Cloud Framework for IoT associated GPS location based applications

J. Lakshmi, S. K. Nandy, Aakriti Gupta, Pavan Kumar Akulakrishna, Vaibhav Ankush Kachore

***Abstract*—Real-time events associated with human activities is catching rapid pace due to feature and function rich mobiles like the smartphones. These devices not just facilitate users to seek access to information derived from large data sources, but also enable them to add data as sources to equally large amount of data repository. One such interesting and widely used application is a location based navigation service. This application derives its utility based on large amount of geo-spatial data repository overlaid with current events happening in the neighborhood of user interest. However, because of amount of data that needs to be handled for processing, most of the times these applications process either static data or use trends based on historical information. What really increases the utility of these locations based services is modulation to real-time changing events. This paper identifies three key characteristics that need to be addressed to fulfill this gap, namely timeliness in responding to user queries based on changing events, data repository to support such functionality while preserving the desired user privacy. Here we present the three facets to a navigator application based on the above three characteristics and demonstrate how its utility can be improved using compute clouds.**

***Index Terms*—Internet of Things (IoT), Compute cloud, location based services, Real-time scheduling, dynamic priority, soft deadline, distributed storage strategy, nine-cell-grid data distribution, load balance, user privacy and data obfuscation.**

I. Introduction

Internet of Things (IoT) is the upcoming trend in usage of internet to connect physical objects worldwide. Extension of cyber physical capabilities to mobiles through smart phones is one such extensively used example. The data dimension associated in this active space is large since most mobile applications thrive on real-time event information. One popular example is the Google navigator application. The Google-maps data associated with this application is estimated to be around 21 Petabytes [1]. While an individual instance on a user phone may use a limited subset of this data, the interest is about live events occurring in and around this data. So storing and using updates on this information adds an extra dimension to this data.

GPS based location services on smartphones are handy tools for many travelers and act as easy navigational aids. Here, they are the clients for many utility based applications that the user has come to depend on. These smartphones can also be likened with sensor devices when they collect and provide data that are used by location based service providers. What is specific about these devices is that they are constrained devices and can do limited computation or store limited amount of data. What is interesting about these applications is that they need the essence of observation over large data and maybe large period of times. This is where the compute clouds offer the versatile platforms required to host the huge data generated by these devices and enable the build of interesting applications over this data as demanded by varying user needs [35].

GPS based applications have proven to be effective in providing useful navigational and utility associated information to the traveler. Applications that aid building of smart cities like disaster warning for earthquake prone areas, waste management and disposal monitoring by municipal corporations under clean cities initiatives, EMF level warnings for health hazards near nuclear or wireless equipment and towers, etc. are now becoming ubiquitous. However, they tend to become limited if these applications cannot modulate to changing events and associated data change relating to the associated events. The three key properties such applications need are:

1. Location based services need to provide timeliness in responding to user queries.
2. GPS applications often have to handle large amount of data pertaining to location. Data distribution that is suitable to meet real-timeliness to these services is desirable.
3. User privacy is important in these applications since most clients also act as sensors to feed data into the system. It is desirable that application specific information is passed on without disclosing other data related to the participating user.

This paper details case studies associated with these three aspects in the following sections. The explorations of these ideas are carried out using road navigation application as the example. While the examples chosen here relate to GPS based applications, it is easily perceivable that for many IoT scenarios the three properties highlighted here indeed a reality.

A. *Real-time routing for GPS based navigation applications*

Routing in road networks is an old problem with renewed interest. The current focus is trending on how to extend real-timeliness to this problem. Navigational queries are normally in the form of seeking information on shortest path between two points in the geographical space. While many algorithms exist to speed up the traditional Dijkstra's algorithm for computing shortest paths in static road networks [2][3][4], only few have shown applicability in the dynamic case adapting to traffic updates on the road network. Most algorithms either use history based modelling of road traffic to compute the travelling time, neglecting the real time updates or distributed algorithms are used which let the client decide which way to go based on its local view of the graph and updates relevant to it. There are limitations to both of these approaches. Firstly, most of these applications depend on pre-defined databases that have not been designed to manage real-time updates associated with various events in locality. Secondly, the route computation requires global viewpoint which involves larger data with associated updates. This mandates involvement of higher and mostly on-demand computational capacity and this makes a case for a compute cloud backend to support the real-time requirements for such applications [33].

B. Efficient Storage of Big-Data for GPS based navigation applications

Navigator and maps kind of applications can potentially use large amount of data for a variety of applications that are associated with route information. The nature and format of the data collected depends on the sensing method used, of which GPS is currently the most popular. Most of these applications today are limited to using static data with current location. However, location based GPS applications need real-time responsiveness and are location-sensitive. GPS data is time-variant, dynamic and large. Current methods of centralized or distributed storage with static data impose constraints on addressing the real-time requirement of such applications. Here we evolve a methodology of storage mechanism for the GPS application's data that can capture and respond to real-time changes. In our approach, data is partitioned into cells giving priority to Geo-spatial location. The geography of an area like a district, state, country or for that matter the whole world is divided into data cells. The size of the data cells is decided based on the previously observed location specific queries on the area. The cell size is so selected that a majority of the queries are addressed within the cell itself. This enables computation to happen closer to data location and thus eliminate data communication overheads. We also build some data redundancy, which is used not only to enable failover mechanisms but also to target performance. This is done by nine-cell approach wherein each cell stores data of eight of its neighbours along with its own data. Cells that have an overload of queries can easily off-load some of their workload to their near neighbours and ensure timeliness in response. Further, effective load balancing of data ensures better utilization of resources. Experimental results show that our approach improves query response times, yields better throughput and reduces average query waiting time apart from enabling real-time updates on data [34].

C. Data Obfuscation to ensure privacy for users in GPS based applications

Incorporating real-time responsiveness to user queries and choosing data storage mechanisms that enable such processing are fundamental to most IoT based applications. The third facet to this class of applications is the amount of personal information that can be made visible while enabling specific functionality to the user query. This is particularly true to smartphone like appliances since these devices not just extend features to applications but they also are sensors to gathering real-time information. Crowd-sourcing in today's context is very common and many users provide extremely useful and relevant information [18][19]. However, while event associated information is what the user really wants to disseminate, unfortunately, personal information associated with the user like his identity and location are also transferred. This information slowly finds ways into other usages like advertisements, tracking etc. Users are increasingly expressing concerns about such information being used by unknown agencies for unknown purposes [19]. Traditionally data encryption techniques have been in use to hide personal information from public consumption. Any attempt to access such data needed specific keys for decryption. While the notion of associating access permissions to end-use based on identification are useful, encryption and decryption of data are compute intensive and can be constraining on IoT scenarios. What is really needed is to be able to pass on useful information without losing out on privacy related information. Hence the need to identify mechanisms for data obfuscation wherein the responses to user queries can still be real-time without any compromise on the source information provider's privacy. In the associated study we dwell on some of the data obfuscation techniques that can be useful in GPS based location applications.

The following sections are organized as follows: Section II describes a published study [33] on a proactive framework for route updation based on real-time events in a GPS based navigation application; Section III describes a storage strategy [34] for the same application; Section IV highlights the importance of preserving privacy in location based services and observes some recent advances in this area; Section V brings up the challenges and directions for future research in view of the increasing interest in both IoT and compute clouds; Section VI concludes the paper with some musings on how this work can be furthered.

II. Framework for Real-time routing in GPS applications

Navigational applications mostly deal with the routing problem, wherein the shortest path between two given end-points is found. The path computation was traditionally based on reducing the physical distance of travel. With the increasing availability of rich time dependent data in the form of current traffic, weather conditions etc. and methods to compute its impact on the travelling time in the given road segment, location based services are increasingly becoming more demanding and parametric [5]. This mandates involvement of higher computational capacity and hence the need for a compute cloud backend to support the real-time requirements for these applications.

In this study the routing problem is modelled as a real time job scheduling problem where jobs are the shortest path queries submitted by the users and deadlines are computed on the fly depending on the user characteristics (current location, speed group, updates on real-time events, etc.). A pro-active framework is built which gets relevant data updates based on real-time updates, that affect a specific user query and the results are communicated back to the user without them specifically asking for it. This is done by sending revised route to the user before user takes the next turn on the pre-specified path. Potentially any routing algorithm [8][9] that is capable of dealing with real time network updates can be used to compute shortest path jobs in our frame-work; we demonstrate results with Dijkstra's algorithm.

A. *Motivation*

Many algorithms reported in recent literature, use history based modelling of road traffic to compute the travelling time [6][7][11][12]. They neglect the real time updates completely or accumulate the updates over time and adapt the model of road traffic accordingly [13]-[17]. But with the real time updates being available along the network, routing systems can now use it to provide better utility to its users. One common limitation of all these approaches is that after route computation, user has to specifically ask for route updation, when a refreshed route query is registered and computed using the current location. In this work, we instead propose a proactive system, which continuously tries to track the user and any updates on the path that the user is on. Consider the scenario in Figure 1. Here, initially when the user asks for a route from point A to point B, the system computes it as A-P-Q-B (Figure 1 (a)). Upon getting this result, user takes the road segment AP and starts moving. However, due to some event (an accident for example) as shown in Figure 1(b), traffic on edge QB has started slowing down, making A-P-R-B a more favorable path (Figure 1 (c)). If however, the system fails to communicate this to the user before she starts travelling on the road segment PQ, then this user ends up travelling on a less optimal path in spite of the system being able to prevent it. A proactive system, which updates the route for its users based on latest information along the network can avoid such scenarios and make sure all the information that it possesses is used to benefit maximum users.

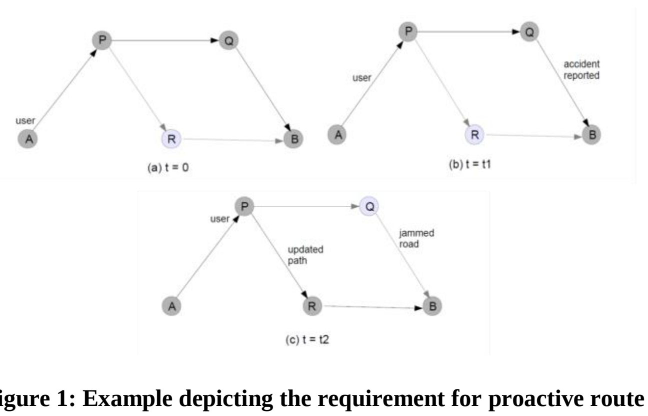


Figure 1: Example depicting the requirement for proactive route updates.

The other trend followed in related work is to use distributed algorithms which let the client decide which way to go based on their local view of the graph and updates relevant to it [15]. Such approaches offer limited utility particularly in view of recent updates based on events that are visible on global view.. In the next section, we propose a proactive framework, which deals with real time updates and tries to communicate the results back to the user in time, followed by our approach for query dependent route computation.

B. *Proactive Framework for route updation*

Routing in road networks is modelled as shortest path problem on a graph with positive edge weights. Users pose queries to find optimal paths between a source and a destination, and navigation systems typically respond to them in a best effort fashion. As discussed in the previous section, route re-computation is of utility to it's users if network condition changes and should reach them before they take the wrong turn. To enable this, we model the problem as a real time job scheduling problem, where such re-compute jobs come with a deadline.

Overall system model is shown in Figure 2. *Fresh jobs* enter in the job queue from one end, and are removed from the job queue for computation from the opposite end. The scheduler then adds a *refresh job* to the queue so that user can be updated about the changing network conditions and the path that has become more suitable as a consequence. These *refresh jobs* come with a deadline, which decides their priority and appropriate place in the job queue. The scheduler decides to execute the *refresh job* if the deadline to compute this next job has not expired otherwise it is dropped.

We make the following assumptions: Users can be polled for their current location and some mechanism is in place which analyses the graph updates in the form of current traffic, weather conditions etc. and appropriately adjusts the edge weights in the graph. For our purpose, we assume the edge weights represent the time it would take to travel on the road segment represented by the edge.

I. *Defining a Job*

Jobs are the shortest path queries submitted into the system either by the users, or by the system itself in response to updates along the network. Three categories of jobs are defined: *fresh*, *refresh* and *redo* as follows:

While *fresh jobs* are added by the users, *refresh jobs* are system generated as a proactive response to the changing network. *Redo jobs* can either be generated by user or system itself.

- *fresh job*: Represents a new user connecting to the system, asking for a shortest path. Each *fresh* job is given a unique ID which represents this user. Subsequent *refresh* and *redo* jobs carry forward this same ID.
- *refresh job*: Represents a case when user is following the path and system pro-actively decides whether the user should continue or switch to a different path.
- redo job: Represents scenario where system was late in giving response to the user, who is now on a different path. Or the user chooses to take a different path and requests for re-computation based on current

location. Presently, in our implementation we treat these jobs as fresh jobs in terms of scheduling priority.

All jobs in the system are aperiodic and independent of each other; they have no precedence relationship among them. All jobs are designed to be non-preemptive to enable relevance of computation with updates within their deadlines. Job scheduling is dynamic priority-driven. Job priority can change based on entering jobs or ageing factor, hence the priority assigned to a job initially when it enters the queue can change during its course of queue traversal till the time it is actually executed.

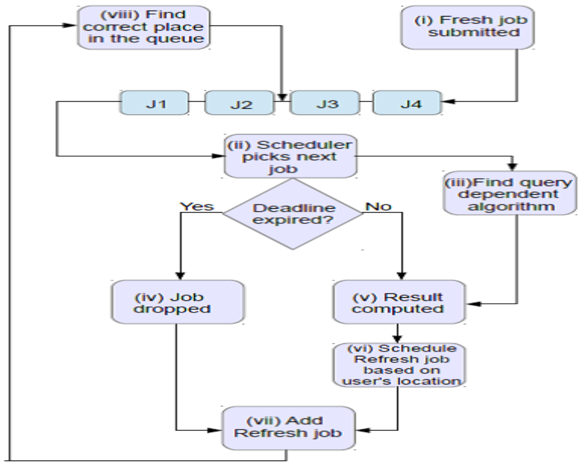

(viii) Find correct place in the queue
(i) Fresh job submitted
J1
J2
J3
J4
(ii) Scheduler picks next job
(iii)Find query dependent algorithm
Yes
Deadline expired?
No
(iv) Job dropped
(v) Result computed
(vi) Schedule Refresh job based on user's location
(vii) Add Refresh job

Figure 2: System Model for proactive route updation

Fresh jobs are submitted without any deadlines, and are added at the back of the job queue (Figure 2, block [i]). So their computation depends upon the length of job queue at the time of their addition. After any shortest path computation is completed (in the form of any job), a *refresh job* with source as the next hop and the same destination is added (Figure 2, block [vii]). For example, in Figure 1 after job (A,B) is computed, job(P,B) is added as a *refresh job*. *Refresh jobs* come with a deadline since the result of these jobs must be computed before the user reaches the next turn (vertex P in the above example), otherwise the user will go on the previously specified path. This ensures that for a given user ID, at-most one job exists in the system at a given time. A refresh job's priority is fixed based on its deadline and the current queue length. As a result when a refresh job is added into the queue, the earlier fresh job's priorities will change depending on where the refresh job is inserted. At the time of insertion priority re-calculation is done accounting for ageing also for the existing jobs in the queue. *Redo jobs* get added into the system when the system identifies that user has changed his route or when a user specifically requests a different route. These jobs normally get appended at the end of the queue.

II. Computing Job Deadlines

Refresh jobs come with a **scheduling window**, marked by two timestamp values that are unique to each *refresh job*. R*efresh jobs* should not be scheduled too soon because it is not very useful to re-compute a user's route as soon as they have started. Thus, the scheduling of a *refresh job* must be delayed as much as possible so we can get the most recent updates on the graph. We call Timestamp1 as the *maturity time* for a job, after which it is eligible for scheduling. A *refresh job* must not be scheduled before its Timestamp1 and must not be scheduled after its Timestamp2 value, which marks its *deadline*. These jobs have a deadline since the system must send the results back to the users before they take any turns forcing them to travel on a suboptimal path. These timestamp values are computed as follows and are illustrated in Figure 3.

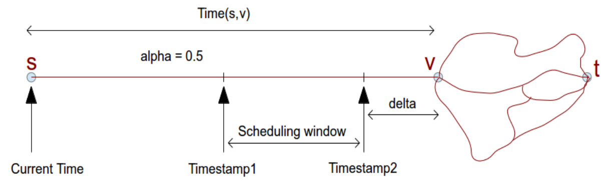


Figure 3: Illustration of Timestamp computation for refresh jobs

$$\text{Timestamp1} = \text{Current\_time} + \alpha * \text{Time}(s,v) \qquad (1)$$

$$\text{Timestamp2} = \text{Current\_time} + Time(s,v) \qquad (2)$$

$$\text{Timestamp1} \le \text{Scheduling\_time} \le \text{Timestamp2} \qquad (3)$$

Here α is any constant in the interval (0,1) which is used to delay the execution of the refresh job so that sufficient updates can be included. Function time(s,v) returns the time it would take for user to travel from point ***s*** to point ***v*** in the current network scenario. Δ is the upper limit on the estimated time taken to compute the shortest path and communicate the result back to user. Higher α value indicates that the route will be computed as late as possible and routing window will be narrow. α is used in the computation of the maturity time (Timestamp1) of a job and Δ is used in the computation of the deadline (Timestamp2) of a job. While scheduling of a job after Timestamp1 ensures that the latest updates in the network are included in the route computation, scheduling before Timestamp2 ensures that the results are sent to the user in time before she reaches the next junction and hence can take the appropriate turn. The job is dropped if the current time exceeds its Timestamp2 value since the computed result would reach the user after she passes the decision point in his path (Figure 2, block [iv]).

III. Job Scheduling

The job scheduling algorithm used in the framework is listed below. Relative priority between two *refresh jobs* is calculated using their deadlines or Timestamp2 values. Earliest deadline first algorithm with Timestamp2 as deadline is used to find the next job to be scheduled for all the jobs that have matured at a given time. Relative priorities between *refresh jobs* and the *fresh jobs* are dynamically assigned using aging for *fresh jobs*. A job queue is maintained, such that at any given time the first job in the queue is having the highest priority for scheduling. If, the job is a *refresh job* and has not matured at the time, then next job is checked and so on. Insertion in the job queue (Figure 2, block [viii]) follows the rules as described below. Assume queue state as shown in Figure 4(a). Here Queue is maintained as a linked list, where cell with 'R' denotes a *refresh job* and 'F' denotes a *fresh job*. Queue head is on the left, from where next job to be computed is searched.

Case I: If a *fresh job* is to be inserted in the job queue, it is added in the back as shown in Figure 4(b).
Case II: If a *refresh job* is to be inserted, such that its Timestamp2 value falls between two consecutive refresh jobs, it is simply added in the middle as shown in Figure 4(c).

Algorithm 1 Job Scheduling
```
while (1) do
    while !(Queue Empty) do
        ptr ← queue head
        if ptr.jobtype = fresh then
            goto compute
        else
            if ptr.jobtype = refresh & matured(ptr) = true then
                goto compute
            else
                if ptr.jobtype = refresh & expired(ptr) = true then
                    dropped jobs ++
                    goto add
                else
                    ptr ← next(ptr)
                end if
            end if
        end if
    end while
    compute:
      compute shortest path
    add:
      find next hop
      add refresh job
end while
```

In case there are *fresh jobs* in between, then depending upon the priority, the position of the new job changes. If *refresh job* is given higher priority over *fresh jobs* then case III is followed as shown in Figure 4(d). If *fresh job* is given higher priority than case IV is followed as shown in Figure 4(e). If the relative priorities of *fresh jobs* and *refresh jobs* are dynamically computed, then any of the case III, IV or V [Figure 4(f)] can be followed. In this framework, dynamic priority is assigned between *fresh jobs* and *refresh jobs* by giving an aging factor to the *fresh jobs*. Initially Case III is followed, giving higher weightage to a *refresh job* over *fresh job*. But, each time a *fresh job* is pushed back in the queue to make room for a *refresh job*, its aging factor increases. If the age increases beyond a threshold, all other *refresh jobs* are added behind it following case IV and case V. Addition of *refresh job* is explained in algorithm2 below.

When a *fresh job* enters queue, its age is 0. After every insertion of a *refresh job* before it, its age is incremented. If its age value exceeds some predefined threshold, then it gets higher priority than all the other *refresh jobs* that will enter the system from then on. Consider two refresh jobs denoted as R and fresh job as F.

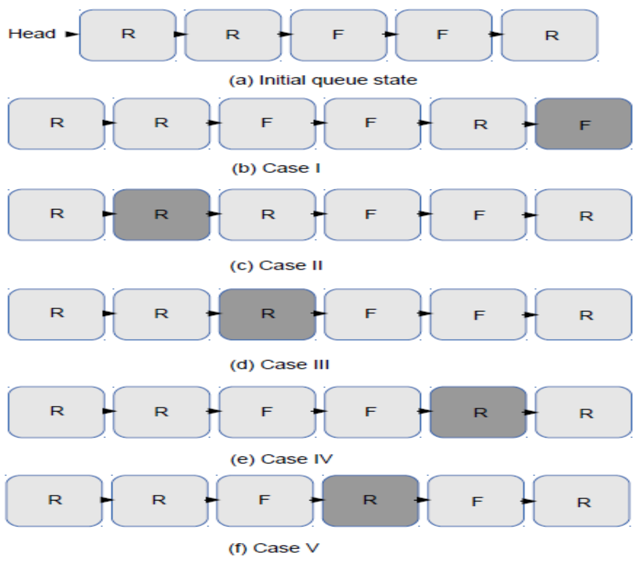


Figure 4: Job Queue Insertion mechanism illustration

The following equations define the relative priorities.
If, Timestamp2(R1) < Timestamp2(R2), then:
$$Priority(R1) \rangle Priority(R2) \quad (4)$$
If, AgingFactor(F) < Threshold, then:
$$Priority(R) \rangle Priority(F) \quad (5)$$
If, AgingFactor(F) $\geq$ Threshold, then:
$$Priority(R) \langle Priority(F) \quad (6)$$

Algorithm 2 Insertion of refresh job
```
if (Queue Empty) then
    queue head ← next_refresh_job
    return done
end if
ptr ← queue head
while ptr != NULL do
    if ptr.jobtype = refresh && ptr.timestamp2 > next_refresh_job.timestamp2 then
        insert next_refresh_job before ptr
        return done
    end if
    if ptr.jobtype = fresh && ptr.age < threshold then
        insert next_refresh_job before ptr
        temp ← ptr
        while temp.jobtype = freshjob do
            temp.age ++
            temp ← next(temp)
        end while
        return done
    end if
    ptr ← next(ptr)
end while
add next_refresh_job at queue end
return done
```

C. *Case Study and Results*

This section describes a case study and results based on the road networks of various cities of USA as provided by the Dimacs Implementation Challenge [10]. Specifically, the Great Lakes, USA (LKS) graph instance, which has 2758119 vertices and 6885658 edges, was used for testing the proactive real time framework. All experiments were performed on a system with 8GB memory, Intel Core2 Quad CPU Q9550 @ 2.83GHz processor and 300 GB disk. The framework implementation is C based, using pthreads library. The graphs are static in nature, and the edge costs represents the time it takes to travel on the road segment represented by the edge. Due to the lack of availability for dynamic datasets, we used random updates along the graph to demonstrate our model. Random shortest path queries are generated using internal Unix *bash* function ***RANDOM.***

Main thread, called the scheduler, initializes a fixed number of worker threads whose job is to compute shortest path and add a refresh job if the destination hasn't been reached yet. Scheduler manages the job queue, and selects the next job to be computed by traversing the queue from the head. The first fresh job or matured refresh job is picked and given to a free worker thread. A separate thread focuses on updating the edge weights along the network. New jobs are regularly fed to the scheduler in a random order.

In this study the basic comparison is with that of FIFO queue setup wherein first-in-first-out priority order is intrinsically maintained. This correlates to the current scenario where requests are responded to in their order of arrival. For our framework FIFO responds similar to the best effort case since job deadlines are not maintained. The proposed framework on the other hand prioritizes jobs based on their deadlines and dynamically changes the priority using aging principle. The aging factor for fresh jobs for our simulation is fixed at 3. This value ensures that at most three *refresh* jobs can push a *fresh* job back in the queue. Also, computations whose results cannot complete within their deadlines are dropped.

For the simulation, keeping everything else same, jobs are added in the FIFO queue using their Timestamp1 values and are computed in first come first serve basis without any regard to their deadlines. If a job is computed after its Timestamp2 value expires, then it is considered as wasted computation and termed as *dead job*. The number of dead jobs in FIFO is then compared with the number of *dropped jobs* from the proposed framework.

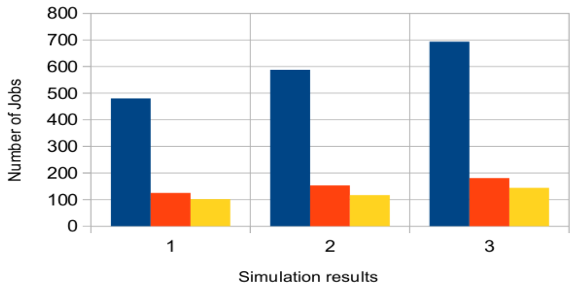


Figure 5: Simulation with α = 0.5 and two worker threads

Figure 5 shows the simulation results with α = 0.5. Here, total queries (*fresh* and *refresh*) submitted are shown in blue, red bar shows the number of dead computations being made in case of FIFO. Yellow bar shows the number of dropped jobs in the proposed framework. The graph clearly demonstrates two key results. First the number of dropped jobs in the proposed framework is lower than the dead jobs in the FIFO queue. This is because of the dynamic priority mechanism which schedules jobs based on their deadline. By correctly ordering jobs based on their deadline, we are able to attend to more jobs. Secondly, in our framework total number of jobs Also increased since a lot of wasteful query computation is avoided. The yellow bar indicates jobs that were not executed since it was not possible to meet their deadlines, while the red bar indicates jobs that were executed but found to be of little use since they completed execution after the deadline. This indicates that by assigning deadlines and ordering job execution based on dynamic priorities that are derived from the deadline constraint and by avoiding computations that are bound to miss deadlines, the proposed framework offers better utilization of compute resources and also increased job throughput. Further, it can also be seen that for an increasing number of total queries, number of *dead* jobs by FIFO also increases, and stays higher than the number of *dropped* jobs in our framework.

Another parameter that affects the way computations are scheduled is α metric. As explained earlier, this metric decides how soon the refresh jobs can be launched once a user chooses a traversal path. Higher α means that the refresh jobs are delayed longer enough to include most recent updates in the results. However, this potentially can result in increased number of dead and dropped jobs because there is not enough time to complete the refresh jobs because of delayed launching. In this case it is better to use our framework as FIFO will spend a significant percentage of time in wasteful computations. Decreasing the value of α means that it is not required to include very recent updates along the network in the path computation, in which case our framework works similar to FIFO. Figure 6 shows simulation results for α = 0.6 for the same number of total queries.

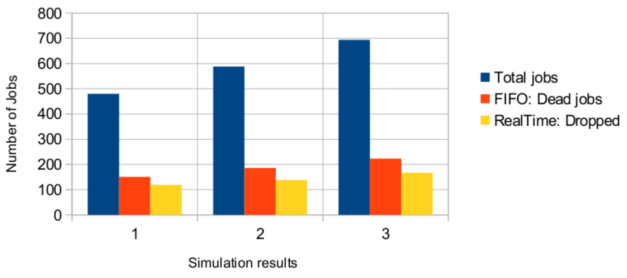


Figure 6: Simulation with α = 0.6 and two worker threads

III. EFFICIENT STORAGE STRATEGY FOR BIG-DATA IN GPS APPLICATIONS

From Google Maps to consumer Global Positioning System (GPS) devices, society has benefited immensely from routing services and technology. The issue with these applications is that they derive their utility by analysing large amount of data from many sources and hence most applications today use only static information. Higher utility for these applications is possible if they can adapt to changing events in real-time. In order to achieve this, better computing and storage mechanisms need to be explored to store and process information based on real-time events surrounding the areas of interest. Interesting aspect about this is that, the areas of interest and events associated with it are highly specific to an application and also the user queries associated with the application even though all these might be using the same basic source of geo-spatial information.

Here we specifically look at the GPS based data and its usage to integrate real-time data updates regarding Geo-spatial networks from the perspective of data distribution and storage such that applications can be built for real-time responsiveness. Methods to store Geo-spatial data are either centrally located or distributed [21] [27][28][29][30]. Most of the data distribution strategies are based on one time computation of the query response. Extending such algorithms to support time-sensitive decision making is a challenge. Not only the limited time response but also accurate response is what is intended. Hence, an efficient mechanism that is effective to meet application response deadlines is what is desired.

Generally, in navigation kind of applications a query in real-time is of the form compute route from A to B with special parameters. Distributed storage gives chance to parallelism when many different route queries have to be answered to achieve speedup. Further, in order to address the need of real-time updates we need to have efficient mechanisms to store, manage and distribute data. The storage mechanism is critical in the computation of route from one location to another. The basic idea is to organize data in a distributed way and always identify and associate the computation on the data to where it is located. This allows for reduction in time for responding to the query since communication overheads are reduced.

The key contributions of this work are:

1. Designing of a storage mechanism that exploits redundancy, to reduce the communication overheads and failure of nodes.
2. Schedule computation closer to the data location and thus reduce latencies associated with data communication overheads.
3. The data load is balanced over the nodes with some relaxation to ensure better utilization of resources and make sure that load balancing is not a frequent operation.

The following sections detail the *NineCellGrid* methodology proposed for storing the data and describes how computations in routing query can be generated using this mechanism. Experimental results demonstrate observations of being an effective storage strategy.

A. *NineCellGrid Method*

The focus of the design approach here is to address the requirement of storing time-based updates on the geo-spatial data. We choose the distributed dissemination method and use the N*ineCellGrid* method for storing data. In this method, data is distributed not based on the available storage nodes, but based on the region of area on which the computation is intended.

I. *Design Approach and Storage Strategy*

The GPS data that is concerned to a location is first decomposed in such a way that entire earth is mapped with a mesh/grid of cell dimension L X L, where L is some fixed value (explained in the following sub-section), as shown in Figure 7.

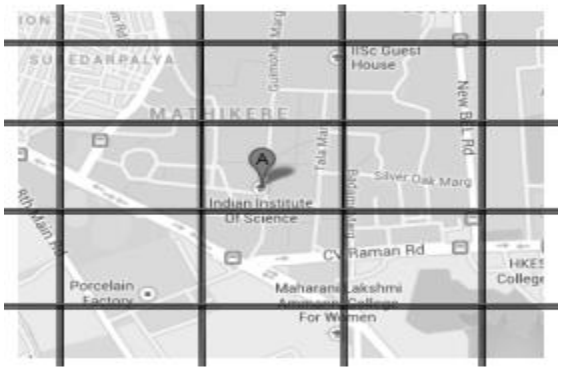


Figure 7: Grid on the Geo-spatial map of earth

Further, in this approach each cell can hold the data of 8 other cells that are present adjacent to it as shown in Figure 8. Hence, the query that can be inherently answered by a cell can now be answered by all the 9-adjacent-cells that contain that data. Each cell is representing a unique L X L area on earth as in Figure 7.

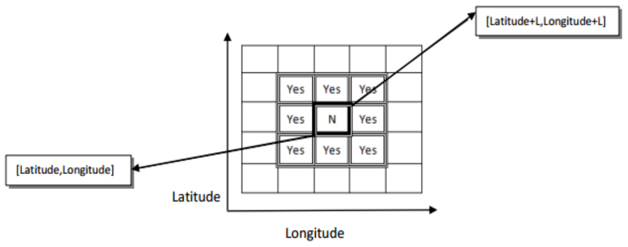


Figure 8: Cell N, with identity [Latitude, Longitude] in NameTable

II. *Name Table*

A table called *NameTable* (Table 1) that is indexed with row key as the 'Latitude' and column key as the 'Longitude' which indexes the address of the node which is responsible for that cell or region in reality of dimensions L X L. The actual region that is covered by the node is a square of four vertices mentioned as [(Latitude, Longitude),(Latitude + L, Longitude),(Latitude, Longitude + L),(Latitude + L, Longitude + L)].

Table 1: *NameTable* - Here *f* and *g* are mappings from miles to Longitude and Latitude units

		Longitude				
		-74.499998	-74.499998 + *f*(L)	-74.499998 + 2**f*(L)	...	-73.500016
Latitude	40.300009	*<NodeAddress>*	...	..	...	
	40.300009 + g(L)	*<NodeAddress>*	...	..	...	
	40.300009 + 2*g(L)	*<NodeAddress>*	...	..	...	
	...	*<NodeAddress>*	...	..	...	
	41.299997	*<NodeAddress>*	...	..	...	

III. *Calculating Dimension **L** of the cell*

Each cell is of length ***L***, where ***L*** is in the standard units of Longitude and Latitude. The reason for replicating nine cell's data within one cell is to make sure that any 'A-to-B route' query of Euclidean distance ≤ ***'L'*** can be addressed by a single cell. ***L*** is chosen in probabilistic manner based on historical data considering the previously observed queries.

Consider all the possible locations of source A, in query "A-to-B" within a cell. This would be a filled square of the cell as shown in Figure 9(A). The locus of point B, such that B is no more than L Euclidean distance away from A is a rounded square with corners as a quarter circle of radius ***L*** as shown in Figure 9(A). To accommodate all the data within a single cell, the best fit would be this Nine Cell Grid shown in Figure 9(B). That is, all the surrounding 8 cells data to be located in this central cell so that it has the complete coverage of B in any case without having to communicate with neighboring cells/nodes.

A general query that is processed in these GPS-applications is to calculate the route from location A to location B. Given such a query, the ***L*** value is so chosen that any query made by the traveller using this application has the A-B Euclidean distance not more than ***L*** for 80% of the total queries. In order to achieve this criterion, L is chosen as follows:

$$L = min\,(X : \,Euclidean\,distance\,of\,P\%\,of\,queries\,are\, \leq Euclidean\,distance\,of\,X\,) \qquad (7)$$

Here, ***P*** is the percentage that gives better performance.

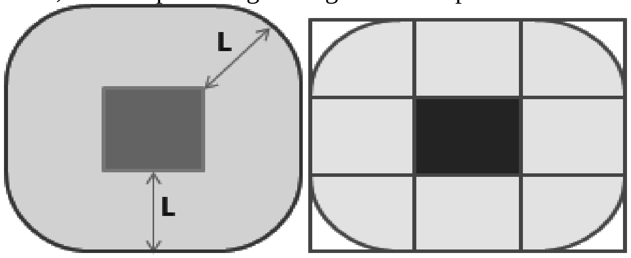


Figure 9: NineCellGrid – (A) Locus of B with A inside the cell (left side picture); (B) Best-Fit for all B with data in the same cell (right side picture)

Choosing the correct value of ***P*** is critical to achieving the design goals and indeed dictates the storage strategy. For the dataset used in the experiments considering ***P = 80%*** of the previously observed queries on the dataset, we get the optimal performance (described in results section). Interestingly, if the percentage is considered high, that is something close to 100% it tends towards centralized storage pattern. On the other hand, if a low percentage is considered then it tends to a scenario wherein most queries will require distributed computation or high data communication. A judicious selection of ***L*** will decide the maximum utility that can be derived out such applications.

IV. 'A-B' Route query Case Studies

In the ninecellgrid method, some extra information is stored for the edges of the graph that are cutting the grid. This information is kept with the vertices of those edges, also called the limiting vertices. For queries that need to cross a cell these limiting vertices provide the necessary information about the traversal edges and the associated data location node. Based on the computation strategy, either the data is fetched from that node or computation is forwarded to that node. Different scenarios of an A-B query that can arise and how they are addressed based on this storage method are illustrated below.

Case 1: Both A and B lie in the same cell of L X L

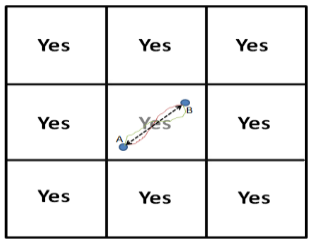


Figure 10: Route query where A and B lie in same cell

With both A and B being in the same Cell (Figure 10), the query can be answered by 9 Nodes. Based on the current load on the node an appropriate idle or less loaded node can be selected to process the query. In our setup, the initial assignment is to the central node which A and B. If that node is busy, then the next preference is given to that cell whose sum of Euclidean distances from center of the cell to source A and destination B is least. Job-scheduling is done this way to ensure better task load balance and yield better throughput.

Case 2: A and B lie at 1-Cell distance apart

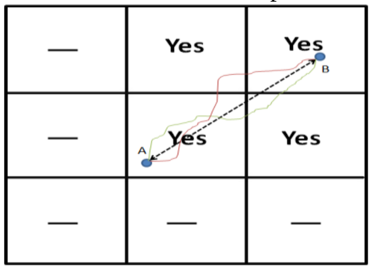


Figure 11: Route query where A and B lie at 1-cell distance apart

In this case number of nodes that can compute the query are 4-6 nodes. Figure 11 shows the scenario of 4 nodes. In this case the priority is given to the node with least sum of Euclidean distances from center of this node to both source A and destination B. However, based on the task load the node, which has next least sum of Euclidean distances from center of the node to A and B, is considered and so on.

Case 3: A and B lie at 2-Cell distance apart

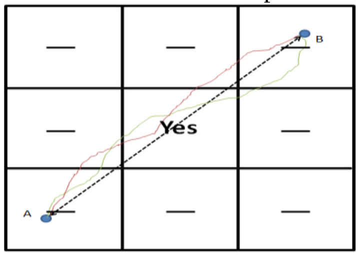


Figure 12: Route query where A and B lie 2-cell distance apart

In this case number of nodes that can compute the query are 1-3 nodes. Figure 12 shows the scenario of 1 node. It has lesser flexibility than the previous two cases in terms of nodes for scheduling query processing. But, this case happens with the probability of less than 0.2 because of ***L***. Similarly, in this case the priority is given to the node with least sum of Euclidean distances from center of this node to both source A and destination B.

Case 4: A and B lie at a distance of more than 2-cell distance apart

If the query is such that the end point of the query A and B are more than 2-cell distance apart, data from the nearest neighbours, based on the information from the limiting vertices, is gotten to the cell that initiated the computation. The other way to handle such a query is to send the job state to the neighbour node whose data is needed and resume the process on that neighbouring node. In order to get complete data of the region, a node has to communicate with at-most 11.11% of the total nodes only thus reducing query processing latency to a large extent. Thus choosing the correct ***L*** is critical to this storage method.

V. *Load Balancing*

Considering the geographical map of the earth and knowing that road networks are restricted to land masses, uniform distribution of data based on just the *NineCellGrid* will lead to many nodes having data that is of little use. Hence, after arriving at the value of ***L***, data re-distribution across nodes is necessary to equalize computational load across the nodes. It is important to note that the storage strategy here assumes that each node is both a data store and an execution node too. The load balancing is done to distribute the data across different nodes to handle query density on the region. The algorithm used to load-balance is listed in the algorithm below.

The load is distributed and load balance is attempted with a chosen relaxation parameter ***δ***. Initially the data is distributed to each ***L X L*** cell considering number of junctions/vertices, roadlinks/edges, landmarks, etc. Mean of estimated load across cells, called as ***M***, is calculated. The load is balanced with relaxation ***δ*** i.e., a cell is said to be balanced if its load lies in [***M-δ, M+δ***]. Relaxation depends on logistics like resources available, rate of data increase etc. To begin with each cell with load less than ***M-δ*** is set to zero and its data is transferred to one of its nearest neighbour that is having load value more than ***M-δ***. After this iteration, each will have zero load value or load value greater than ***M-δ***. Those cells with load value greater than ***M-δ*** are shared by N nodes such that after dividing the load value with N the resulting value lies in [***M-δ, M+δ***]. This load balancing exercise improves resource utilization.

Algorithm: ***Load Balancing***

Inputs: [*NameTable*; Relaxation δ]

Outputs: [Load balanced distribution; Updated *NameTable*]

```
1.  Initialize load based on number of edges, vertices, etc.,
2.  for each node $E$ do:
3.      if( Load(E) $< M - \delta$ )
4.          Find node P closest to E with load $\geq M - \delta$.
5.          Load(P) $\rightarrow$ Load(P) + Load(E);
6.          Load(E) $\rightarrow$ 0;
7.          NameTable[Latitude(E)][Longitude(E)] = Address(P).
8.      endif
9.  endfor
10. for each node $E$ do:
11.     if( Load(E) $\neq 0$ )
12.         Share the load among N nodes.
13.         $(Load(E)/N) \in [M - \delta, M + \delta]$ & N $\in$ (1,2,3,...)
14.         Update NameTable.
15.     endif
16. endfor
```

B. *Experimental setup and Results*

Simulation studies on the application-level performance of centralized, fully distributed and the proposed ***NineCellGrid*** approach based data distribution models is presented in this section. Simulations are achieved using MPI for task distribution and Hadoop Distributed Filesystem [32] for data distribution and storage. The same New York city dataset [10] that was used for the routing experiments is also used here (Figure 13).

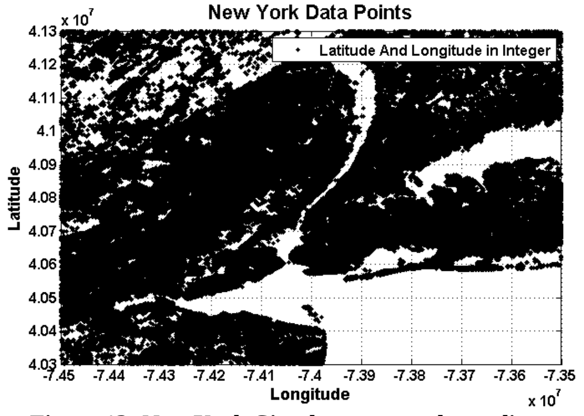


Figure 13: New York City dataset actual coordinates

The load estimation of a node is done considering the number of junctions/vertices, links/edges in the region assigned to it. The value of the node's load is shown as the sum of all these values. Figure 14 and Figure 15 show the load distribution of the nodes before replication, after replication and finally after the load-balancing respectively. These figures show the load in color based on the value of load a node has. Regions with water have low value of estimated load and are shown with low color value in the scale. The cells with higher load have higher color value in the scale i.e., towards red. After the load balancing (Figure 15) the regions with less than ***M-δ*** load values are assigned to nearest cell with load value greater than ***M-δ***. Hence, the regions associated with lower load values move to lower color values.

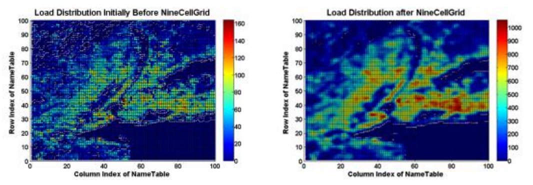


Figure 14: Initial Load distribution before (left) and after (right) replication

Here, the relaxation used is ***δ = 75***. So, the load values in the Figure 15 would be between 150 and 300 as the ***M*** is around 225 or zero.

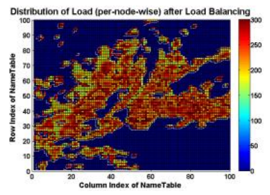


Figure 15: NineCellGrid Load distribution after load-balancing

To get a better perspective of the effect of load balancing, Figure 16 **Error! Reference source not found.**and Figure 17 show the histograms of load distribution of the nodes before replication, after replication and finally after the load-balancing respectively. Comparing the load distribution before and after replication it is observed that the maximum value of the load before replication is one-ninth that of the load after replication. This is due to the fact that replication is done almost nine times. Figure 17 captures the scenario after load balancing with ***δ=75.*** We notice that by load balancing the nodes required for data distribution reduces significantly and well below the naïve distribution before replication.

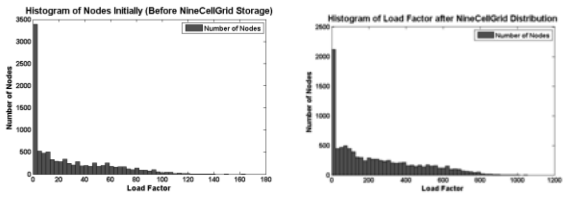


Figure 16: Histogram of initial load distribution before (left) and after (right) replication

The selection of ***P,*** which denotes the query density for a specific region based on historical observation data, which determines the ***L*** value, is done based on the analysis shown in the

Figure **18** (A). To begin with, taking 70% as the initial guess it is increased in steps of 5% so as to get to the optimal percentage. The graph shows 80% as optimal for this dataset. Same value of ***P*** may not be applicable to different geographical regions as the query density depends on the city, country, etc. It is perceivable that different ***L*** values will benefit different locations. Such derivation of ***L*** will hold when this algorithm is extended to cover larger areas like a district, state, country or the whole world. The study presented here is limited to a single city and hence choosing one value of ***L*** is justified.

The impact of different schemes of data distribution on application performance is analysed by running a Dijkstra's greedy algorithm for shortest path computation. MPI library is used to simulate the algorithm. ***L*** value for the dataset New York City is taken as 6.8 miles (considering ***P = 80%***) from NHTS (National Household Travel Survey) [31]. The performance is measured as throughput representing the number of user queries that can be answered in a given time unit.

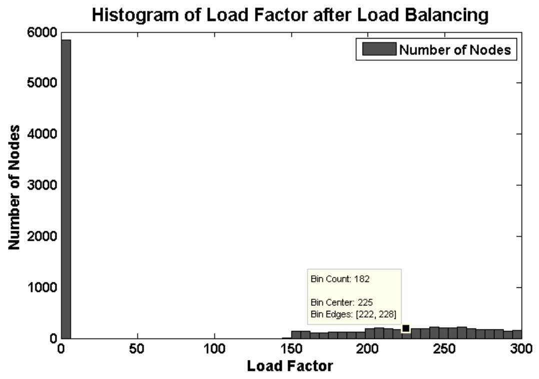
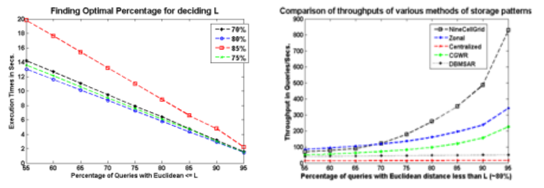


Figure 17: Histogram of NineCellGrid load distribution after load balancing



(A) (B)

Figure 18: (A) Execution times vs. Percentages of queries with Euclidean distance of query AB less than L
(B) Throughput Comparison of various storage methods

Error! Reference source not found., shows the comparison of throughputs for various storage methodologies. In **Error! Reference source not found.** the notations, *NineCellGrid* represents the proposed *NineCellGrid Method*, *Zonal* represents fully distributed pattern, '*Centralized*' represents centralized storage pattern, '*CGWR*' represents *'NineCellGrid Without Replication'*, and finally '*DBMSAR*' represents the DBMS as replication (replication value of 5 taken). Each of the methods is compared based on the throughput they yield. This performance is measured against the workload of user queries wherein the number of queries falling within the ***L*** distance is varied from 55% to 95%, as represented by the x-axis in Figure 18 (B). The total number of queries answered by each setup is constant at 1000 queries. Graph in Figure 18 (B) shows 'Percentage of queries with Euclidean distance less than ***L***' vs. 'Throughput in Queries/Sec'.

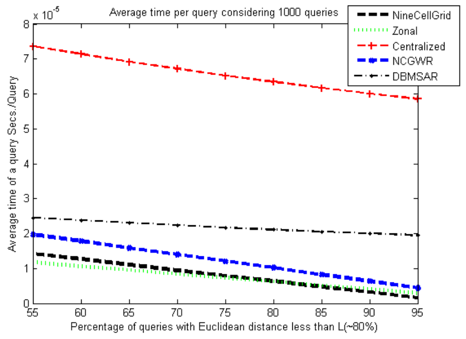


Figure 19: Average Time Per Query Comparison of various storage methods

The throughput of *NineCellGrid* starts dominating other methods from 70% onwards. On comparing the throughput of a general scenario of above 80% expectancy, it is seen that *NineCellGrid* shows better throughput values than the others. For the same set of compute resources, the difference between the *Fully distributed* (zonal distribution of data) and *NineCellGrid* is about 470 (810-340). Hence, it can process about 300 more queries in a second.

Figure 19 shows the 'Percentage of queries with Euclidean distance less than L' versus 'Average time per query taken in seconds'. It is observed that the *NineCellGrid* storage method has least 'Average time per query' after about 70% onwards. Fully distributed has better 'Average time per query' before 70% because of the fact that they communicate relatively less at that stage. The other storage methods exhibit higher average times. The centralized has highest since there is no data parallelism. '*CGWR*' (*NineCellGrid* without replication) has relatively higher average times than *NineCellGrid* and *Fully-distributed* since there is no replication at all, leading to high data communication.

IV. Data obfuscation to ensure user privacy in GPS applications

Incorporating real-time responsiveness to user queries and choosing data storage mechanisms that enable such processing are fundamental to most IoT based applications. The third facet to this class of applications is the amount of personal information that can be made visible while enabling specific functionality to the user query. Traditionally data encryption techniques have been in use to hide personal information from public consumption. Based on the trust agreement between the user and the service provider, information to decrypt is distributed. Any attempt to access such data needs specific keys for decryption. While the notion of associating access permissions to end-use based on identification are useful, encryption and decryption of data requires associated software, is compute intensive and can be constraining on IoT scenarios. What is really needed is to be able to pass on useful information without losing out on privacy related information. Hence the need to identify mechanisms for data obfuscation wherein the responses to user queries can still be real-time without any compromise on the source information provider's privacy or the end user data.

Based on this requirement, in this section we propose an architecture that essentially obfuscates the user's location information when he is querying for a route from a source to destination. The basic idea behind this is to change the location of the desired source and destination to obfuscated locations which are in the close neighbourhood of the respective source and destination. The key idea is that, given the route from the obfuscated source to the obfuscated destination, it is possible to derive the path from the actual desired source and destination. Following sections describe how this achieved.

A. Architecture:

The intuition behind the novel architecture which is describer here is to get trusted service by making use of different independent untrusted services. The proposed architecture provides user privacy in navigation applications by using neighborhood finding location based services (LBS) and an anonymizer. This protocol used by the architecture to provide protection against various attacks is depicted in Figure 20.

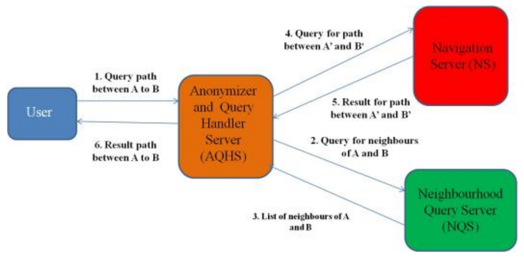

User
1. Query path between A to B
6. Result path between A to B
Anonymizer and Query Handler Server (AQHS)
4. Query for path between A' and B'
5. Result for path between A' and B'
Navigation Server (NS)
2. Query for neighbours of A and B
3. List of neighbours of A and B
Neighbourhood Query Server (NQS)

Figure 20: Location Obfuscation System Architecture

In addition to the navigation server, which actually computes the route from a given source-destination pair, we have introduced two more services, namely, the Anonymizer and Query Handler Service (AQHS) and the Neighborhood Query Service (NQS). In this system following protocol is used:

Step 1: User queries for path between A to B to the Anonymizer and Query Handler Server (AQHS).

Step 2: AQHS removes the identification information from this request and queries for neighbourhood locations of point A and B to Neighbourhood Query Server (NQS).

Step 3: NQS gives a list containing neighbourhood points of point A along with distance of those points from point A. Similar list is also sent for point B by NQS to AQHS.

Step 4: AQHS takes a random point from neighbourhood list of A, called A'. Similarly, AQHS takes a random point from neighbourhood list of B and selects the point B'. The AQHS then queries for path between A' and B' to Navigation Server (NS).

Step 5: NS sends an optimal path between A' and B' to AQHS.

Step 6: AQHS checks if point A and B are present on this path. If point A and B are present on this path, then AQHS filters out the path between A and B from optimal path between A' and B' (Figure 21a) and sends that path to the user.

Let us assume that filtered path is not optimal. In that case, just by replacing this path with optimal path between A and B, it will be possible to get another optimal path between A' and B' with lesser cost than the optimal path which is sent by NS. This is a contradiction. Hence, this filtered path is optimal path between A and B.

If in case, point A and/or B are not present on this path then following 3 cases can occur:

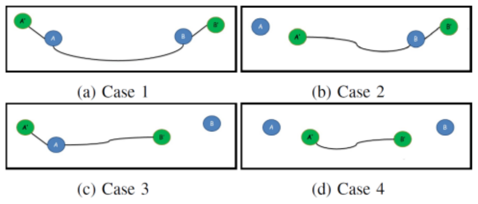


Figure 21: Different cases for query processing

- Point A is not on the path between A' and B' (Figure 21b): In this case, AQHS filters out the optimal path between A' and B from optimal path between A' and B'. Now as point A is in neighborhood of point A', AQHS makes an approximation of concatenating point A in path A'B to get path AB and sends that path to user. Note that path AB may not be an optimal path but experiments show that for real world graph this approximation works really well.

- Point B is not on the path between A' and B' (Figure 21c): In this case, AQHS filters out the optimal path between A and B' from optimal path between A' and B'. Now as point B is in neighborhood of point B', AQHS makes an approximation of concatenating point B in path AB' to get path AB and sends that path to user.
- Both points A and B are not on the path between A' and B' (Figure 21d): In this case, as point A and B is in neighborhood of point A' and B', AQHS makes an approximation of concatenating point A and B in path A'B' to get path AB and sends that path to user.

In this protocol, as queries are sent through the AQHS, user's identity is hidden from NS. Moreover, actual location of source and destination is also obfuscated from NS. Hence, this architecture provides two layers protection to preserve user privacy.

B. *Experimental setup and Results:*

Simulation studies and results of proposed architecture are presented in this section. All experiments were performed on a system with 4 GB memory, Intel(R) Core(TM) i5-2430 CPU @ 2.40GHz processor. Servers are made using flask which is a micro framework for Python based on Werkzeug, Jinja 2 [36]. For evaluation of the services, clients are generated using python based scripts. The architecture is evaluated on the basis of 2 evaluation matrics i.e. delay and error caused due to the proposed protocol.

Table I lists the 4 USA road networks that are used for experimentation [10]. The nodes in the graph represent the intersection of roads. The edges of graph represent the roads in that particular region.

Name of Region (Graph)	No. of nodes	No. of edges
California and Nevada	1,890,815	4,657,742
Colorado	435,666	1,057,066
Northeast USA	1,524,453	3,897,636
Florida	1,070,376	2,712,798

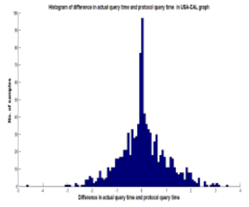
TABLE I: Graphs used for Simulation.

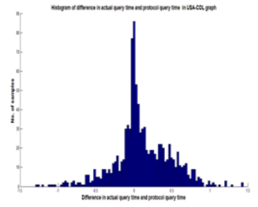
Name of Region (Graph)	Actual query time (sec)	Protocol query time (sec)
California and Nevada	14903.71	14842.73
Colorado	3113.2097	3004.53346
Northeast USA	13080.988	13041.1418
Florida	8688.89201	8659.96642

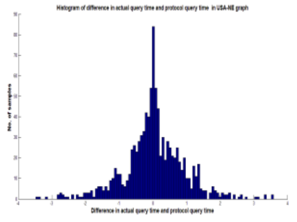
TABLE II: Total query processing time for 1000 queries.

Actual query time is the time taken when a user queries path from source to destination directly to navigation server. Protocol query time is the time taken to serve query by using protocol proposed in this paper. Figure 22 shows the histogram of difference in actual query time and protocol query time. In this figure, the X - axis shows difference in actual query time and protocol query time in seconds, and the Y - axis shows number of samples (queries). Figure 23 shows histogram of percentage error in distance. Here, the X - axis shows percentage error and the Y - axis shows number of samples (queries).

Complexity of single source - single destination Dijkstra algorithm depends on topology of graph. It is observed that in some cases, time required to get path AB is more than the time required to get path A'B' where A' and B' are neighbours of point A and B. NQS gives the list of all nodes which are adjacent to node for which query has come. Time required to get neighbourhood points and communication delays are of the order of milliseconds. Hence, complexity of protocol is actually complexity of Dijkstra algorithm. Due to these facts, inFigure 22, difference in actual query time & protocol query time follows a Gaussian distribution with center at 0.


(a) California and Nevada


(b) Colorado


(c) Northeast USA

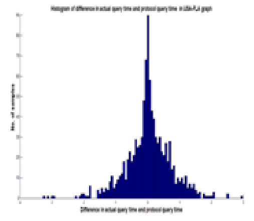
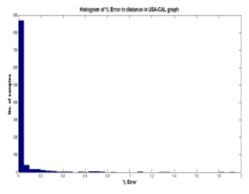
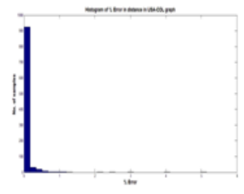
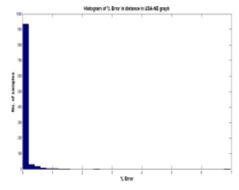

(d) Florida

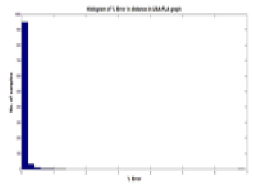
Figure 22: Histogram of difference in actual query time and protocol query time

It can be concluded from table II that on an average this system works as good as the system in which user queries path from source to destination directly to navigation server. Moreover, this system provides user privacy. It is also observed that percentage error in distance due to approximation made in this protocol, is less than 0.2% in most of the queries (more than 90% of queries). The number of queries with percentage error more than 0.2% decreases exponentially with very high decay rate. Thus, experiments show that user privacy can be maintained without affecting navigation results and without incurring significant overheads.


(a) California and Nevada


(b) Colorado





(c) Northeast USA **(d) Florida**

Figure 23: Histogram of percentage error in distance

V. Cloud IoT Frameworks Future Directions

To quote Doug Fisher, Intel corporate vice president and general manager of the Software and Services Group "The rise and ultimate success of the Internet of Things depends on the ability for devices and systems to securely and reliably interconnect and share information," [20]. While it is important to address the issue of IoT based interactions, it is time to look at the emerging application in this scenario and identify useful characteristics and traits of these applications to build systems framework that can seamlessly support them. This paper essentially does that. Here we have identified the key characteristics that most location based services have to adopt to, namely, adapting to changing real-time events, build storage constructs to capture the data associated with the changing events and modulate processing to the changes at the same time maintaining the desired privacy of the user. In the paper we choose the navigator as an example application and demonstrate the methods that can be used to build such cloud frameworks. We detail a real-time scheduling algorithm to advise proactively navigator users on changing situation along their traversal path. We also demonstrate a storage mechanism that can aid such scheduling and computation. We are currently working on the user privacy associated aspects of this application and believe that interesting data obfuscation solutions exist.

Thinking futuristically we believe cloud platforms can offer the rich and diverse applications hosted on IoT feature enhancements desired for better utility. Many initiatives using GPS based applications have proven to be effective and useful. Emerging applications that aid building of smart cities like disaster warning for earthquake prone areas, waste management and disposal monitoring by municipal corporations under clean cities initiatives, EMF level warnings for health hazards near nuclear or wireless equipment and towers, etc. are now becoming ubiquitous. However, all these have limitations because at present they cannot modulate to changing events and associated data change relating to the impacting events. However, how a change effects an application is highly localized to the application and it necessitates all the three dimensions identified here. It would be interesting to study and explore how different strategies work for different applications and this can bring in innovation in using the cloud as the versatile platform to experiment and potentially even deploy future applications.

References

[1] Matt Petronzio, "11 Fascinating facts about Google Maps", Aug, 2012, Available: http://mashable.com/2012/08/22/google-maps-facts/

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs,"*Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Experimental Algorithms*. Springer, 2008, pp. 319–333.

[4] H. Bast, S. Funke, P. Sanders, & D. Schultes,"Fast routing in road networks with transit nodes",Science,vol.316,no.5824,pp.566–566, 2007

[5] P. Sanders and D. Schultes, "Engineering highway hierarchies," in *Algorithms–ESA* 2006. Springer, 2006, pp. 804–816.

[6] D. Delling and D. Wagner, "Time-dependent route planning," *Robust and Online Large-Scale Optimization*. Springer, 2009, pp. 207–230.

[7] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies." *ALENEX*, vol. 9, 2009.

[8] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for a*: Efficient point-to-point shortest path algorithms."*ALENEX*, vol. 6, no. 2, 2006, pp. 129–143

[9] Abraham Ittai, et Al, "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms", *ACM-SIAM Symposium on Discrete Algorithms*, pages 782-793, 2010.

[10] 9th DIMACS Implementation Challenge: Shortest Paths, Available online at :http://www.dis.uniroma1.it/challenge9, 2006.

[11] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[12] J. Sankaranarayanan and H. Samet, "Query processing using distance oracles for spatial networks," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 1158–1175, 2010.

[13] Yuan J., Zheng Y., Xie X, and Sun G. T-drive: Enhancing driving directions with taxi drivers intelligence. In *Transactions on Knowledge and Data Engineering* (TKDE '12).

[14] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. 2007. Adaptive fastest path computation on a road network: a traffic mining approach. In *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*.

[15] Vasilis Verroios, Konstantinos Kollias, Panos K. Chrysanthis, and Alex Delis. 2008. Adaptive navigation of vehicles in congested road networks. In *Proceedings of the 5th international conference on Pervasive services (ICPS '08)*. ACM, New York, NY, USA, 47-56.

[16] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. 2009. Monitoring path nearest neighbor in road networks. In Proceedings of the 2009 *ACM SIGMOD International Conference on Management of data* (SIGMOD '09)

[17] Boriboonsomsin, K.; Barth, M.J.; Weihua Zhu; Vu, A., "Eco-Routing Navigation System Based on Multisource Historical and Real-Time Traffic Information," *Intelligent Transportation Systems, IEEE Transactions* on , vol.13, no.4, pp.1694,1704, Dec. 2012

[18] Juan C. Herrera, Daniel B. Work, Ryan Herring, Xuegang (Jeff) Ban, Quinn Jacobson, Alexandre M. Bayen, Evaluation of traffic data obtained via GPS-enabled mobile phones: The Mobile Century field experiment, *Transportation Research Part C: Emerging Technologies*, Volume 18, Issue 4, August 2010, Pages 568-583, ISSN 0968-090X, http://dx.doi.org/10.1016/j.trc.2009.10.006.

[19] Krumm, John, A survey of computational location privacy, *Journal of Personal and Ubiquitous Computing*, V 13, N 6, 10.1007/s00779-008-0212-5, http://dx.doi.org/10.1007/s00779-008-0212-5, Springer-Verlag, 2009-08-01, P391-399

[20] James Middleton, Vendors Develop Opensource Framework for Internet of Things, in Business Cloud News, July, 2014, available online: http://www.businesscloudnews.com/2014/07/08/vendors-develop-open-source-framework-for-internet-of-things/

[21] John S. Otto and Fabián E. Bustamante. 2009. Distributed or centralized traffic advisory systems: the application's take. In *Proceedings of the 6th Annual IEEE communications society conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'09)*. IEEE Press, Piscataway, NJ, USA, 709-718.

[22] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.C. Herrera, A. Bayen, M. Annavaram, and Q. Jacobson, "Virtual trip lines for distributed privacy-preserving traffic monitoring," in *Proc. of ACM/USENIX MobiSys*, Breckenridge, CO, June 2008.

[23] Wu, Haiyan. "Research on the Data Storage and Access Model in Distributed Computing Environment." In *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, vol. 2, pp. 621-624. IEEE, 2008.

[24] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A distributed storage system for structured data." *ACM Transactions on Computer Systems (TOCS)* 26, no. 2 (2008): 4.

[25] Zhu, Hongzi, Yanmin Zhu, Minglu Li, and Lionel M. Ni. "HERO: online real-time vehicle tracking in Shanghai.", *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[26] Logenthiran, T., and Dipti Srinivasan. "Intelligent management of distributed storage elements in a smart grid.", *Power Electronics and Drive Systems (PEDS), 2011 IEEE Ninth International Conference on*, pp. 855-860. IEEE, 2011.

[27] Taliwal, Vikas, Daniel Jiang, Heiko Mangold, Chi Chen, and Raja Sengupta. "Empirical determination of channel characteristics for DSRC vehicle-to-vehicle communication.", *Proceedings of the 1st ACM intl workshop on Vehicular ad hoc networks*, pp. 88-88. ACM, 2004.

[28] Wang, Zhe, and Mahbub Hassan. "How much of dsrc is available for non-safety use?." In *Proceedings of the fifth ACM international workshop on VehiculAr Inter-NETworking*, pp. 23-29. ACM, 2008.

[29] Hull Bret, et. Al,"CarTel: a distributed mobile sensor computing system." In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pp. 125-138. ACM, 2006.

[30] An Overview of Distributing Data with Geodatabases, An ESRI Technical Paper, June 2007, available online http://downloads2.esri.com/support/whitepapers/other_/ArcGISServer_DistributingData_nov2007.pdf

[31] A. Santos, N. McGuckin, H.Y. Nakamoto, D. Gray, and S. Liss, Summary of Travel Trends: 2009 National Household Travel Survey, U.S. Department of Transportation Trends in travel behavior, 1969-2009 Federal Highway Administration, Washington, 2009.

[32] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (MSST '10). IEEE Computer Society, Washington, 2010.

[33] Aakriti Gupta, J. Lakshmi, S.K.Nandy, “Real Time Routing in Road Networks”, *4th IEEE International Conference on Big Data and Cloud Computing*, Dec. Sydney, Australia, 2014.

[34] Pavan Kumar Akulakrishna, J. Lakshmi, S.K.Nandy,“Efficient Storage of Big-Data for Real-Time GPS Applications”, *4th IEEE International Conference on Big Data and Cloud Computing*, Dec. Sydney, Australia, 2014

[35] De Zhou, Z., R. Valerdi, S. Zhou, and L. Wang. "Guest Editorial Special Section on IoT." *Industrial Informatics, IEEE Transactions on* 10, no. 2 (2014): 1413-1416.

[36] Flask. [Online]. Available: http://flask.pocoo.org/

J. Lakshmi received her doctoral degree in computer science from Indian Institute of Science in 2012. She is working as a Principal Research Scientist in the Department of Supercomputer Education and Research Centre (SERC) at the Indian Institute of Science (IISc) for more than 17 years. She is the convenor for the Cloud Systems Lab in SERC. Her areas of interest include systems software and architectures for virtualized and distributed systems. She has published research papers in the areas of QoS on virtualized servers, I/O virtualization architectures, fault tolerance in operating systems, etc. She is currently the convener for the Cloud Systems lab in SERC. She also manages and supports the HPC data center at SERC. She is a member of the IEEE.

S. K. Nandy is a Professor at the Supercomputer Education and Research Centre, and the Department of Electronic Systems Engineering of the Indian Institute of Science, Bangalore, India. Prof. Nandy obtained his B.Sc.(Hons.) degree in Physics from the Indian Institute of Technology, Kharagpur in 1977. He obtained his B.E.(Hons.) degree in Electrical Communications Engineering in 1980, M.Sc.(Engg.) degree in Computer Science and Engineering in 1987, and Ph.D degree in Computer Science and Engineering in 1989 from the Indian Institute of Science, Bangalore. He joined the faculty of Indian Institute of Science in 1982. He has to his credit several research publications in International Journals and proceedings of conferences and workshops. His research interests include SoC design methods, multicore and any-core SoC platforms, architectures and compilation techniques for high performance runtime reconfigurable massively parallel computing systems on silicon.

Aakriti Gupta completed her Masters in Computational Science from the department of Supercomputer Education and Research Centre at the Indian Institute of Science, Bangalore, India in June 2014. She is currently working as a system software engineer at Nvidia in Bangalore, India.

Pavan Kumar Akulakrishna, was born in Hyderabad, India in 1989. He received the B.Tech degree in Computer Science and Engineering from the Indian Institute of Technology, Patna in 2012 and M.Tech degree in Computational Science from the Indian Institute of Science, Bangalore, India in 2014. Prior to his master’s degree he has also done his internship at CDAC, Bangalore, on Network Monitoring and Security in the year 2011. He is currently working for Nvidia Graphics Pvt. Ltd., at Pune, Maharashtra, India as Systems Software Engineer, in the area of GPU architecture. His research interests include Big Data Storage and Applied Mathematics.

Vaibhav Ankush Kachore completed his B.Tech. (2011) in Electronics and Communication Engineering from National Institute of Technology, Nagpur, India. He has worked on research project of Indoor Positioning System (IPS) which was his B.Tech. thesis project. His research interests includes security, cloud systems, algorithms and machine learning. He completed his Masters in Computational Science from the department of Supercomputer Education and Research Centre at the Indian Institute of Science, Bangalore, India in June 2015. He is currently working for Nvidia Graphics Pvt. Ltd. as Systems Software Engineer, at Bangalore, India.