Jordan Maier
SlugTransit
A Bus Tracking System at UCSC
August 19, 2010

ABSTRACT

With such a wide, spread out campus, UCSC shuttle buses make it convenient to get around campus. However, there is always uncertainty as to when a bus will arrive at a specific stop. Nodes are in the process of being installed on campus buses as part of the development of the SCORPION networking testbed. These nodes transmit GPS and other data every three seconds and store the data in a MySQL database. For this project, I developed a program that would give real time estimates for bus arrival times at the bus stops around campus. The program looks at the current location of the bus, calculates the arrival time, and then displays it to the rider using a Google Maps interface.

INTRODUCTION

This report is aimed at anyone curious about how the bus tracking system was developed. It should be useful for anyone trying to understand how the code in the system works. And as such the report is fairly detailed.

The bus tracking system is based on the SCORPION network testbed. Computer nodes are being installed on campus buses as part of this testbed. Each node on the bus has a GPS device and a 900mhz radio. The radio in the nodes communicates to base stations throughout campus which relay the information to a central server. The server stores the information received into a MySQL database.

The UCSC buses are not on a very rigid schedule. They simply depart for a route at a given time, so there is much uncertainty about when a bus will arrive at a stop. The goal of this project is to provide bus riders with more certainty while waiting for the bus. With a program that provides users with real time arrival predictions at each stop would reduce most of this uncertainty. Hopefully this will increase ridership on campus and will motivate TAPS to keep nodes up and running in their buses. If there are more nodes in the SCORPION network then there will be more data available for network research.

When I took on this project, there was a simple bus tracking system in place. It simply showed the current location of the buses on a Google map, and only had a few icons for bus stops. This isn't ideal since it doesn't provide any solid data on when a rider can expect a bus. Before starting the design process I first had to tackle learning new languages. I learned about MySQL, PHP, and the Google Maps API.

RELATED WORKS

OneBusAway is a system utilized in Seattle, WA to provide residents with real time arrival info for the King County Metro. It has an online interface that users can also access from smart phones. It also has an interactive voice response system for users without smart phones that want to use the system while at a bus stop.

One project at Bridgewater State College approached the problem of predicting bus arrival times by dividing the route into zones. They created artificial sign-posts to see when a bus entered and left and zone and used this data to make calculations. Their predictions were simply based on the most recent time it took a bus to traverse a zone.

DESIGN PROCESS

The design was broken down into multiple parts. Figure 1 shows a general overview for the system. Each of the components will be explained in detail in the following sections.

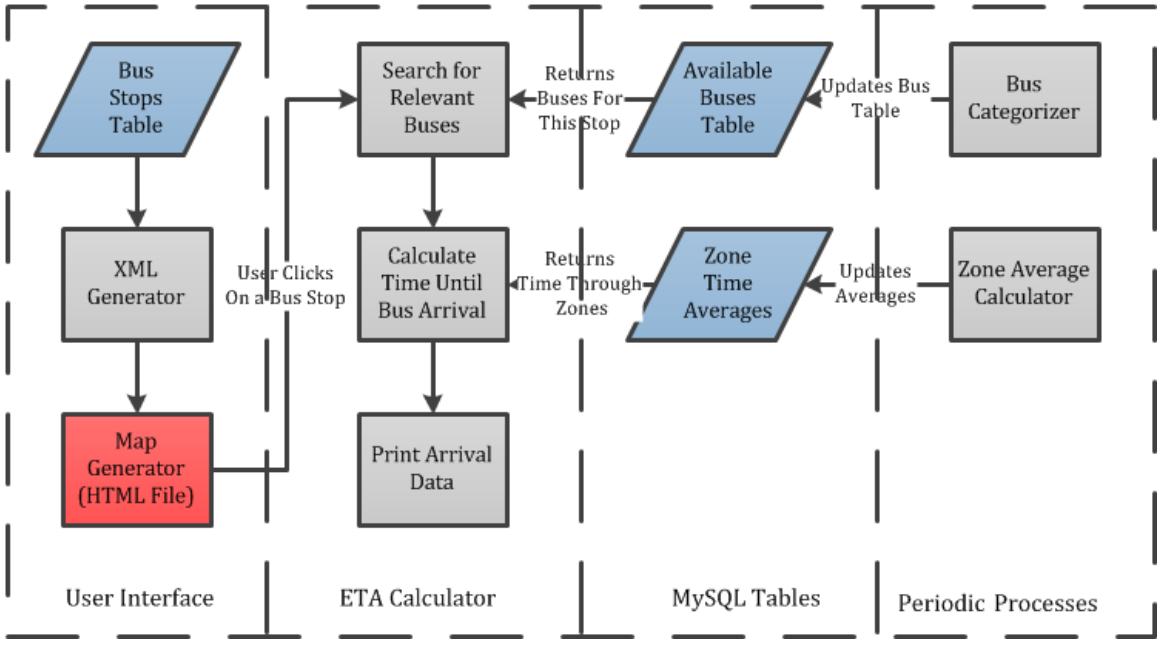

Figure 1: Design Overview

Map Interface

The first problem I addressed was to create a new map interface. The first step was to create a MySQL table containing information about the bus stops on campus. The table has 6 fields: id, name, lat, lng, dir, and card. “lat” and “lng” contain the GPS coordinates of the stop, “dir” contains the direction of buses that pass through the stop, and “card” is for the direction of the bus icon to point on the map. With the table in place, there is PHP file that generates an XML file. The XML file is need for creating a map. The PHP file queries the bus stop table and then prints info about each stop as markers in the XML file.

The next step in creating a map is the html file. It utilizes the Google Maps API to generate a map with the bus stops on it. The custom bus stop icons are declared in this file. The file loads a map from and using the properties in the file such as zoom level and center location. It loads the XML file and creates a marker on the map for each bus stop. It also declares an info window that contains data about the stop and a link to real time arrival info. The link passes through the id number of the bus stop through the url when a stop is clicked. See Figure 2 for a screenshot of the map interface. This will be used in the calculation of bus arrival times.

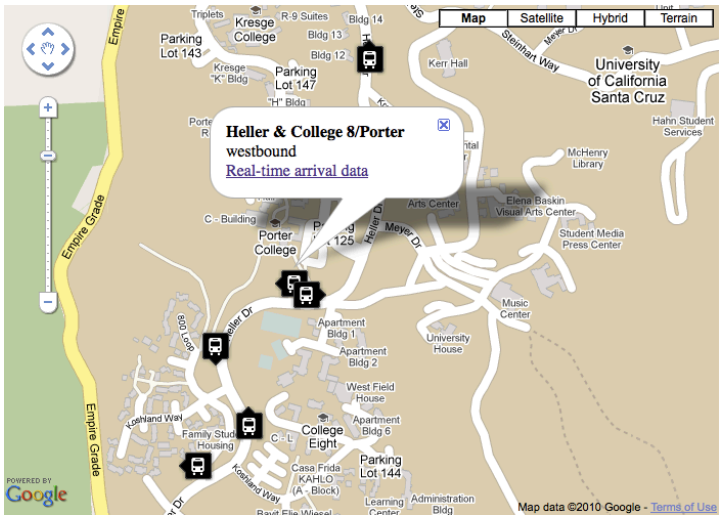

Figure 2: Map Interface

Zones

The next problem was to actually calculate the time for the bus arrival. The first thing I did was to divide the UCSC campus into zones. These zones enabled the problem to be broken down into smaller pieces. Time in each zone could be calculated independently from each other. With small zones, a straight line could approximate the bus's route inside the zone. Zones also provide a convenient way to analyze bus behavior as described later in the Bus Sorter section of the design. See Figure 3 for zone outlines along with a day's worth of GPS data.

There are currently two types of zones. The first type created were the zones for a loop route. The second type are core zones that fill in the space in the middle of campus to connect the loop zones and core zones that make up the core route. There is also a table with a different numbering for the core zones. These are usually referred to as "new" core zones. These were implemented in order to calculate estimated arrival times for core buses. These "new" core zones essentially consist of all the core zones and loop zones 7, 8, 9, 10, 11, 18, 19 and 20. There is tabled called "zonekey" that can be used to convert between zone types.

Information about the zones are currently saved in tables named "zones" and "core_zones." The table columns west, east, south and north establish the zone boundaries. The columns longitude_start, latitude_start indicate where the route

enters the zone. Longitude_end and latitude_end indicate where the route leaves the zone. Note: “leaving” and “entering” a zone is dependent on bus direction.

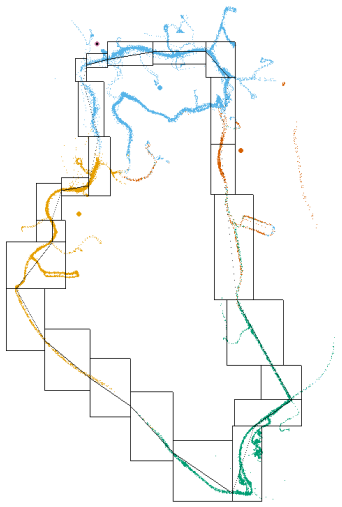

Figure 3: Zones with sample GPS data

Once zones have been established we can calculate how much time it takes a bus to cross a zone. This is accomplished with program that calculates average time through a zone. It is set up to accept one day’s GPS data and will go through it and find how much time it took to pass through each zone and calculate an average. This program can be easily modified to look at different times of the day or create specialized tables containing averages based on season.

The program goes through each zone selecting all GPS data within a zone. The program then looks for large gaps of time. For example a loop bus usually shows up in a zone every 20 minutes. The program then looks at the first and last GPS record in the zone during the time the bus was present. It also calculates which direction the bus was going while in the zone. It does this by comparing the entrance and exit points of the bus to the start and end points of the known route stored in the zone table. It saves each of the individual times into an array and then calculates the average time once it has all of the entries. Finally, the program updates a table to store the time information. The loop times are currently saved in “zonetestv1” and the new core zone times are stored in “corezonetimesv2.” These tables also include the distance across a zone. There are also columns that store the last time through a zone.

ETA Calculator

The program first saves the id that is passed through the url from the map into a variable. It then queries the database and prints information about the stop. It also saves information about the bus stop into an array. The program also sets core flags. These tell the program to not look for core buses if the bus stop is not along the core route. The program then calls the getBusCoords function and saves the returned data in a 2d array. The current date, direction of bus, and the stop zone are passed

into this function. The function first queries the "todaysBuses" table to see which buses are available and going in the same direction as the bus stop direction. It then gets the most recent GPS data from each of these buses. The function then attempts to find which zone the bus is currently in. For loop buses it starts in the zone with the bus stop and then looks at zones along the bus's path.

For core buses, the function just looks at the todaysBuses table for the bus's current zone. However, limits should be placed on this to only allow buses that are in zones previous to the stop zone. When the getBusCoords function finds an appropriate bus it saves the type of bus, current zone, current zone type, and x-y coordinates of the bus into a 2d array. Once the program runs through all the buses it returns the array.

After returning form the getBusCoords function, the program starts processing each bus that was returned. It first processes the zone with bus in it. It queries the time table to determine how long it should take to cross that zone and multiplies it by the percentage of the zone that the bus has left to traverse. Next, the program processes the zones between the bus and the bus stop. It selects the zones between the bus and stop from the timetable depending on which type of route the bus is on. See Figure 4 for a simple example of this process. If the bus is on a core route it first converts the zones to "new" core zones before creating the MySQL query. After the query, the program simply adds all these times to the time that was calculated from the zone with a bus in it.

The zone containing the bus stop is done in a similar manner to the zone containing the bus. It selects the time through this zone and multiplies it with the percentage of the total distance through the zone determined by the stop's position. This time is added on to the time calculated above. This is saved into the 2d array containing bus information along with the timestamp for the estimated arrival.

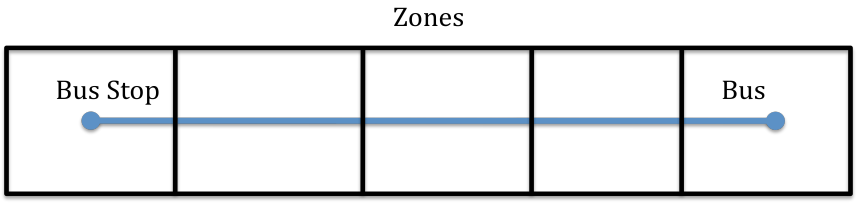

Figure 4: Simplified zone map

Once all the appropriate buses have been processed the program calls the displayInfo function to output the arrival data. This function creates a table containing route info and the estimated time of arrival. The next function, displayNearby, shows nearby bus stops. It simply queries the table containing bus stop information and looks for any other bus stops within a certain radius and provides a link to any stops it finds. It also outputs a link to return to the map.

The last section of code in the program is for providing information about scheduled buses. It can set a time interval for when buses are supposed to depart along with an

offset. An offset would be used if the buses ran every 20 minutes on the 10, 30 and 50.

Bus Sorter

In the previous sections many queries were made to the "todaysBuses" table. This table contains information about each bus running. It describes the type of route and the direction along with information about the zone containing the bus. It would be convenient if TAPS would just enter which bus was running each route. However, it will be more reliable for the program to determine which routes the buses are running. Currently the program can distinguish core and loop buses.

The program selects a sample of previous GPS data from each bus running, typically about 10 minutes worth. The program attempts to establish a pattern of zone traversal for each bus and then compares them to known routes. First it has to decide which zone the bus is in. The program passes the coordinates of the bus into a function called shortSearch. The function attempts to find the bus's current zone based on its previous zone. It first checks to see if the bus is still in the same zone. It then checks to see if the bus has moved up or down one zone. If these fail the function calls the searchZone function. The searchZone function goes through all the zones to attempt to establish a zone for the bus. If it fails it returns -1.

One common problem with GPS data is occasionally one point will occur out of a zone or in the incorrect zone. Figure 5 shows an example of this incident. In order to resolve this, the program makes sure that there are two entries from each zone before declaring that the bus has switched zones. To do this, it implements a 5-element array known as the checker. It stores 5 GPS entries in it and works its way down the list. Each time a new entry is added is performs a new check. The check starts of by comparing the 2nd and 3rd elements. If these are different and the first two elements are identical then it decides the zone was changed only if the 3rd and 4th elements are identical as well. If the 3rd and 4th elements are not equal, the checker sees if the 4th and 5th elements are equal. If these elements are equal and they do not match the 1st and 2nd entries then it declares that the zone was changed.

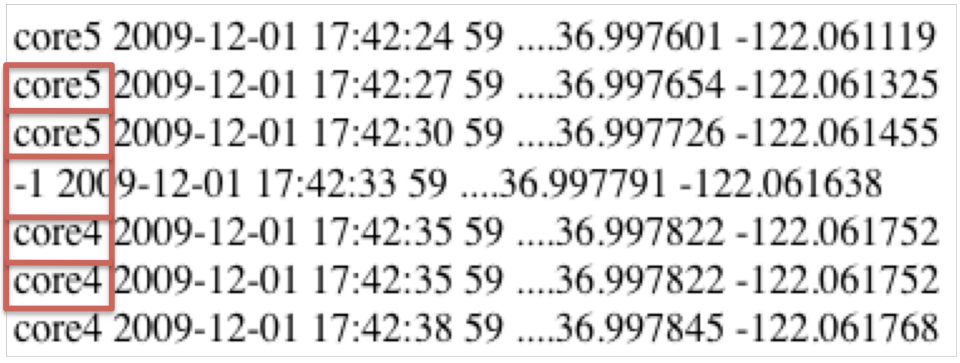

Figure 5: Sample GPS data with the checker

After returning from the performCheck function the program saves the previous zone if there was a zone change. It saves this in an array known as "behavior." This checking for zone changes continues until all GPS entries are exhausted. The program then calls the busRouteId function, passing through the behavior array.

The busRouteId function compares the behavior array to arrays that contain known route patterns. It looks at the first element of the behavior array and finds where that is in either the loop or core pattern. It then looks both directions in the known patterns to see if the next element in behavior matches one of these. The function also allows for exceptions in certain zones. Some zones don't have good radio coverage and might not acquire any GPS coordinates in a zone if the bus does not stop in that zone. The function returns an array that contains the type and direction of the route that the bus is running. If the behavior doesn't match any route it returns "unk" as the route type.

After returning from the busRouteId function, the program updates the todaysBuses table. It updates information about route type, route direction, current zone type, current zone number, and the time. Once all the buses have been sorted the program sets all the buses that weren't updated to "unk." It looks at the seconds category of the time updated and sets type to "unk" if it isn't within the last 4 seconds.

RESULTS

The program can currently distinguish core and loop buses on the UCSC campus and provide users with estimated arrival times. When the program was ran with previous data it was found that most of the estimates provided by the program were within 90 seconds of the actual arrival time of the bus.

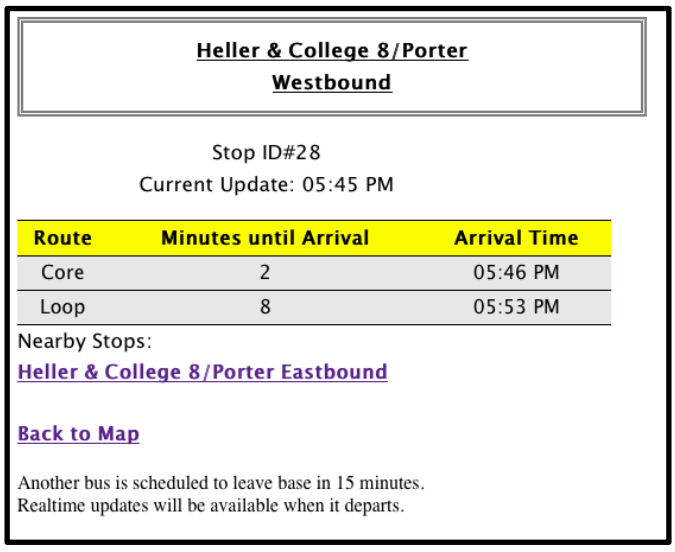

Figure 6: Actual Interface Output

DISCUSSION

The program is still in early development. There is still much room for improvement. Currently the interface works on web browsers only with limited usage on mobile devices. The interface displays on the iPhone but everything is small and it is difficult to click on a bus stop. Another future improvement would be to allow the program to use the GPS data from a person's iPhone, Andriod or other smart phone. It could use this data to automatically find the nearest bus stop to the user.

The timetable that stores the average time through zones could also use improvement. Currently it just contains the zone averages based on two random days of data. It could be expanded to include more data and distinguish between things such as time of day or season. The program could also be set up to use the most recent data collected. For example, when creating its predictions it could put more weight on the most recent pass through a zone than on the average time.

Only two bus routes currently work with the program. Other less popular routes should be added to the program.

Another helpful feature would be an interactive voice response (IVR) system. This would be useful for people without smart phones. These users could call the system and enter in a stop number and receive real-time data through this interface.

It would also be convenient for users if we could somehow incorporate data from the Santa Cruz METRO buses into the system. However this was far beyond the scope of the project for this summer.

CONCLUSION

The bus-tracking program works reasonably well with core and loop buses. It provides riders with a higher degree of certainty as to when the UCSC buses will arrive. They can check online for information and plan their transportation accordingly. This program will hopefully increase rider satisfaction and increase the number of riders. Ideally this will result in increased interest by TAPS to keep nodes up and running in the buses. This will provide more data to improve the bus tracking and provide important data to the Inter-Networking Research Group for network research.

At the current time the network needs some improvement before this system can be fully deployed. There are currently only one or two buses with working nodes installed. There also two base stations down, one at Baskin and the other at TAPS.

REFERENCES

Ferris, B., Watkins, K., and Borning, A. (2010) "OneBusAway: Results from Providing Real-Time Arrival Information for Public Transit." *Proceedings of CHI 2010.* Atlanta, GA, USA, April 10 - 15, 2010.

James Koshimoto, Matt Bromage, Vladislav Petkov, and Katia Obraczka. 2009. SlugTransit: a location-based public transportation management system. In *Proceedings of the 6th International Conference on Mobile Technology, Application \&\#38; Systems* (Mobility '09). ACM, New York, NY, USA, , Article 34 , 4 pages. DOI=10.1145/1710035.1710069 http://doi.acm.org/10.1145/1710035.1710069

Kidwell, Brendan. 2001. Predicting Transit Vehicle Arrival Times. Bridgewater State College. http://www.e-transit.org/eta/paper/predicting_transit_vehicle_arrivals.htm.

Wei-Hua Lin and Jian Zeng, "An Experimental Study on Real Time Bus Arrival Time Pre- diction with GPS Data", Virginia Polytechnic Institute and State University, 1999.

Jordan Maier's work was supported by the UCSC SURF-IT Research Experiences for Undergraduates Site, NSF grant CNS-0852099, surf-it.soe.ucsc.edu