OPTIMAL ROUTING ON ROADS USING DYNAMIC TRAFFIC BALANCING

Abhinav Mishra	amishra@andrew
Amber Palekar	apalekar@andrew
Rahul Iyer	rni@andrew
Vishakha Gupta	vsgupta@andrew

18-842 Distributed Systems
Advisor: Prof. Raj Rajkumar
Mentor: Brandon Salmon

Project Website: http://www.andrew.cmu.edu/user/amishra/cruisecontrol.htm

Table of Contents

Acknowledgements

At the very outset, we would like to thank Prof. Raj Rajkumar for his guidance on this project. His enthusiasm and entrepreneurial fervor resonated through the 18-842 class. 18-842 Spring ’05 has past, but this spirit will continue to live in our hearts all our life. We would like to express our heartfelt gratitude to him for firstly offering this course, but later providing us with the strong concepts that are the very foundation of Distributed Systems, and by that measure, of *CruiseControl.*

Thanks also goes out to Brandon Salmon for his keen interest and able guidance on *CruiseControl*. We learnt quite a few lessons on scalability just by listening to him. We would also like to show our appreciation to the other TAs Eno Thereska, Mike Mesnier and Obaidullah Khawaja for their help and guidance with the course work, which has impacted many phases of this project.

We would like to express our gratitude to Rahul Mangharam of the GrooveNet project for his endless patience and guidance. It is his advice that enabled us to add the map generation features to *CruiseControl*.

Thanks is also in order to Mathew Henson of the INI for his able systems support and help with the DNS issues.

Abstract

Today, there is a definite skew between the rate of growth of traffic in the cities and the rate of growth of commuting infrastructure. While the traffic grows at a much larger rate, the number of roads seems to grow at a much slower rate. This leads to congestion and wasted efficiency. An interesting observation, however, is that traffic tends to be concentrated on some roads, causing them to choke, while other roads go underutilized. This observation is the basis for CruiseControl.

CruiseControl is a software framework that attempts to "balance" the traffic across the roads in order to get commuters to their destination in the least possible time. In order to achieve this, CruiseControl first maps the area covered into a directed graph. Then, using sensors, it collects data from the various "nodes" about the traffic load at those nodes and uses this data to assign costs to the routes. It then computes the shortest path based on the graph constructed and conveys this shortest path to the user.

In the document that follows, we present CruiseControl as an infrastructure to achieve dynamic traffic load balancing.

Introduction

1. Projects Requirements

1.1 Purpose and scope of this section

This section specifies the requirements of the project through use cases which essentially depict the usage scenarios for the system. It highlights the high level requirements of the project.

This section gives a high level overview of the requirements. It does not delve into the intricacies of the system. That portion will be covered in the low-level design section.

1.2 Overview

Our product, CruiseControl, is a traffic load balancing system that gives its users the best possible path from a source to a destination based on different parameters like traffic density at the time instance, capacity of the road, distance etc.

1.3 Business Context

This product will be of utmost utility to the automobile industry. The ultimate goal is to incorporate the product in cars and other automobiles. We aim at balancing the traffic load and avoiding congestion to the maximum possible extent.

2. General Description

2.1 Product Functions

The functions that CruiseControl performs can be categorized as:

- Provide time-optimal path routing to end-user
- Provide routing information for the end-user on other parameters such as minimizing distance, etc.

2.2 Similar System Information

CruiseControl is intended to be a stand-alone system. However, given the functionality it offers, it is an ideal candidate to interface with an auto-driven vehicle, providing the vehicle's "driver" with information of what route to take.

2.3 User Characteristics

The typical users of the system are drivers of automobiles. They need not have any expertise with software systems. They need a basic understanding of computer operations (e.g. How to use a web browser)

2.4 User Problem Statement

Users at present face the problem of bearing with heavy traffic congestion. This is a result of insufficient information about the traffic conditions. There are radio services at present which give a brief idea about the traffic conditions but they are inadequate, to say the least. The information given is pertinent only to the major routes of the city. If a user wishes to know how to get from a given source to a destination facing minimum traffic, there is no system which provides him this information.

2.5 User Objectives

The user wants a system that will give a fairly accurate estimate of the traffic conditions of a given route at any given point in time. The user should be able to choose parameters while choosing a route. He will enter the source address and the destination address and will expect the system to give him the best route depending on the parameters chosen.

The user will always want a highly accurate estimate of the traffic conditions. However due to cost limitations and other issues there will be an approximation on the accuracy of the results. The exact approximation will be determined during the course of the low level design. The constraint here is that the approximation should be within acceptable limits, such that the primary function of the product is served.

2.6 General Constraints

The load identification sensors for the traffic information will have to be simulated in the software. In reality, if/when CruiseControl is deployed as a production system, the load identification will be done by sensors in the automobiles and/or at the traffic signals. The accuracy of the traffic information that a user gets cannot be 100% in theory and practice due to the latencies involved with the network as well as computational nodes in the system.

System Design

The requirements laid out in the previous section place stringent demands on the system as regards computation and high availability. We have tried to design our system with minimum possible coupling although maintaining cohesion in the system objects. The design, as is, supports fault tolerance, carries out load balancing at the compute servers and the web servers. The design is elaborated in sections that follow.

4. Architecture

The architecture description can be split into two parts. First part deals with the high level design of the system in terms of the functioning layers and their interactions. The next part goes into finer details and lays out the components that make up the system.

4.1 High-Level Design

The system can be viewed as a layered architecture with well-defined interactions between the different layers. Following figure shows the various layers which are described below.

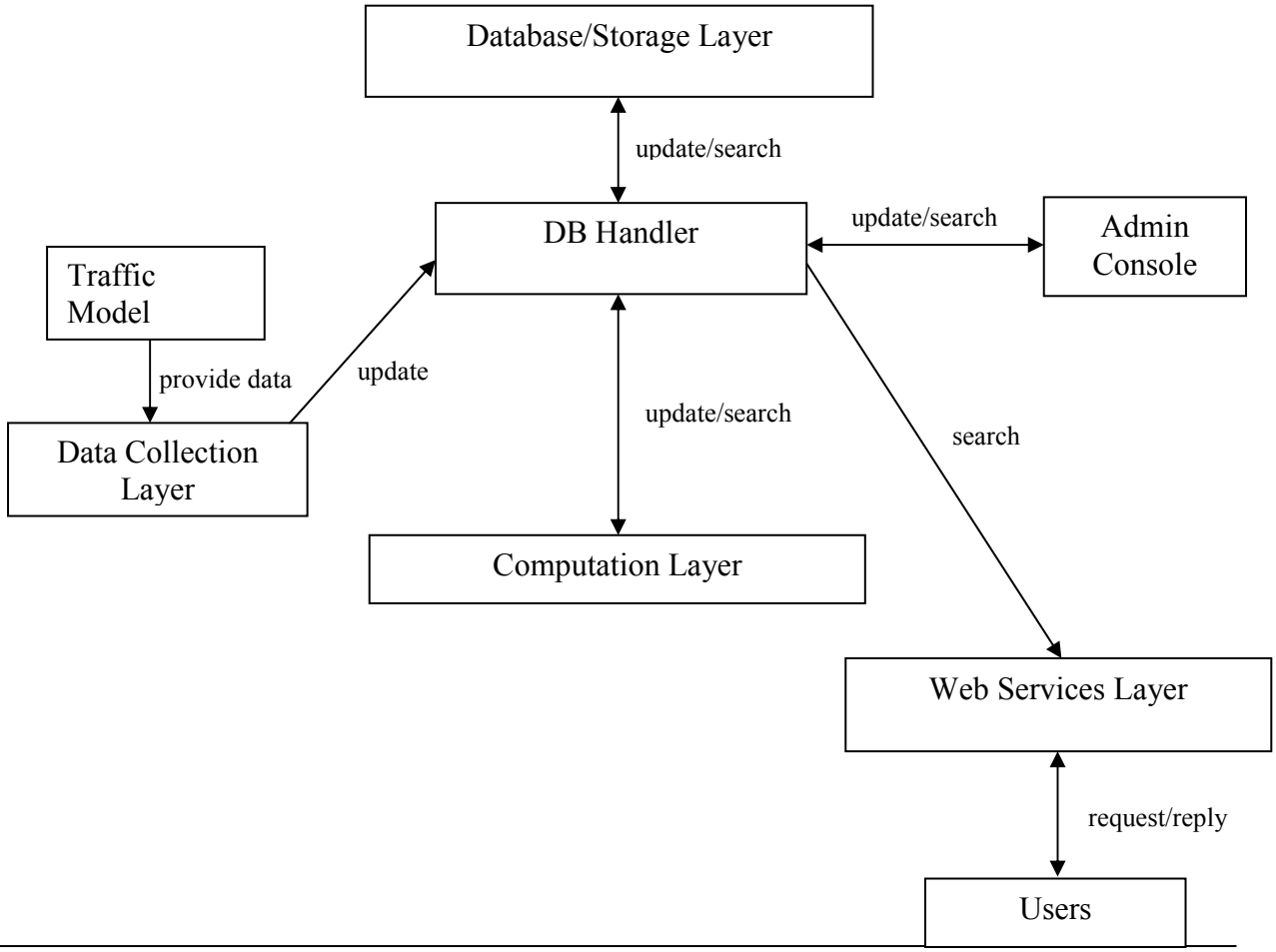


Fig. 4.1 BASIC FUNCTIONAL LAYERS AND INTERACTION BETWEEN THEM

1. Database/Storage Layer and DB Handler – This layer stores all the relevant data for example, the traffic conditions acquired from the data collection layer, the map details in order to respond the to web service's request for route information and system related data input by the administrator. It interacts with the other layers through the DB Handler in order to decouple the operations. The data can also be accessed and updated by the system administrator which is responsible for adding new routes and other such details.
2. Data Collection Layer – As mentioned in the previous sections, our system requires the current traffic conditions which include information regarding the number of vehicles on streets, if there is a traffic jam condition or accident or some blockage etc. This information is collected by this layer and updated in the database. The traffic model is a plug and play kind of an implementation which generates traffic at regular intervals according to a specified pattern
3. Computation Layer – This layer is responsible for picking up the traffic data from the database and computing the fastest/shortest route for the user. The computations are performed on demand depending on requests that it receives through the web services' layer. The graph is read once and updated for link load information every certain time interval. Route for a given source and destination pair is calculated as requested by the web services' layer
4. Web Services' Layer – This is the layer which serves as a front end to the system with which the users can interact. Users here can be cars equipped with route depiction systems, cell phones or normal desktops which access the service through their browsers. On finding the requested route, a map is also displayed on the front end which graphically depicts the route to be followed.

These are the layers which summarize the design of the system. The text on the arrows shows the operation that is performed during the communication between different layers and the arrow heads indicate the permissible directions of communication. These layers are further explained and explored at a finer level in the following diagram and the explanation following it.

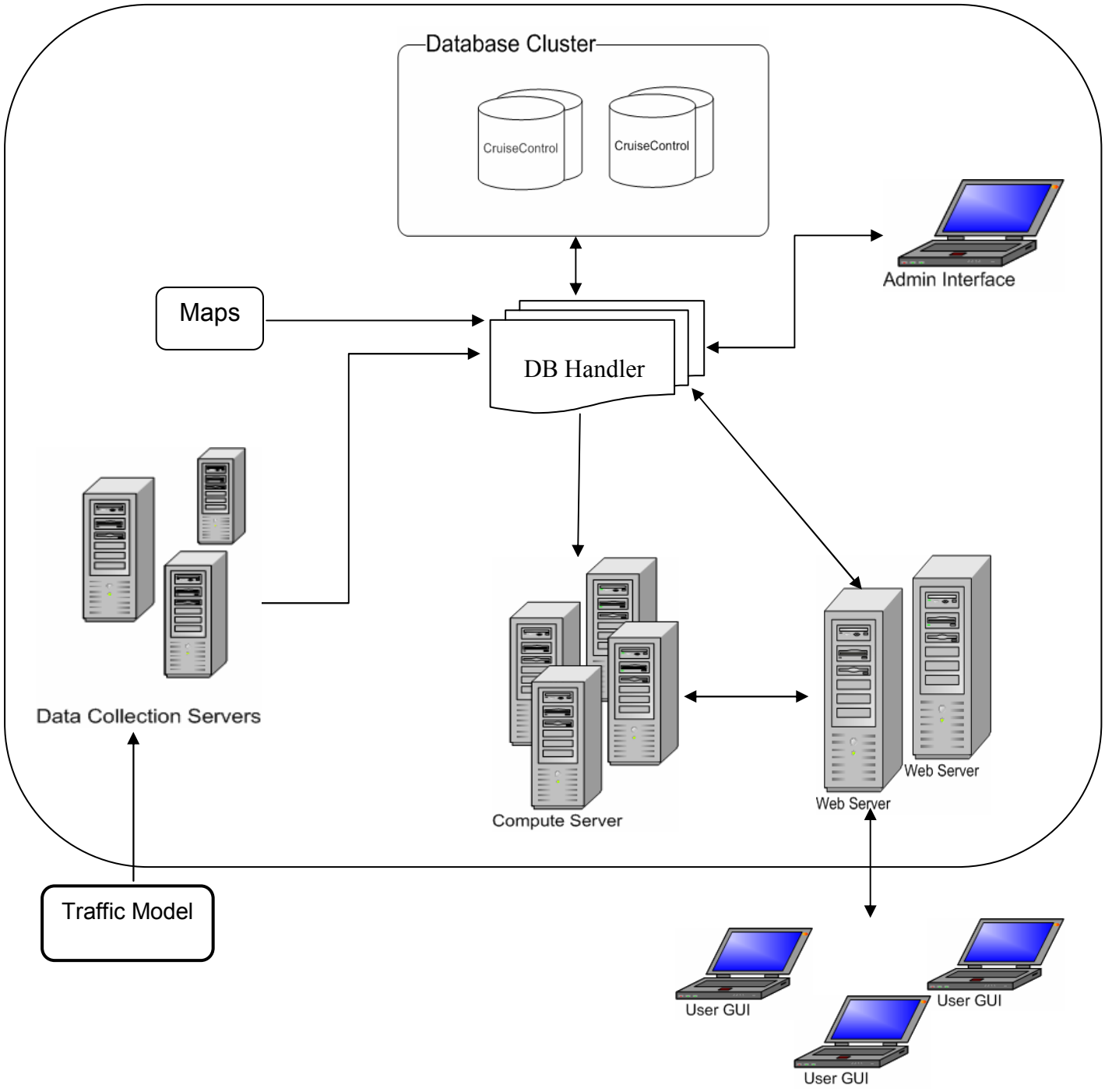


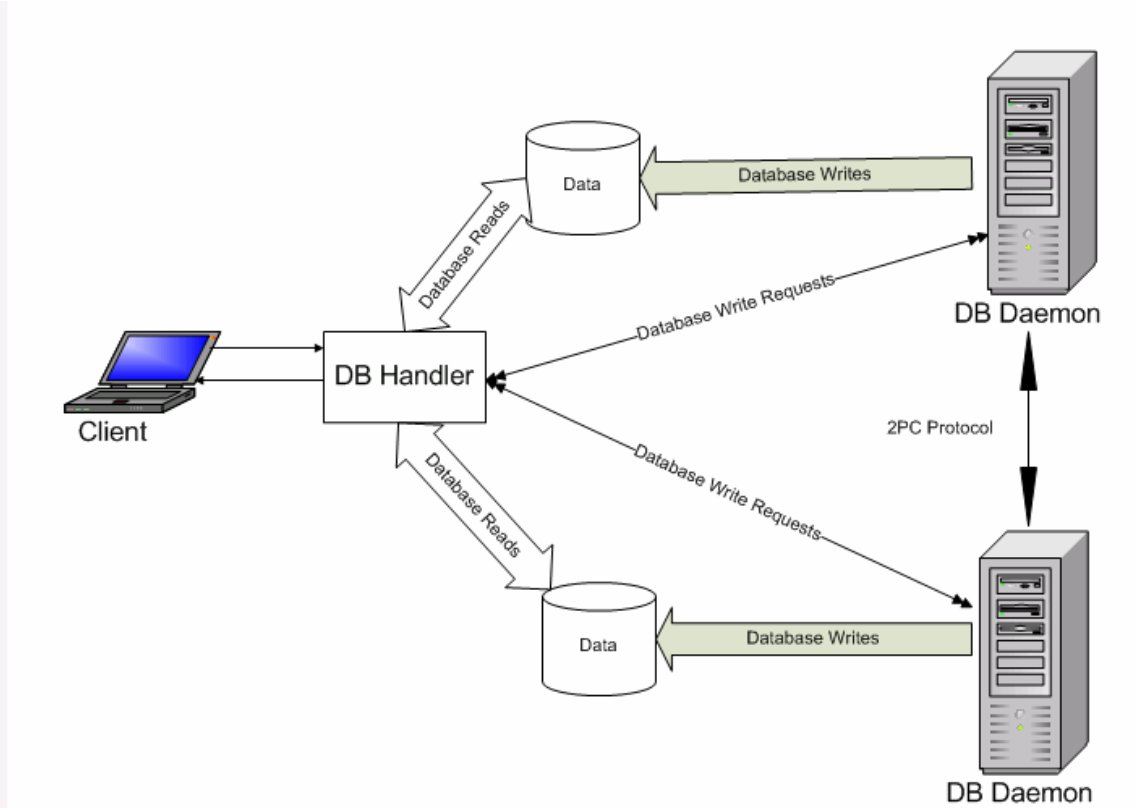
Fig. 4.2 COMPONENT/LAYER DESIGN

1. Data Storage and Access – The system consists of one CruiseControl database which is replicated on two database servers. The databases store information about intersections such as their names, latitude and longitude information, about links (streets in real life) such as link load, length, capacity and average speed limit. These parameters are used in the calculation of weights when finding the route using Dijkstra's algorithm at the

computation servers. The two replicas are maintained in sync using 2 phase commit protocol implemented by the DB Daemons. All the update/insert requests to the databases have to go through the daemons so that they remain actively synchronized at all times. Whenever a database server fails and comes, it checks the latest transaction id and synchronizes itself using the log of changes that the alive database maintains. The read requests to the databases from clients like compute servers, web servers and traffic data collection servers are directed to the databases directly through the DB Handler.

2. DB Handler – This component provides the abstraction for interaction with the database for the rest of the layers. It provides APIs for the other layers to connect to the database query the database, insert into the database etc keeping the 2 phase commit transparent to the upper layers.

The following figure shows the interaction at the database layer:



3. Traffic Model – This component is responsible for supplying current traffic data to the system. Due to resource limitations, we cannot use actual sensors or a GPS to provide real time traffic information to the system, due to which there is a need for traffic model. The traffic model currently assumes three kind of links namely, heavily loaded, moderately loaded and lightly loaded. The links are classified as heavy, moderate and light at different hours of the day and this data is picked up by the traffic generator.

Depending on the classification of the link, the load is generated at random to fall in heavy, medium and light loaded links. This data is then sent to the data collection servers which finally update the databases so that the compute servers can pick up the latest load information in order to find least congested routes.

4. Data Collection Servers – Multiple instances of the data collection server are responsible for receiving information about the traffic, modifying the format of received information according to database requirements and storing the information into the traffic DB. These servers are highly available due to a multi-threaded implementation of functionality and they are fault tolerant which means failure of one data collection server doesn't stall the load update process.
5. Compute Servers – This is the primary component of the system where the route finding algorithm is executed. These servers run the Dijkstra's shortest path algorithm using a priority queue based implementation in order to improve the efficiency. The algorithm terminates the moment a shortest route to the destination is found to further reduce the compute time. The algorithm runs on a graph which is read once to get all the details like vertices (intersections) and the neighbors along with edge information (link information) like length, capacity, speed limit and load. Thereafter, load in the links is updated every fixed time interval and its frequency is similar to the frequency at which the traffic model updates link load information into the table.

   The compute server implements an interface which it uses to register itself with the .NET remoting registry for access by the web server backend. The interface is visible to the client that wishes to query the compute servers (which is the web server in our case). The interface exports two methods, one for querying the shortest route (distance as weight on edges) and the other for querying fastest or least congested route for which the weight function is given below:

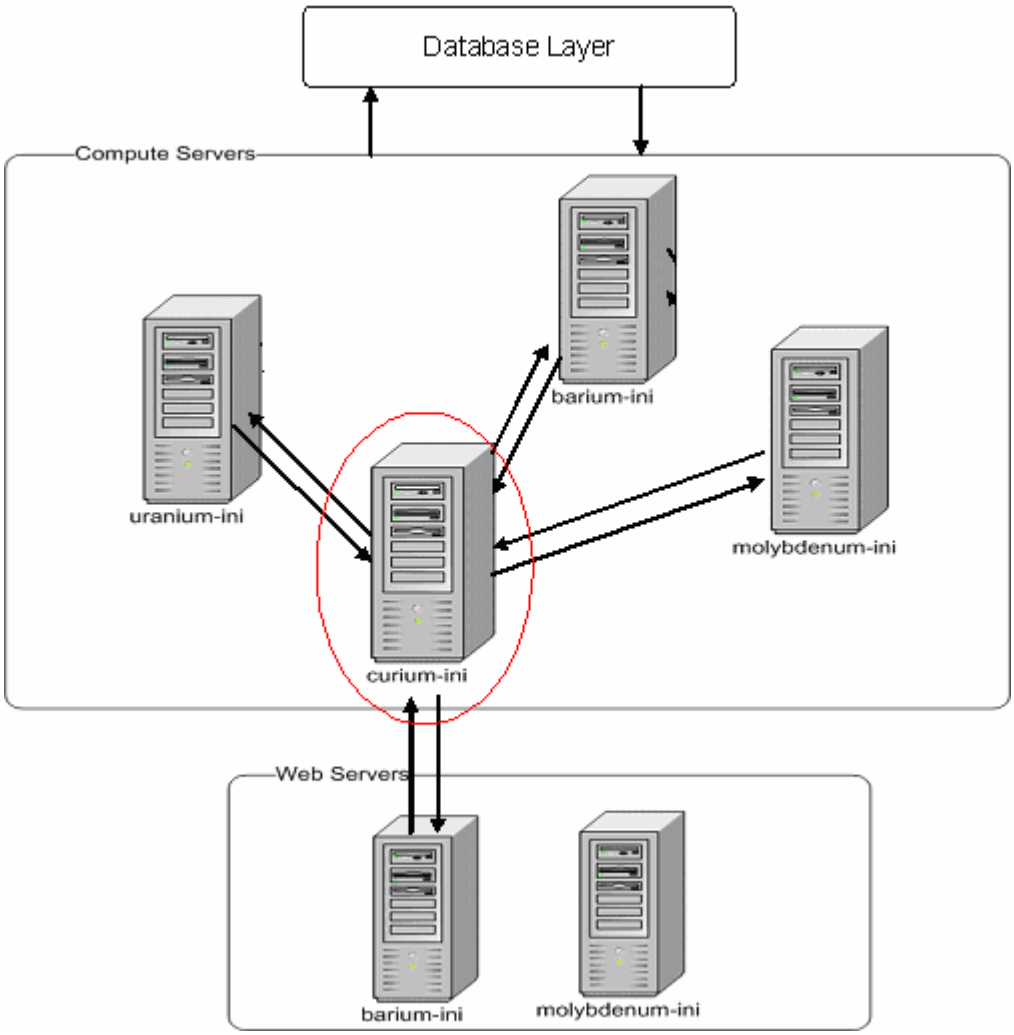
```
weight = (link load) * (length of edge / speed limit) / (capacity of road)
```

One very important requirement at the compute servers is their scalability. The number of intersections and links in a graph is huge and there could be millions of client requests at each time. If all the web servers query just one compute server, the latency in calculation is bound to increase and the performance of the compute server could degrade. Therefore, we have another component at the compute servers which is a software load balancer. It works in a distributed manner and whenever a server gets

overloaded, it instructs the web server to change over to a less loaded compute server (server-initiated load balancing).

The load balancer works as follows: every compute server has an instance of the load balancer running. It is completely multi-threaded and every server maintains load information about every server. There is a thread which monitors the local machine load using the Performance Counters provided by .NET framework (it uses the % Processor Time, Pages/sec exchanged in memory, Total Bytes/sec sent and received at the network interface to calculate a weighted average which we denote as machine load). The detail about Performance Counters to be chosen and the weight to be attached to these for calculating the weighted average is found out from a config file (an XML file which has tags for the counters to be used, the load balancer port etc.). This load is sent at intervals to the Master. This master is nothing but a peer compute server which was least loaded in the previous run of the load balancer. The master collects load information from all the peers and sends the overall load information to the other compute servers so that every node has the latest load information about all its peers. If any of the peers other than master goes down, the master accordingly updates the total machine load information using which the peers know which compute server is alive or dead. If the master goes down, the other peers who are waiting to receive communication from the master timeout and send their load information to the next least loaded server which becomes the master, finds out if the old master is dead and accordingly updates the total machine load information.

Now the load balancer and the route finder need to communicate in order to inform the web server the moment the compute node gets loaded beyond a threshold (this threshold is a system parameter set by the administrator), it asks the querying web server to start communicating with another compute server. At startup, web servers pick up the existing compute servers from the database and choose anyone at random for querying about route information. The load eventually gets balanced as the load information starts getting exchanged. The following figure shows one run of the entire route finding process including load balancing. The arrows between the Compute Servers show load information being sent to a Master for the time under consideration and how web server gets connected to the least loaded server (which is the master). The other web server is yet to start querying the compute nodes.



6. Web Servers – For clients to access route information between the desired source and destination pairs, web servers provide the pages or GUI as required. This component also needs to be highly available so that clients do not have to face delays. Also the interaction with the database should be highly optimized so that the route retrieval is fast. The Advertisement Server provides advertisements to be displayed along with the route display.

   Our system uses TIGER to generate a map to display the recommended route. This takes the form of an HTTP request. The URL formed holds the longitude and latitude at which we want the map to be centered, along with the width & height of the map, the size that we want the map to be (e.g.: 1024x768) and the type of image that we desire (e.g.: GIF/JPEG). To mark intermediate intersections on the map, TIGER requires that we setup a file in a specific format containing data (longitude, latitude, symbol to use to mark & label) about points on the map that we want to mark. This file must be accessible to TIGER over HTTP and therefore must be made available on a web-server.

7. Maps - CruiseControl displays the final route between the source and the destination in the text as well a graphical format. The graphical format is essentially a map between the source and the destination with the fastest/shortest route as chosen by the user.

   Currently, we are using the Tiger Map Server maintained by the US government to generate maps for CruiseControl. We send queries to the Tiger Map Server in the form of an URL the format of which is discussed in detail later. We are compatible with the Tiger Line Format. This implies that our databases store street information which has a one-to-one mapping with the Tiger Line database which is again a census information maintained by the US Government which contains information about street names in all counties in the US.

Figure: Sample map displayed by the Tiger Map Server

Why Tiger Map Server?

After an extensive search for free tools to generate maps from the Tiger Line data, we chose to use the Tiger Map Server. The primary reason behind this decision is lack of a comprehensive "free" mapping tool based on Tiger Line data. We did find an open

source mapping server but the resolution offered by the current version of the tool was unacceptable for the purpose of our project.

GrooveNet

The mapping data present in CruiseControl's database, including the information about street intersections, zipcodes and adjacency of intersections, has been extracted from GrooveNet – a similar project at CMU. GrooveNet uses Tiger Line database to generate maps. Using simple perl scripts, we have extracted all information needed by CruiseControl from GrooveNet. A future enhancement for CruiseControl would be integration of our fastest path algorithm with GrooveNet.

URL Format

The URL contains the latitude and longitude information of the centre point of the map to be displayed. To display a map between a source and destination, we estimate the longitude and latitude of the centre of the map and send it across to Tiger. The URL includes the parameters like the size of the map and latitude/longitude information about markers to be displayed. To display a route in a user conceivable way, we send across the latitude/longitude information for each road intersection on the route to the Tiger Map Server to display them as markers on the map.

8. Administrator Console – The administrator of the system needs to be provided an interface for carrying out updates to the database in cases when some new street has been constructed or some static weights used in the system change etc.

5. Application Programming Interfaces

The system provides certain standard APIs which are enlisted and described below. This list is not exhaustive as more APIs will be added as we work further on the system.

5.1 Database Layer

```
/* Connects to the DB
 * Parameters:
 *    server - the name of the machine the DB is housed on
 *    db - the name of the database
 *    user - username of the db user
 *    password - password for the user
 *    dbtype - the database used. Right now only Mysql is
 *    supported. Pass value as Mysql
 * Returns: true on success, false on failure
            */
public bool connect(string server, string db, string user, string password,
string dbtype)
```

```
/* Disconnects from the database
 * Parameters: void
 * Returns: none
 */
public void disconnect()
```

```
/* Inserts the row represented by rowset[] into the table specified
 * Parameters:
 *    tableName - name of the table
 *  row - row to be inserted
 * Returns: true on success, flase otherwise
 */
public bool insertIntoTable (string tableName, Rowset[] row)
```

```
/* Updates a table unconditionally
 * Parameters:
 * tableName: name of the table
 * row: columns to be updated
 * Returns true on success, false otherwise
 */
public bool updateTable (string tableName, Rowset[] row)
```

```
/* Updates table based on condition specified */
public bool updateTable (string tableName, Rowset[] row, string condition)
```

```
/* Deletes from table as per condition specified
 * Parameters:
 * tableName: name of table
 * condition: specified condition
 * Returns: true on success, false otherwise
 */
public bool deleteFromTable(string tableName, string condition)
```

```
/* Gets the rowcount; unconditional
 */
public int getRowCount(string tableName)
```

```
/* Get the row count
 * Condition: condition specified
 */
public int getRowCount(string tableName, string Condition)
```

```
/* get rows specified based on condition */
public bool getFromTable(string tableName, Rowset[] cols, string condition,
ArrayList result)
```

```
/* Get rows specified - unconditional */
public bool getFromTable(string tableName, Rowset[] cols, ArrayList result)
```

```
/* Get all rows from table based on condition */
public bool getFromTable(string tableName, string condition, ArrayList
result)
```

```
/* The workhorse for getting rows
 * Parameters:
 * tableName: name of the table
 * cols: the columns needed; null for *
 * condition: the condition to match rows with
 * orderCol: the column to 'order by'; set to null for no preference
 * groupCol: the column to 'group by'; set to null for no preference
 * result: result set
 *
 * Returns: true on success, false otherwise
 */
public bool getFromTable(string tableName, Rowset[] cols, string condition,
string orderCol, string groupCol, ArrayList result)
```

```
/* Fire a direct SQL query to the back end. Use is discouraged!
 * Parameters:
 * query: the sql query
 *
 * Returns: true on success, false otherwise
 */
public bool directQuery(string query)
```

```
/* Get the latest error string
 * Params: none
 * Returns: last error string
 */
public string getError()
```

```
/* Gets the latest error number
 * Params: none
 * Returns: last error number
 */
public int getErrorNum()
```

5.2 Admin Console

```
/* This method authenticates the administrator to grant him access to the
database using the parameters passed*/
authenticate(string login, string passwd, string db, string host)
```

```
/* This method inserts node information into the database using the data
passed by the GUI */
insertNode(string name)
```

```
/* This method updates existing node information into the database using the
data passed by the GUI */
updateNode(int node_id, string new_name)
```

```
/* This method deletes existing node information into the database */
deleteNode(int node_id)
```

```
/* This method queries the database for existing node information
 * The search could be based on node_id or node_name. If node_id, pass
 * node_name as empty string. If node_name, then node_id passed as -1.
 * Result is returned in the array node and function returns true on success
*/
queryNode(int node_id, string node_name)
```

```
/* This method inserts new link information into the database using the data
passed by the GUI */
insertLink(Link new_link)
```

```
/* This method updates existing link information into the database using the
data passed by the GUI */
updateLink(Link new_link)
```

```
/* This method deletes existing link information from the database */
deleteLink(int link_id)
```

```
/* This method queries the database for existing link information */
queryLink(Link lnk)
```

```
/* This method inserts new system parameter information into the database
using the data passed by the GUI */
addParam(SystemParam s)
```

```
/* This method deletes an existing system parameter information from the
database */
deleteParam(SystemParam s)
```

```
/* This method modifies existing system parameter information from the
database using the data passed by the GUI */
modifyParam(SystemParam s)
```

```
/* This method queries the DB to retrieve information about all system
parameters */
getParams()
```

5.3 Traffic Model

```
/* Generate load for current time period */
getCurrentLoad()
```

```
/* Send the update to the Data Collectors over the wire after converting the array containing the load into a parseable format. */
public void sendUpdate(Load[])
```

5.4 Data Collection Server

```
/* Query the system_params table and setup the parameters (such as number of links in the system) required by the traffic collector */
goQuerySysParams()
```

```
/* Start listening for updates */
startServer()
```

```
/* Process the update received. Verify that the update is valid and initiate update to DB */
processData()
```

```
/* Parse the update received to identify current loads on each link. */
parseTraffic()
```

```
/* Update the load value for the linked passed. */
updateDB(link_ID, load, dbHandler)
```

5.5 Web Server

```
/* Get an array containing the details of all intersections (nodes) in the system.*/
getNodeTable()
```

```
/* Get an array containing the details of all intersections (nodes) in the array of nodes passed that lie in the specified zipcode */
getByZip(nodeTable[], zip)
```

```
/* Get the shortest path from specified source intersection to specified destination. Remotely invokes the getShortestPath() function at the Computation Servers. */
getOptimalDist(src, dest)
```

```
/* Get the fastest route from specified source intersection to specified destination. Remotely invokes the getFastestPath() function at the Computation Servers. */
getOptimalTime(src, dest)
```

```
/* Gets the unique murlID the webserver should use while forming the URL to query the TIGER database for the map. */
getMurl()
```

5.6 Compute Server

```
/* This method implements the getFastestRoute functionality and calls dijkstra
to calculate the least congested path to reach from the given source to the
given destination */
getFastestRoute(int source, int destin)
```

```
/* This method implements the get Shortest Route functionality and calls
dijkstra to calculate the least length path to reach from the given source to
the given destination */
getShortestRoute(int source, int destin)
```

5.7 Load Balancer

```
      /* This method queries the Machine Load structure and returns the
hostname of the least loaded machine at the query time */
      getLeastLoadedMachine()
```

```
      /* This method returns the local machine load so that the node can
check if its load has increased beyond a threshold and if it needs to give
away its load */
      getMyLoad()
```

6. Use Case Scenarios

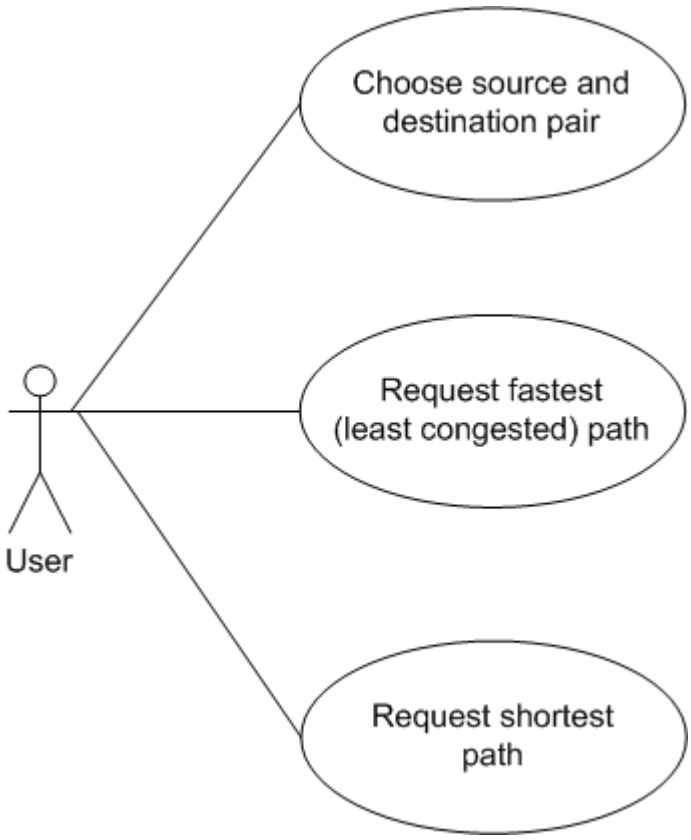


Fig.3.1 USE CASE 1: INTERACTION OF AN END-USER WITH THE SYSTEM

- **Actors:** The end-users interact with the system as shown above.
- **Pre-conditions:** An accurate model of the entire network of roads along with current traffic conditions should exist in the system.
- **Basic Flow:** When the end user requests routing information, he sends the source & destination locations along with the parameter that he wishes to minimize (time or distance). The optimal route is computed based on whether the user has chosen to get the shortest route or fastest route. The system then advises the user to use this optimal path.
- **Alternative flows:** None.
- **Use case relationships:** None.

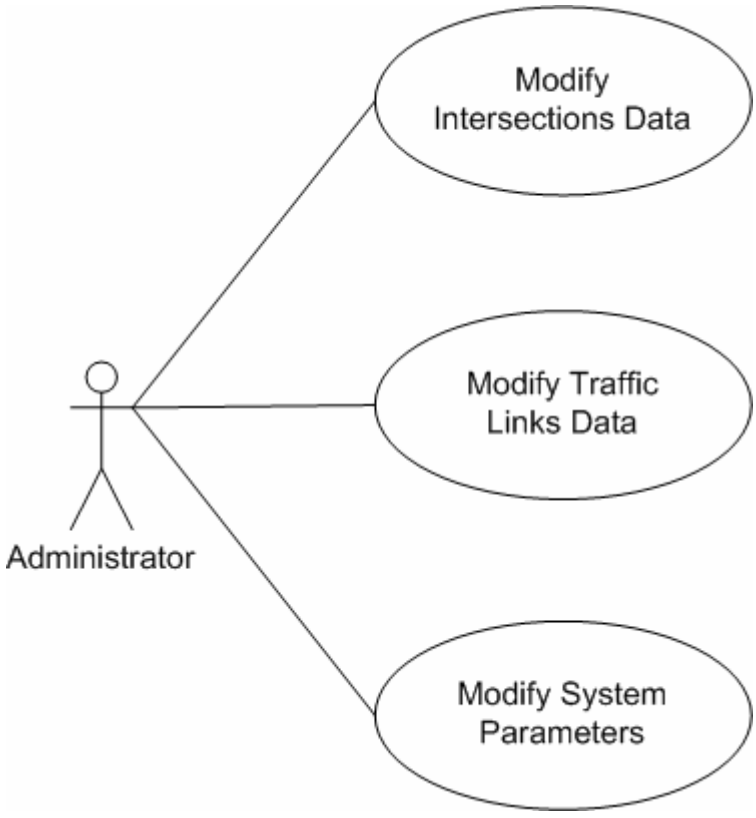


Fig. 3.2 USE CASE 2: INTERACTION OF ADMINISTRATOR WITH THE SYSTEM.

- **Actors:** The system administrator interacts with the system as shown above.
- **Pre-conditions:** If the administrator needs to interact with the system, the current model of the road network in the system must be requiring a change or the system parameters

like refresh interval for graph etc. need to be updated. Such a case may arise when new roads are added in the system or existing roads are to be removed or the administrator wants a change in the overall system behavior.

- **Basic Flow:** To edit or remove a route, node or a system parameter, the administrator sends an update to the database that holds the list of routes, nodes and system parameters and their associated weights.
- **Alternative flows**: None.
- **Use case relationships**: None.

Implementation

7. Platform

The implementation of *CruiseControl* is done using the .NET platform. Choosing this platform had a multitude of reasons. Firstly, though a more "powerful" language such as C would have been a good choice from a performance perspective, a lot of the requirements, such as RMI, database access etc. would have been an attempt at reinventing the wheel. Hence we had to go with a middleware.

At this point, we were confronted with two ubiquitous choices – Java and .Net. We chose .Net because of two reasons. Firstly, all of us had used Java in some form or the other in past projects. We felt this was a welcome chance to take a look at a new middleware platform and maybe, in the process, get a comparison with Java. Another reason for picking .Net is that unlike Java, it produces native binaries – hence we expected better performance from .Net. After all, the facilities of a middleware are important, but performance is not a dispensable objective. We used C# for most of our code and ASP.Net for our web pages.

We use MySQL as our database. We chose MySQL for a number of reasons. Firstly, it is a light database. It comes with less "baggage" than Oracle and delivers acceptable performance. Also, MySQL cluster provides a distributed database facility, hence taking care of a lot of implementation detail, without having to reinvent the wheel. We used ODBC.Net as the glue between our application code and MySQL.

Visual Source Safe was used for source control.

In the end, with a few exceptions with regard to MySQL, we are happy with our choices and would recommend them to anyone who has a inclination towards using them.

8. Experiences

To sum it up, our experiences with .Net were pleasant. Our choice of C# as the primary programming language was n excellent one. C# provides the all the flexibility of Java (as opposed to the less rich C++), while delivering performance comparable to C++. We have made extensive use of features such as Remoting (.Net lingo for RMI), ArrayLists (vectors),

SortedLists (HashMaps). The Visual Studio IDE too is an excellent IDE and besides providing all the "standard" IDE features, its coupling with MSDN provided immense help. For us beginners with .Net, MSDN was an invaluable resource, and we have no qualms in suggesting it as a guideline for anyone developing documentation for a product. As regards performance, The Common Language Runtime, or CLR, is the price you pay. It is a heavyweight set of libraries and can be a real memory hog at times. If one has the memory, then .Net delivers acceptable performance. It may not blaze like C or C++, but then again, neither of those have RMI.

We'll kick off our grouse list with MySQL. After a lot of experimentation, we figured that MySQL clustering is not supported on Windows. We agree that this is a port, but then again, this wasn't really mentioned really clearly in the documentation. This forced us to implement a layer for distributed commit (which, to be fair, we enjoyed doing). So, that's the first cross for MySQL. Another reason where MySQL could improve is integrity checking. MySQL does not do any integrity checking. Insertion of a value in an incorrect format causes MySQL to insert a set of zeros rather than throwing an error. While we did not face this first hand, this could be a cause of a lot of nasty bugs and database inconsistencies.

Finally, Windows, ODBC.Net and IIS all seem to suffer from a common problem – number of connections. Windows seems to restrict this to 25 or so. ODBC too has a similar limit. Ditto with IIS. Problems such as these hampered our scalability testing and it would have been a lot better if such an artificial limit were not imposed.

Evaluation

While our system is quite complex, there are only a few ways in which the user can interact with the system. The basic use case scenario, as illustrated in the previous figures, is that the user requests for either the optimal path, either on the least distance or least time metric. The other scenario is the administrative one – one in which the System Administrator changes the existing maps.

As of now, both these sequences work. We have tested both scenarios and seem to have both in working order. However, there does exist one issue - when the administrator changes the map, the compute servers do not know about this change until they reread the database on a timeout. Hence, there is a window of time in which the compute servers will operate on stale data.

Internally, there are a few components vital to making *CruiseControl* a distributed, highly available, fault tolerant service.

- **The Database Handler:** The database Handler is the interface to talk to the database. At this stage, it has full support for the 2 phase commit protocol for writes to the database. Reads, however, are direct. This component is fully functional and is currently servicing the data needs of *CruiseControl*.
- **Database Daemons:** The database daemons are the components that are responsible for the distributed atomic commit protocol. This component is fully functional and is currently performing satisfactorily under the load of the Traffic Collectors. The current scheme is 2 database daemons working in collaboration.
- **Admin GUI Backend:** This class is the "glue" between the admin GUI and the Database handler. Currently it is fully implemented and deployed.
- **Web Server Backend:** This class connects the web server to the compute servers. It makes a remote call to the compute server class. This is fully functional.
- **Compute Servers:** This component is responsible for the route computations. It is the heart of the *CruiseControl* logic and is currently running an implementation of Dijkstra's algorithm. We have tested two compute servers in parallel and have found no issues.

- **Load Balancer:** The function of this component is to balance the loads between the compute servers. This has been implemented and currently uses CPU time, amount of paging and I/O time as metrics to make its decision.
- **Data Collection server:** The Data Collection server is responsible for collecting traffic data from the Traffic model. It has been implemented and uses TCP to get the data from the traffic model and then passes it to the database via the Database Handler.
- **Traffic Model:** This is a framework to simulate real-time traffic. It is "plug and play" and currently supports a pseudorandom traffic generation scheme.
- **Advertisement Server:** The 'Ad server' generates random advertisements for display on the *CruiseControl* webpage.

9. Performance

As part of our implementation, we carried out a few performance tests on our system. Unfortunately, due to time and other constraints we could not be as rigorous as we would have liked. Given our constraints, we decided to perform some basic performance tests that targeted critical sections of our system, rather than going for an approach driven by profiling.

- **The Database:** As part of our database testing, we targeted two critical properties – the scalability of the database and the number of operations per second. We tested these aspects by running two programs, namely dbsmash and dbsmash2.
    - *Scalability:* The former creates a large number of threads, each of which fires queries continually at the database. This worked satisfactorily. While many queries were lost (due to Windows' maximum connection limit), the database daemons did not crash. In our opinion, this is a plus as the missed queries were primarily an artifact of the underlying operating system and not our software.
    - *Operations Per Second:* The latter, dbsmash2, fired a large number of queries from a single thread. We were satisfied with the results of dbsmash2. All queries were processed and we achieved about 25 operations per second. MySQL without the distributed commit layer (the database daemons), gave us 40 operations per second. The select performance on the other hand was the same as the read data path does not go via the database handlers. We could read a table with approximately 115,000 tuples in about 6 seconds. Optimizing this is non trivial as this is the base data read rate that the database gives us.

- **The Compute Servers:** The compute servers, like the database are hotspots in the system. Poor performance at the part of the compute servers would negatively impact the response time seen by the user. As part of our testing of our compute servers, we saw varying response times, depending upon the distance to be traversed on the graph. However, we never did see this time cross 1 second. We deem that this is satisfactory.

10. Lessons learned

This project was, for all of us, the first large distributed systems project we had worked on. It was an invaluable learning experience on many fronts. At the end of it, we all have only pleasant experiences of it, so we guess that's a good thing.

Arguably one of the biggest gains from the project was the importance of planning. We initially focused on planning quite a bit, due to which out design phase, and subsequently, the implementation phase, was delayed. We focused our planning on two major planes. Firstly, we mapped the components on to owners. The focus of this phase was to map components to owners in a way that each person got a component he/she was most interested in. This proved to be a very wise decision as the overall enthusiasm towards the project was maximized, and the quality of the work produced reflects this. Secondly, we mapped the various components to timelines that we thought would be acceptable. However, this proved non trivial. Each one of us had varying schedules and workloads, and so arriving at the ultimate match was quite an effort.

Another aspect we focused on was assigning secondary owners for every component. The need for this was twofold. We strongly believe that 2 heads are better than one and that while drawing up the implementation level data structures for any particular component, we had two people work on it so that we could get it to be as robust as possible. Also, this fosters an eXtreme Programming style pair programming technique, which again produces higher quality code. In a nutshell, we learned a lot of Project Planning 'on the job'.

Another important lesson was that of scale. Very soon we realized that building a system of this magnitude was quite different from writing a lab assignment for a course. Data structures that have acceptable memory size and performance for smaller data sets can scale very poorly and soon become memory hogs and give abysmal performance. A case in point is an adjacency matrix. This data structure scales as N^2 and soon grows too large to be part of the Resident Size

of the application, leading to incessant paging and performance levels that are unspeakable. Changing this into a multilevel priority queue seemed to scale much better.

Threads are generally deemed good. We agree… almost. While threads are an excellent way to exploit the parallelism in tasks, they are not a panacea. Arbitrarily increasing the number of threads increases the concurrency of the system, but not necessarily the overall performance. The Database daemons were massively multithreaded in an attempt to support as many parallel transactions as possible. Unfortunately, at one point in time, these parallel transactions started interfering with each other, leading to a drop in performance.

Finally, there is no such thing as free beer (seriously!). Powerful middleware comes with a grave performance hit – the cause of our worry for quite a while. All in all, it was an excellent look at systems design and the tradeoffs in making them go fast.

Project Status

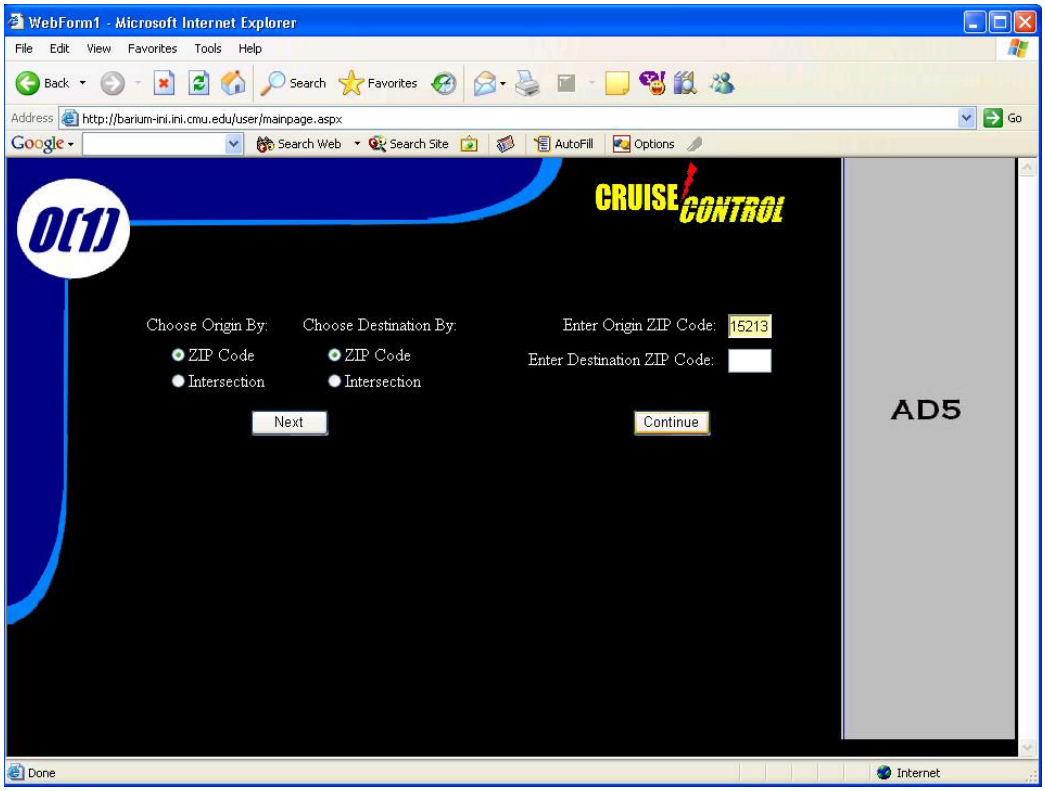
CruiseControl is currently in version 1.0 Beta. The functionality has fully been tested and *CruiseControl* seems to be performing its basic function. *CruiseControl* has in its database all the intersections and streets in Allegheny county and can route between any two intersections the user may wish to mention.

The user and Administrative GUIs have been implemented. The user GUI can display the requested route both in text form and on a map. The web servers reside on barium-ini and molybdenum-ini. All the requests from the users have to go through the web servers. The load is balanced between the two using DNS.

The Compute servers run on two machines namely uranium-ini and curium-ini. The load of incoming requests is load balanced between these two machines by a distributed load balancing component. This makes the compute servers highly available and 1 fault tolerant.

The database is replicated on the aforesaid machines. It follows hot replication and uses DNS to load balance between the two machines. There database daemons that implement the distributed commit are also in place. Hence the database is a replicated, highly available, distributed facility. We tolerate 1 fault tolerance in the database.

The Data collection server too is up and running. It runs on both barium-ini and uranium-ini and uses primary backup. The traffic generation program is implemented too and runs on barium-ini.
The interfaces for user GUI and admin GUI are as shown below


WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://barium-ini.ini.cmu.edu/user/mainpage.aspx
Go
Google Search Web Search Site AutoFill Options
O(1)
CRUISE CONTROL
Choose Origin By:
ZIP Code
Intersection
Choose Destination By:
ZIP Code
Intersection
Next
Enter Origin ZIP Code: 15213
Enter Destination ZIP Code:
Continue
AD5
Done
Internet


Administrator Login
Username
o1
Password
Database
CruiseControl
Host
DBServer
Login

Future Work

While *CruiseControl* is in usable state now, we there is quite some scope for improvement. Some of the changes mentioned below should have been in this release but weren't because of constraints in time or otherwise. We have followed the motto "It is better to ship a sub optimal solution that works 100% of the time rather than an optimal one that works 50% of the time". Hence some of these features were not included.

We currently have a web based user interface. One of the first things on our agenda is to port this GUI to one that can be displayed on a handheld as well.

The compute servers currently run an implementation of Dijkstra's algorithm. While this gives us acceptable performance at present, moving to a larger scale, such as a whole state may make the running time of this algorithm larger than would be tolerable. We were looking at an algorithm A*, which has a quicker run time than Dijkstra's algorithm. The price you pay, however, is that A* is a heuristic and could produce a suboptimal result. Our initial investigations into this algorithm show that this margin of error is fairly small and the sub optimality introduced will be acceptable.

Secondly, we would like to improve the traffic model. There are many models that have been proposed as an approximation to traffic flow in real life. These vary in complexity from "non-trivial" to "fiendishly complex". While we could have incorporated one of these into the current version, this would have come at the cost of some of the features of *CruiseControl,* such as its availability or fault tolerance. We hope to replace the current model with a more realistic one in the future.

For map generation, we use the US Government's Tiger database. While this is an excellent resource and is serving us well currently, its performance at times leaves a lot to be desired. In this regard we have looked at two options. The first option is to switch our map provider. We have been considering using Google maps and are looking into it at present. The other option is to get all the map data to our local database and use our own map generation routine for this purpose. The future course of this feature is unclear at present, largely due to questions about the feasibility of generating the map information locally.

Currently, the database runs a two phase commit protocol. In environments where there can be a large number of failures, the vulnerability period of this protocol can be too high to be acceptable. We are considering switching this to a 3 phase commit protocol.

Also, we hope to extend the Admin GUI to have features to monitor the health of the various components. Another enhancement is to add asynchronous notification to the compute servers whenever a change is made to the database.

Finally, we hope for the opportunity to try to port *CruiseControl* to other middleware so that we can leverage the advantages of other high performance Operating Systems and cluster based computing.