumptions about who will be using the information made an automated use of the data quite difficult.
A similar problem was faced when trying to parse the bus schedules of a specific stop. The listed schedules are made with the intention of providing a human interface only where readability was the only requirement to be satisfied, which put little restriction on formatting standards (although even for that purpose it ought to have better readability than the existing one) on the internal structure or formatting of the schedules. The following link illustrates this poor readability for human And computer interfacing: 24-West.
This meant that we had to deal with all the special cases individually while parsing this kind of schedules. A clear example of this is when there is frequent bus-service, during rush hour for example, the stop schedule would be given as "..." for the minute and this notation could appear anywhere in the table, sometimes as the only content of the row and other times at the beginning, middle or the end of a row. This not only forced 'acrobatic' coding for all the workarounds but is also a cause for poor readability of the currently posted schedules.
Processing this kind of data even with the HTML parser component required more integration code than was expected. We were forced to make assumptions of our own, e.g. the "..." notation was expanded to place bus arrival times with 6 minute intervals continuing from the previous hour's posted time, etc.
In the end, the parsing component of the BusService package ended up ripping out all the text from the webpages and re-creating all the information in the required format, so that bus stops could match valid map addresses and schedules could tell exactly where a bus could be found at any given time.

7. Framework:

The GWT(Google Web Toolkit) framework has been chosen to develop the Bus Tracker application due to many reasons, which are discussed in this section.
GWT provides a set of tools for developers to build run-time javascript applications using the power of the static checking and simplicity of the Java language , and which was essential since Google Maps uses a javascript API. The only thing it was missing was support for google maps API from within the

java code. This problem was overcome by using a library that provides a translation for the google maps API in java, and back to javascript (gwt-google-maps-API) . GWT also provided support for eclipse which added another set of powerful tools into the developers hands. And most importantly it provided a skeleton for the programmer to follow in order to implement RPCs (Remote Procedure Call) in an easy and manageable way.
The framework choice has affected the overall architecture of the application by giving more flexibility and power to the developers, but at the same time forced a certain style to be followed, that is the RPC in this case.

8. Deployment Issues/Decisions

Due to the way the bus schedule website (stm.info) stored and displayed data, an HTML parser component was needed to parse the data into a more standard and usable form. This forced the use of an application server to run the parsing step, which could have been eliminated if the website provided a structured RSS containing the schedules and the stops addresses. If such file is present, it would have allowed to deploy the client only to run anywhere without having the need to callback to a server. which creates an extra overhead. And it could have possibly allowed for more deployment choices.
With the current format, once we run the server on one of the machines in the lab, we were able to access our application from any of the other CS lab machines but not from outside the firewall.

9. Future Work:

The application has great potential for developing into a more useful system and we plan specifically to achieve the following (in order of priority) over coming months

- Transform the client side to serve as API to simulate any transportation type movement(bus, train , metro...etc), any client who wants to use this proposed API will need to create a special RSS or a static XML file that is well structured and standardized, which will allow the application to avoid any parsing mismatches and be deployed as a client version only . we think that this has great potential and we will proceed with it .

- The BusService module can be further developed to provide times for bus-arrival at any selected stop.
- We can improve the BusTracker module to allow the user to be able to enter an address and a bus route and the application can pin-point the closest stop to the given address on that route and the time until the next bus arrives at that stop.

Glossary:

- Geocoding: The positioning coordinates in terms of Latitude and Longitude equivalent for googleMaps.
- STM: Montreal Transit Service.
- GWT: Google Web Toolkit. http://code.google.com/webtoolkit/
- AJAX: Asynchronous Javascript And XML
- RPC: Remote Procedure Call
- COTS: commercial off the shelf
- HTMLParser : http://htmlparser.sourceforge.net/
- GoogleMaps API: http://www.google.com/apis/maps/documentation/reference.html
- Bus Scheduling website: http://www.stm.info , this is the official website of the montreal public transport association