Enriching Driving Experience with Cloud Assistance

Paper #299, 12 pages

Abstract– This paper shows that driver safety technology such as collision detection can be built from sensors that are available on commodity smartphones and a cloud service that processes the data from nearby vehicles. Such technology would help since our analysis of accidents in the US shows that surprisingly many happen due to mundane reasons such as rear-ends, occur at low speeds, and are likely due to human fault. Our proposal is in contrast to high-cost offerings from car manufacturers that require laser, camera or radar equipment. Unlike systems that make vehicles communicate with nearby vehicles or roadside infrastructure, this proposal has fewer technical challenges and requires no further infrastructure deployment or standardization.

Our contributions include an experimental feasibility study of smartphone sensors (e.g., availability and accuracy of location and speed); the design of a cloud service to reason about future events involving one or more vehicles based on reports of vehicle state in the past; and the design of algorithms that compensate for missing or inaccurate sensor readings by combining information from multiple sensors, historical trip information and readings from other vehicles. To scale, the cloud service partitions work across many servers yet can respond in near real-time because the partitioning strategy ensures that all the data required to raise any alert is located on the same server with high probability.

1. INTRODUCTION

Safety is consistently ranked as the most important consideration in choosing a new car often beating style, luxury and convenience. A surprisingly large fraction of accidents are of the simple kind—our analysis of the National Highway Traffic Service Authority's dataset of accidents reveals that about 35% of the vehicles were involved in a rear-end, about 10% involved drivers suddenly departing the road, and another 9% involved drivers departing their lane to side-swipe vehicles in adjacent lanes. Considerably less frequent are impact with pedestrians and animals (4%), parked vehicles (2%) and head-on collisions (2%). A significan number of collisions occurred when at least one of the vehicle turned and hit another (20%) and when vehicles on intersecting paths at road junctions hit one another (8%). Further, most driving is mundane; one study estimates that over 80% of the work done during highway driving involves staying in the lane and at a safe distance from the vehicle in front.

To improve driver safety, it is enticing to use technology to help with such tasks that are simple but need to be done correctly over long periods. We are interested in building an early warning system that alerts drivers when they drift off their lane or off the road; and when there is another vehicle in their blind-spots; or are about to collide with another vehicle. Unfortunately today such *assistive technologies* are not widely available. The high cost and the pace of innovation has been glacial. We think there are two reasons for this Commercial attempts from car manufacturers are hindered by their multi-year refresh cycle and the difficulty inherent in building embedded systems that have to be reliable for as long as the lifetime of the car. Features such as lane departure warning [2], collision detection and blind-spot monitoring [5] are available, more commonly in high-end cars, at price points of several thousands of dollars. They use a combination of camera, laser and radar technology. Upgrading the software or hardware of such features is rarely easy and often impossible.

While the above commercial implementations are standalone, i.e., each vehicle fends for itself, most of the research comunity has focused on inter-vehicle (v2v) or vehicle-to-infrastructure (v2i) communications, with vehicle and roadside monitors warning drivers. In late 1999, the United States Federal Communications Commission (FCC) allocated 75 MHz of spectrum in the 5.9 GHz band for the so-called Dedicated Short-Range Communications (DSRC) to be used by Intelligent Transportation Systems (ITS). Despite over a decade of research, we are unaware of any sizable deployment of this technology for the purpose of driver safety. There are many technical hurdles that we see no easy solutions for; how to quickly establish and effectively use such ephemeral connections remains a hard technical problem. Further, deploying dedicated roadside infrastructure is expensive and unlikely to be widely available, and standardization across car manufacturers has been very slow.

In this paper we attempt to answer the following the question: *is it possible to build safety features for vehicles with just a smartphone and a service in the cloud?* If the answer is *yes*, we would have significantly lowered the barrier to bringing driver safety technology to the masses. Such a system would cut costs and provide value today without waiting for infrastructure deployment and standardization, would be future-prrof and would have a faster cycle for innovation.

We set out with a goal to develop assistive technology that warns drivers upon lane departure, impending collisions and vehicles in blind-spots. We required these warnings with a low false positive rate (i.e., few incorrect warnings) and a low false negative rate (i.e., miss few actual events). While it is no-doubt desirable that the warnings be 100% accurate and always available, we consider that not to be a show-stopper

metric for an assistive technology. We would like to deliver such technology to a large number of drivers, quickly, and at a low cost. Today, that means without having to rely on changes from car manufacturers, prolonged standardization efforts and no costly infrastructure changes. Energy consumption is a non-goal since most vehicles have power jacks.

Our basic idea is rather simple—we assume that every vehicle is equipped with a smartphone that is participating in our system. We know that this is a strong assumption and we will relax it later. Then we run an application on the smartphone to obtain location, speed, heading and related data from every vehicle. We ferry this information to a cloud service, which warns drivers by correlating the data from vehicles that are near each other on the road.

At first blush, this smartphone+cloud solution has some desirable aspects though open questions remain. Let's begin with the positives. Smartphones are relatively inexpensive, pervasive and have a wide array of sensors, such as the GPS, compass, camera and accelerometer. SmartPhone apps are easy to update and sensors improve with each new generation of hardware. Connectivity between the smartphone and the cloud continues to improve due to ongoing investments in cellular networks such as 5G.

Several open questions remain: (1) can we obtain precise estimates of vehicular location, speed and heading? (2) can we refresh these estimates quickly when vehicles move? (3) can we keep an always-available, low-latency connection between the cloud and the smartphone? (4) can we design a cloud service that scales adequately and delivers warnings in a timely manner? Much of this paper focuses on answering these questions. Our key contributions include:

- We describe a novel smartphone+cloud system for improving driver safety. We perform feasibility study of this system by examining its various aspects based on a sizable dataset collected from real drivers.
- We describe the architecture and design of the cloud service that makes our system possible. Our design supports a low-latency, fault-tolerant, highly scalable backend service for processing spatio-temporal data. We describe the solution to the primary challenge: for scalability, a partition strategy, since no one server can handle everything, but one that supports joint computation that are performed over vehicles that can potentially interact. Consequently, our partitioning strategy keeps such vehicle sets on the same server. We idenitfy several secondary challenges and describe solutions including making hand-offs simple; dealing with server failures; and dealing with hotspots–certain roads are more likely to be congested than others.
- We perform end-to-end evaluation of our system and demonstrate its feasibility.

2. FEASIBILITY ANALYSIS

We examine the feasibility of the proposed architecture with a set of measurements and data analysis.

# of accidents	264324
# of vehicles	461679
# of persons	658454
# timespan	5 years

Table 1: Summary statistics of the NHTSA accident dataset.

Type Of Accident		%Vehicles
Same traffic-way, same direction		**43.00%**
rear end	34.37%	
sideswipe	8.59%	
Turning (changing traffic-way)		**19.40%**
turn across path	9.82%	
turn into path	9.58%	
Single Driver		**15.68%**
roadside departure	9.08%	
impact: pedestrian/ animal	3.99%	
impact: parked vehicle	2.19%	
Intersecting Paths		**8.02%**
Same traffic-way, opposite direction		**2.45%**
sideswipe	1.56%	
head on	0.86%	
Other		**11.45%**
backing vehicle	2.06%	

Table 2: Breakdown of vehicles by accident type (NHTSA)

2.1 Characterizing automobile accidents

Since 1988, the National Highway Traffic Safety Administration (NHTSA) collects detailed information about automobile accidents based on reports from police across the US. We examined the data for the last five years. Summary statistics of the data are in Table 1. For each accident, for each vehicle, the dataset contains information about the speed of vehicles, location of the accident, nature of the accident, attributed causes, road and atmospheric conditions, and actions performed by all the relevant entities before and during the accident. To make the dataset size tractable, the NHTSA post-processes the data by grouping similar accidents (and vehicles) together.

Table 2 classifies the vehicles that are involved in accidents by the type of the accident. We see that a significant number are involved in rear-ends (35%). A collision detection system can help here: it is possible to build one given the locations, speeds and course of the involved vehicles. Drivers drifting off their lane to side-swipe cars in other lanes account for 9% of the vehicles in accidents and another 9% involve drivers departing the roadway. A lane departure warning system will help the latter and a blind-spot vehicle detector will help the former. It is possible to build a lane departure system based on frequent and precise estimates of the vehicle's location. The blind-spot detector has similar requirements. Another major cause involves collisions in intersections, when at least one of the vehicles is turning (19%) or when the paths otherwise intersect (8%). Warning for such intersection-related trouble could be done given information about the direction in which the vehicle is heading. Together these account for over 80% of the vehicles that are involved in accidents.

Among factors that contribute to accidents, about 11% of accidents in the dataset were due to distracted drivers who were using phones or other gadgets in the vehicle or were dis-

tracted by other people within the car or on the roadway. 5% of the accidents involved alcohol. The driver was impaired in 3% of the cases, i.e., drowsy, fatigued or ill. About 30% of the accidents happen outside of daylight hours and about 9% happen during rain or fog. In all of these cases, assistive technology that warns or raises an alert could improve safety.

Delivering usable assistive technology can save lives. 29% of the accidents lead to significant injury for at least one person; 8% involved injuries to multiple people. Many of the accidents happen in settings that are relatively easy for assistive technology to get right. 57% of accidents happen on straight roadways; it is easier to estimate vehicle trajectory along road segments rather than at intersections. Further of the accidents that reported vehicle speed at the time of accident, 58% had speeds less than 10mph; only 6% involved vehicles moving faster than 50mph; slower speeds impose less stringent latency constraints on assistive technology.

2.2 Suitability of the Smartphone

We are unaware of any dataset that helps verify whether smartphones can collect the information needed to help with driver safety. Hence, we built a smartphone app that observes various sensors on the smartphone and the network connectivity. Whenever the user is driving, the Gadfly app obtains readings from the GPS, accelerometer and gyroscope. We built the app for the iPhone; it works with iPhone 4 and above hardware and iOS version 4 and above software. The app configures a callback for whenever a new GPS reading is available and the user's location has changed by at least .5m. The latter check avoids duplicate entries when the vehicle is idle. The GPS on current iPhone hardware can be polled once each second. The Gadfly app also obtains accelerometer and gyroscope readings once every 50ms. We confirmed that these sensors can be sampled as fast as once every 10ms but chose the slower rate to reduce overhead. The app also monitors network connectivity. It has a persistent HTTP connection open to services in four Azure datacenters that are distributed across the US. Once every 100ms, the app initiates a GET request to one of the randomly chosen datacenters. Also once every 100ms, the app issues an ICMP ping to the first IP hop, which is likely the GGSN in the cellular provider's network [24], and a ping to the last IP hop that responds on the path to one of the four Azure datacenters, which is often the edge router at that datacenter. With this data, we would like to understand how well a smartphone can capture vehicular information such as location, speed, course and the nature of its connectivity to the cloud.

We deployed the Gadfly app to 8 volunteer users, logged 67 hours of driving time, mostly in cars, and covered 699 miles in 3 US states. The observed roadways are biased towards those that are commonly used by our volunteers– they cover urban and suburban areas including tunnels and bridges. The dataset does cover some interstate highways and a few roadways in rural areas. Volunteer's used 3 different hardware SKUs across 2 cellular providers.

What	Data Logged	Frequency	Device
Cabspotting [16]	GPS	10s	*custom receiver*
Seattle Cars [21]	GPS	5s	*custom receiver*
GeoLife [33]	GPS	*varying*	*varying*
TestMyNet	NetworkProbes	a few times	Smartphones; WinPhone 7.1
Gadfly app	GPS, Compass, Accelerometer, NetworkProbes	$\leq$1s	Smartphones; iPhone 4/4s/5

Table 3: Datasets analyzed to examine the suitability of smartphones for driver safety

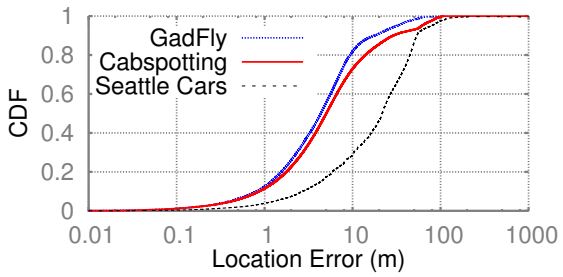


Figure 1: For each location reported by GPS, the CDF of the distance to the nearest point on a *road* as provided by major map sites.

We also analyze relevant datasets from earlier work that look at subsets of the relevant aspects (see Table 3). Three of the datasets capture the output of custom GPS receivers, carried by San Fransico area cabs (Cabspotting project [16]), volunteers in the Seattle area [21] and around Beijing (GeoLife project [33]). GeoLife also contains some data from smartphones. Each of these datasets has data from over 20 users and over 30 days per user. TestMyNet [27] performs a battery of tests of a smartphone's connectivity including the latency and bandwidth along the path from the phone to a microsoft datacenter. TestMyNet data has results from over 111K phones and 287K tests spanning all the major link types (HSPA, 3G, WiFi) except 4G/LTE.

2.2.1 *Location Accuracy*

For each GPS reading, i.e., lat/long coordinate, we use public APIs provided by mapping sites (e.g., Google, Bing) to identify the closest on-road location to that reading. The distance between these two points is a measure of the location inaccuracy [1]. Fig 1 plots the value of this metric for each of the datasets that have GPS readings. The x axes is on a log scale. These results use Bing maps; Google maps provided similar results but since it imposed strict rate limits on the number and frequency of queries, we could not fully analyze the datasets. We omit the results from GeoLife because both Bing and Google maps led to much larger error, perhaps because they have poor quality maps in and around Beijing where most of the GeoLife dataset was collected.

We see that the Seattle cars dataset has the worst accuracy,

[1]Throughout this paper, we use the haversine function to compute distances between geographic locations, which is more accurate than the spherical function for nearby points.

only 30% of the samples are within 10m off the nearest road. Suppose a system's turn-around time is 100ms. During that time, a vehicle traveling at 30mph would travel 1.33m. The typical width of a road lane is about 3.5m and since velocity perpendicular to the direction of the road is often much smaller, lateral motion that leads to lane drift or driving off the road occurs more slowly. Location errors above 10m are hard to correct but smaller errors can be corrected using the accelerometer and gyroscope or consecutive GPS readings. The Seattle cars dataset has large error because they use an inexpensive GPS device (Realtek RBT2300) which is known to be inaccurate [28]. The CabSpotting dataset uses a much better GPS device that the cabs also use for navigation. Here about 75% of samples are within 10m error.

Surprisingly, the Gadfly app shows that smartphones obtain better accuracy. This is perhaps because GPS technology has improved since 2009 when the cabspotting data was collected. Modern smartphones use state-of-the-art GPS chips, e.g., Qualcomm MDM 96xx and Broadcomm BCM 475x. Many of these are integrated chipsets that simultaneously support 4G/ 3G/ WiFi and GPS. This means that the GPS signals are captured on the primary antenna which is placed on the best possible location on the phone as opposed to the case of older phones which used standalone GPS chips whose antennas had to compete for the precious space.

We note a couple things. First, measuring the accuracy of GPS' location is hard, especially at scale and when the device is moving, due to the difficulty in obtaining ground truth. In micro-benchmarks, we manually synchronized the Gadfly app with readings from a high precision GPS device (uBlox [32]) during several test drives. We saw the above methodology to correctly estimate error for samples during the ride. But at the beginning of rides, wildly incorrect GPS readings did happen and some were mistakenly categorized as small error since they just happen to be close to *some* road. To correct for this, Figure 1 plots the CDF of error for only samples that were obtained after five minutes into the ride. The reason for more inaccuracy at the beginning of a trip is that the GPS chip needs time to discover visible satellites and to calibrate. Ideas to get a quicker first fix exist, e.g., A-GPS but there is still room for improvement. This also means that for most of the ride the GPS on a smartphone is quite accurate–80th percentile error is less than 10m. Second, not using the GPS device and instead relying on known locations of WiFi access points and cellular towers leads to higher error [31, 30]. Finally, we believe that leveraging GPS for location is future-proof due to ongoing investments in infrastructure and technology by the Federal Aviation Administration (FAA), the Coast Guard (CG), the US military and other countries.

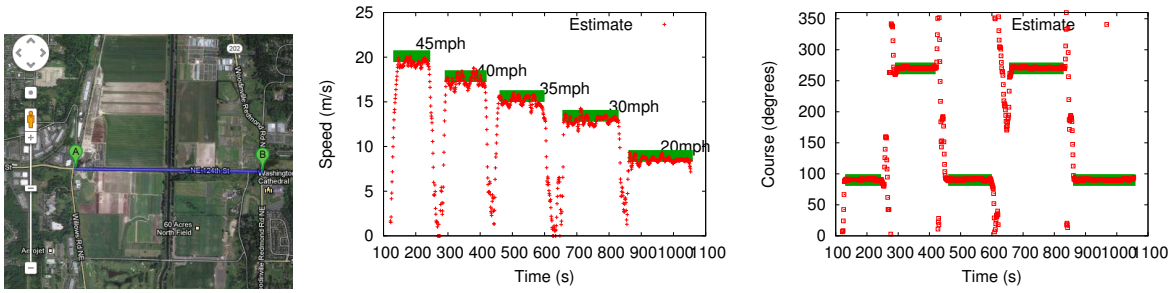
Accuracies of up to 10cm have been demonstrated by processing the raw GPS signals from precision chipsets in conjunction with augmentation signals (WAAS, RTK) [3]. A bit of background to explain. Commercially available GPS signals in the US were intentionally degraded until 2009, a technique called Selective Availability, so that only military receivers with access to an encrypted signal could get good accuracy. To offset this, many commercial entities including the FAA and CG developed Differential GPS (D-GPS). In D-GPS, physical locations broadcast the difference between their known location and that estimated by GPS. Whenever this difference changes slowly and remains the same over a large area, correcting observed GPS using this difference signal improves accuracy. Many sources of error have this property including the noise to degrade GPS, transmission delays in the ionosphere, clock drift in satellites and ephemeris (i.e., satellite location) prediction errors. FAA and CG operate a Wide Area Augmentation System (WAAS) which routes differencing signals from several locations to one of a few master stations that refine the differences and relay to receivers through a collection of satellites. Another form of correction, known as RTK [3], is based on the phase of the GPS signal's carrier wave which is over three orders of magnitude higher frequency than the GPS content, so locking on to the carrier can improve accuracy. Today, several states have deployed RTK stations. A few other satellite constellations are coming online, e.g., GLONASS from Russia and Galileo from Eurozone and chipset manufacturers argue that accuracy will improve by using multiple constellations.

2.2.2 *Accuracy of Speed, Course*

The Gadfly app logs speed and course estimates of the vehicle, which are computed by the GPS chipset based on the doppler shift of signals from the GPS satellites.

Figure 2 depicts a controlled experiment wherein we picked a speed value and traversed a road segment at that speed. To keep the experiment simple, we picked points that lie along an east-to-west line as shown in Fig. 2(a). We drove from point A to B and back. Both points have traffic lights but there were no other stops in between. We turned around after passing the points. The experiment was performed at night, so traffic was light. We expect the driver error in holding a desired speed value to be $\pm$2mph. Figure 2(b) (and 2(c)) depicts the speed (and course) goal with green solid lines and the speed (course) estimate from the phone with points. We see the speed estimate is quite accurate, the average error is 1.14 mph. Acceleration and deceleration episodes are clearly visible. Higher speeds do not lead to larger errors. Course is computed in degrees counted clockwise with due North being 0°. We see that the course estimates are close to ground truth, 90° when driving from A to B and 270° when going the other way. The average error is 1.4°. Accurate speed and course information allow Gadfly to effectively extrapolate the path taken by a vehicle.

Over the entire dataset collected by the Gadfly app, Figure 3 plots a CDF of the error in the speed and course estimates. It is hard to obtain ground truth information for speed and course, so we compare the estimates from the GPS chipset with those derived from the path between successive samples. We make the simplifying assumption that the locations are accurate and that the vehicle travels along the shortest



(a) Driving route (b) Estimate of actual speed vs. Driver's Goal (c) Estimate of Course vs. Ground Truth

Figure 2: Controlled experiment to verify smartphone's speed, course and acceleration estimates.

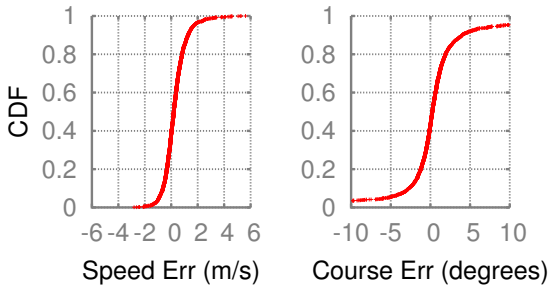


Figure 3: Comparing the speed and course estimates from GPS with a simple baseline.

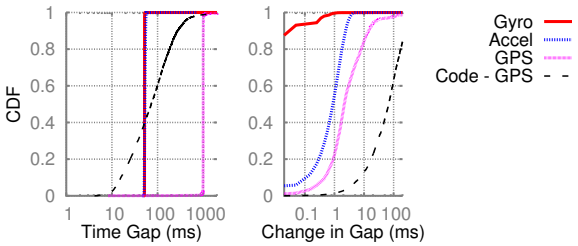


Figure 4: Interarrival time between successive samples from various sensors and the change in interarrival time

geographic distance between successive points at a uniform speed. The figure indicates that the speed is within ± 2m/s almost always and that the course error is within $\pm 5^\circ$ over 90% of the time perhaps because of turns. Together with the above micro-benchmark this leads us to believe that the speed and course estimates are highly accurate.

2.2.3 *Misc: Frequency, Lag, Energy draw*

How often can we sample smartphone sensors and can we do so in a predictable manner? Ideally, the sampling rate should be fast enough to keep up with vehicular motion and have less jitter. Fig 4 plots a CDF of the gap between successive samples from various sensors and the change in gap.

The figure shows that the Gadfly app gets accelerometer and gyroscope readings every 50ms (spike at 50ms on the Figure 4, left). This is expected since the app polls these sensors at 20Hz. We see that these readings arrive quite predictably, the jitter between readings is less than 2ms almost always for both (see Fig. 4, right).

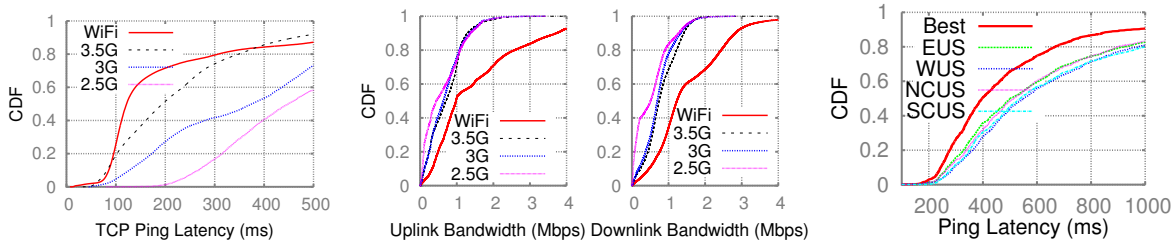
Similarly, the figure shows that GPS can be sampled at 1Hz (spike at 1000ms) and is quite predictable– the change in gap between samples is rarely above 10ms. For vehicular motion, 1Hz is not ideal. GPS chipsets that provide samples at up to 5Hz exist and could find their way into future smartphones [6]. Gadfly uses well known techniques to interpolate based on accelerometer and gyroscope readings as well as information about roadways and historical patterns of of the user (§3.3). Recall that most accidents happen at low speeds (§2.1) which makes interpolation easier.

When the phone is under load or due to poor support from the smartphone OS, there may be a large lag before the hardware readings are available in software. Fig 4 left plots the gap between when the GPS hardware obtains an estimate and when the Gadfly app receives it in software (Code-GPS). This lag is similar for the other sensors. The figure shows that GPS readings are available in software within 100ms only about 60% of the time and the jitter is quite large. We are not yet sure of why this happens; the phones were not plugged in, so this could be an artifact from voltage scaling to save energy or the phone software may not be able to keep up with high rates of events.

The Gadfly app logs the remaining battery level every 10s. Our dataset shows that, with Gadfly app running, the phones consumed energy at the average rate of 1.5W. We did not constrain the volunteers in any way except to run the Gadfly app; hence this energy draw includes Gadfly and any other apps that the user may have running during the drive. iPhone's have a typical battery capacity of 1500mAh or 5.5Wh which means that Gadfly can run for 3.5 hours of driving before charging. We believe an energy optimized version of Gadfly can do much better. However, phone chargers for vehicles have become inexpensive (< 100\$) and are widely available. We defer considerations of energy efficiency to future work.

2.2.4 *Network Connectivity*

TestMyNet is an app for windows phone 7.1 which reports the average two-way latency for a TCP ping from the phone to several popular web-sites. We analyzed the results from over $287K$ experiments and $111K$ unique phones. Fig-

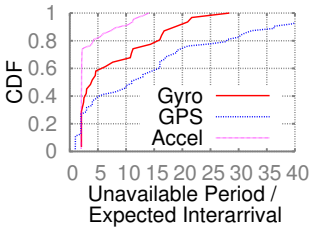


(a) Round trip latency by connectivity type (b) Up- and Down-link Available Bandwidth by connectivity type (c) For 3G/ AT&T, latency to several datacenters in the US

Figure 5: Understanding the latency of the path between smartphones and the cloud (an Azure service)

GPS	.93%
Gyro	.02%
Accel	.04%

(a) Fraction of time each sensor is unavailable



(b) CDF of the duration of unavailability

Figure 6: Examining the frequency/duration that sensors are unavailable.

ure 5(a) shows the latency for each connectivity type. We see that both 2.5G (e.g., EDGE, 1xRTT, GPRS) and 3G (e.g., EVDV, EVDO) technologies result in poor latencies. 3.5G technologies (e.g., HSPA) and WiFi are pretty good; substantial fraction of the experiments show an average latency below 100ms. While this dataset only had a few samples from 4G (e.g., LTE), our micro-experiments with the Gadfly app and those from prior work indicate 4G to be even better; average latency of 60ms and low variability [19].

Gadfly's bandwidth demands are quite small. Each reading from the GPS, accelerometer, and gyroscope is 40B. To upload raw samples every 50ms, Gadfly would need an uplink bandwidth of 19.2Kbps. Fig 5(b) shows that much more bandwidth is available on the phones in the TestMyNet dataset. If users drive 2hours every day, their bandwidth usage would be 518MB per month. To reduce the amount of data uploaded, Gadfly uses a few simple optimizations; it uploads raw samples only when the current sample is different from what would be predicted based on data the cloud already has.

Cloud providers have geographically distributed data centers. So, would picking an appropriate site reduce network latency? To answer this question, we set up a service at four Azure datacenters and had the Gadfly app ping a random site every 100ms. Figure 5(c) shows the CDFs of latency to each site for a user with 3G access to AT&T. Most of the data was collected on the west coast but a sizable amount was collected in the mid-west. Surprisingly, we find very little difference across sites. Probably because the shared portion of the path that lies in the cellular provider contributes most of the latency. We also find substantial variability at short time scales. The figure plots a CDF of the best latency observed across sites in each second. This value is often over 100ms smaller, and each destination has a roughly equal probability of contributing the best sample. We conclude that appropriate site selection is unlikely to lower path latency.

2.2.5 *Availability of sensors and network*

We examine how often the smartphone could potentially become unavailable for safety alerts. This could happen due to missing readings from any sensors (GPS, accelerometer or gyroscope) or lack of network connectivity. From traces, we say that a sensor is unavailable whenever the gap between successive readings is larger than twice the expected inter-arrival time. Additionally, we consider GPS to be unavailable when it cannot acquire a location, speed or course fix. Fig 6 (left) shows the fraction of time that sensors are unavailable for. Fig 6 (right) plots a CDF of the length of such gaps. We see that most sensors remain available for extended periods of time. When they do go unavailable, the sensors often are back within 5x their frequency, i.e., within $5 \times 1s$ for GPS and within $5 \times 20ms$ for accelerometer and gyroscope.

We note the several concerns around sensor availability. Besides the delay to get the first fix (§2.2.1), GPS is known to be unavailable or have high error in the so-called urban canyons, streets with high rise buildings, and in tunnels if the device is unable to *see* four satellites. The backhaul into the cloud may be congested or suffer dead spots in coverage, e.g., in rural areas and rush hours. The Gadfly app and other sensor hardware (accelerometer, gyro) on the phone may misbehave due to bugs or other faults. Inspite of these potential problems, our dataset collected from many drivers and across many road segments shows that the overall availability numbers are quite high. Gadfly only intends to assist drivers and not to replace them. So a sizable availability could still deliver value.

That said, there is active work in both research and industry that mitigates these concerns. Smartphone sensor hardware has become a major market leading to significant improvements in reliablity. There are two major leads for better GPS. First, *dead reckoning* techniques calibrate accelerometer and gyroscope when GPS readings are available and use them to extrapolate when the GPS is not. Some devices plug into

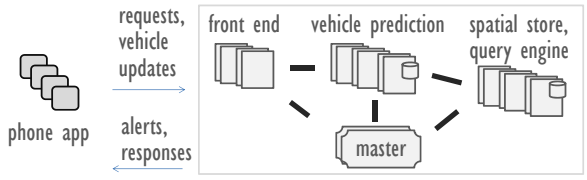


Figure 7: Gadfly **architecture**

the car to capture more accurate motion information (speed, acceleration, braking) [32]. Second, recent research shows that higher sensitivity GPS chipsets can make sense of the weaker GPS signals that bounce off buildings [26].

2.3 Summary of findings

Based on our experimental findings we are cautiously optimistic that smartphone sensors backed by a cloud service could, in the near future, be useful for driver safety. The chief impediments today are the low frequency of GPS readings–1Hz, the need for better accuracy in location–80th percentile is 10m today while we would like it to be about 1m, and the need for a low latency path into the cloud–4G (e.g., LTE) and 3.5G (e.g., HSPA) appear usable but 3G does not. However, we note that technology is heading in the right direction on each of these fronts. GPS accuracy up to 10cm is achievable by combining raw GPS values with differential GPS and RTK signals [3]. Dead reckoning techniques to effectively extrapolate location are becoming available [32]. And, all major providers are investing in 5G. The rest of the paper addresses some of these challenges.

3. Gadfly DESIGN

Figure 7 depicts an overview of Gadfly architecture. It consists of a smartphone application and a cloud service. The app periodically collects information from GPS and other sensors on the phone and sends them to the service. Combining this information across phones and other relevant info, the cloud service raises alerts and responds to queries from the phone.

The key components in Gadfly's cloud service are a vehicle prediction layer, a spatial store and a query engine. Each installation of the Gadfly app has a unique ID. Gadfly uses this ID to identify the vehicle that the phone is in. Frontend servers hash this ID and forward the updates and requests from that phone to the server in the vehicle prediction layer that is responsible for this ID. Servers in this layer predict the future state of the vehicle between now and when the next update from this vehicle is expected. The predicted trajectory is stored as a function of time (§3.3). The geographic region being monitored is divided into variable sized grids and each grid is assigned to one of the servers in the spatial store (§3.4). Upon computing the predicted trajectory of a vehicle, servers in the vehicle prediction layer forward the information to all the grids that the vehicle will pass through before its next update based on the prediction. As a result, the server corresponding to a grid is aware of all the vehicles that are currently in the grid or will be in the grid soon, i.e., before the next expected update from the vehicle. This information is kept in an in-memory data structure. The query engine executes on the same servers that comprise the spatial store. Gadfly executes queries that raise safety-related alerts periodically, e.g., once every 100ms (§3.5).

A master server orchestrates the Gadfly architecture. It monitors the load and failure status of servers in both the vehicle prediction and the spatial store layers. In response to overload or server failure, the master server can bring new servers online, can change the hash function that maps phone IDs to servers in the vehicle prediction layer and, can adapt the mapping of grids to servers in the spatial store. (§3.6).

3.1 Requirements and how Gadfly meets them

Here we summarize the requirements for a cloud service focused towards helping drivers. We also describe Gadfly's key ideas that help it meet those requirements.

- Need high throughput for both updates and queries: Up to O(10^5) cars per metropolitan area, updates per car once every 100ms and a similar frequency of alerts requires a cumulative update and query throughput of up to O(10^6) per second.

Gadfly leverages the fact that the coupling between data items is sparse and structured: to assist a driver, Gadfly only needs to process updates from *nearby* vehicles. For high throughput, Gadfly parallelizes all of its components, the vehicle prediction layer is indexed by app ID whereas the spatial store is indexed by grids. More complex representations for spatial data exist, e.g., R-trees and others [18, 14]. However, they cannot handle a high volume of writes at different locations since they adjust the boundaries/ regions of the spatial index when new data arrives.

- Latency requirements: to be useful for driver safety, the system has to respond at driver timescales, say about 100ms. We expect the network latency to dominate and hence, aim to limit the cloud service's latency to 50ms.

For low latency, Gadfly's spatial store keeps records in-memory. A typical spatial index would execute queries per-vehicle, e.g., are there any vehicles that will collide with a given vehicle? Gadfly's query engine executes queries per grid, e.g., will any vehicles collide in this grid in the next 100ms? Since there are many fewer grids compared to the number of vehicles and collisions or other alerting events are rare, per-grid querying is faster; there are fewer queries to execute and no duplication of work. Further, whereas queries for items near-a-vehicle can require data items that reside just across the boundary in another grid, changing the scope of queries to be per-grid allows Gadfly to not worry about such items. Hence, Gadfly's queries are truly parallel, all the data required to execute a per-grid query lies within the server responsible for that grid. Queries that touch only one server would not encounter potential contention on the network or at other servers and can finish faster.

- Continuous change in the data items and also of the set of other items that an item is coupled with: for any vehicle, the cloud service only knows its state (location, speed and, course) at some time in the recent past when the app generated an update. To be relevant, an alert should be based on the current and future locations of this vehicle and that of other vehicles that are or will be in its vicinity.

Gadfly has a vehicle prediction layer that uses sensor readings from the vehicle (speed, course, acceleration, rotation) and supporting information such as the user's route history, estimates of traffic on road segment and roadway information to predict the trajectory of the vehicle (§3.3).

How to execute queries over trajectories? The predicted trajectory for a vehicle is a function of time. The per-grid queries that Gadfly has to execute are functions over multiple vehicles and time. For e.g., checking for collisions in a grid translates to $\exists$ vehicles v_1, v_2, time t^* $s.t.$ $location(v_1, t^*) \sim location(v_2, t^*)$. Naively storing the predicted trajectories in discretized form would increase both the storage space needed and the time to compute the query by a linear amount. For e.g., if the predicted trajectory values are computed at each 10ms epoch, then checking for collisions in the next 100ms requires $10X$ more space, $10X$ more computation since the query has to run at each epoch and is likely to be imprecise due to discretization error. Instead, Gadfly uses continuous math to perform the queries directly upon the predicted trajectory functions (§3.5).

- Robustness to failures and load hotspots: we would like driving alerts to be available in spite of server failures and load hotspots on roads due to congestion, accidents, construction or busy intersections.

Gadfly's master server(s) are responsible for monitoring and adapting the architecture in response to load changes and faults (§3.6). In particular, Gadfly's spatial structure allows a grid to be divided when there are too many vehicles in that grid without having to move a lot of data or having to create a lot of un-necessary grids (§3.4).

- As a secondary goal, we would like the system to support queries on arbitrary, much larger location ranges (e.g., accidents, disabled vehicles or congestion further ahead).

Gadfly's spatial store serves as a sieve to other data stores that are geared towards lower update and query rates but can persist data and serve arbitrary queries.

3.2 Smartphone Application

We begin with Gadfly's phone app which collects GPS, accelerometer and gyroscope readings. On the iPhone, the app uses CLLocationManager for GPS and CMMotionManager for the rest. For both cases the app registers callbacks at the timescales shown in Table 4. Current phone hardware

GPS	1Hz
Accelerometer/Gyroscope	20Hz
Phone↔Cloud	10Hz
Cloud: Standing query to raise alerts	10Hz
Cloud: Time budget to process query	50ms

Table 4: Timescales of various parts of Gadfly.

restricts GPS readings to 1Hz. While accelerometer and gyroscope readings could be sampled as quickly as 100Hz, we noticed that doing so worsens the lag before the phone software receives the reading from hardware, perhaps because the phone's timers become more variable at higher load. The app maintains a persistent connection open to the cloud service, to periodically send updates about vehicle state and to receive alerts if any. The app also performs simple redundancy elimination on the sensor readings to keep the messages short and save on data use. We described the quality of each of the sensors, the network connectivity and energy use in §2.2. In total, the app is about 1K lines of code with another 5K in shared libraries. We now move on to the cloud service.

3.3 Vehicle State Predictor

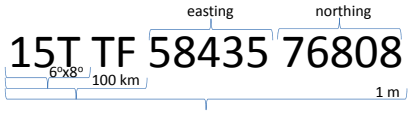
Based on the updates from the smartphone, Gadfly computes a predicted trajectory for the vehicle as follows:

$$location(t) = \begin{bmatrix} x \\ y \end{bmatrix} + \left(st + \frac{at^2}{2} \right) \begin{bmatrix} \sin(\theta + \gamma t) \\ \cos(\theta + \gamma t) \end{bmatrix}, \quad (1)$$

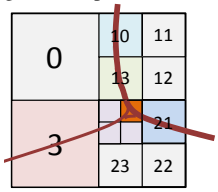
where x, y is the reported location of the vehicle, s is its speed, a is the acceleration, θ is the course and γ is the yaw, i.e., lateral change in course. To understand the equation, note that x corresponds to latitude, y to longitude, the course values count clockwise from due North and the yaw of the vehicle indicates the rate of change in its course.

Gadfly obtains location, speed and course from the phone's GPS reading; acceleration from the phone's accelerometer and yaw from the gyroscope. The location, speed and course of the phone are the same as that of the vehicle. However the accelerometer and gyroscope readings have the phone as their frame of reference and need to be transformed. For example if the driver holds the phone with the screen facing her, and points with the hand holding the phone towards where the vehicle is heading, then the along-road acceleration of the vehicle corresponds to the accelerometer reading along the phone's z axes. Gadfly uses calibration to do this correction. In theory, the calibration can be very hard, because whenever the phone moves relative to the vehicle the calibration has to be redone. In practice, without being given any specific direction, we found that drivers in our dataset kept the phone steady, either in the cup or sunglass holders or in their pockets for a significant majority of the observed driving time, so we find such calibration to be feasible.

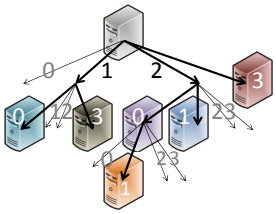
Gadfly compensates for errors in location, speed, and course by *map matching*. It uses road segment information from a large commercial vendor and from OpenStreetMaps to place the car on the most likely roadway based on current and previous readings. Unlike prior work which post processes the sensor readings after the drive finishes [31], Gadfly uses



(a) Gadfly uses the MGRS naming convention. This value identifies a 1×1m point near the airport in Omaha, NE. By dropping some of the numerals in the easting and northing, one could generate values that correspond to all of the larger sized spaces enclosing this point.



(b) An example of recursively partitioning space into grids. Here, at each level, the space is divided into four equal quadrants and labeled 0-3 from top left and counting clockwise.



(c) Gadfly divides grids that are highly loaded. It uses the longest prefix match on the grid label to determine the server responsible for that grid.

Figure 8: Illustrating Gadfly's spatial partitioning that balances load yet retains query efficiency.

similar techniques in real-time to place the vehicle on the correct roadway. Further, Gadfly uses prior trips from the same user when available and expected traffic patterns otherwise to predict whether the user would continue along the same roadway or which way she would turn. Whereas curvy roads can be handled by Eqn. 1 with an appropriate amount of yaw, Gadfly uses piece-wise function to model turns. Our results show Gadfly's predictions to be accurate.

3.4 Spatial Data Management for low latency

Gadfly uses spatial partitioning to divide work across servers. By keeping nearby data in-memory on the same server, Gadfly keeps queries local to a server thereby achieving low latency and allows many queries to run in parallel thereby achieving high throughput.

Spatial partitioning is trivial if the data items (vehicles) are distributed uniformly at random across the geographic area. In practice, however, vehicular density is highly skewed; most of the space has only a low density, while regions that overlap arterial roads or intersections have much greater density. The load in a region can also change during rush hours, construction and accidents. Dividing space, statically, into large regions will overload servers responsible for dense areas. Dividing into small regions avoids overload but, by creating many regions that have very little work, increases management overhead.

Gadfly divides space into square grids that have roughly even load; it recursively subdivides grids that have too much load and collapses grids with too little load. To do this efficiently, Gadfly should be able to uniquely identify geographic regions of varying sizes and be able to quickly determine which server is responsible for any location.

For naming, Gadfly uses the standard military grid reference system (MGRS); an example is shown in Fig. 8(a). The numeric suffix consists of two equal sized parts known as the *easting* and the *northing* (five numerals each in the example). The alphanumeric prefix, 15TTF in the example, uniquely identifies a 100×100km region on the surface of earth. Recursively, this region is divided into 10×10 smaller regions and a pair of numerals identify the east, north location of a particular smaller region. That is, in the above example, 15TTF57, 15TTF5876, 15TTF584768, represent the 10×10km, 1×1km and 100×100m regions containing the above point. MGRS lets Gadfly uniquely identify varying sized regions in a hierarchical manner.

To determine which server is responsible for a location, Gadfly uses the longest prefix match on the MGRS label of that location. An illustrative example is shown in Figure 8 b,c. On the left is a region with two roads; the thickness of the roads corresponds to average vehicle density. The figure shows how Gadfly might partition this space. The higher density of the thicker north-to-east road forced a $\frac{1}{16}$'th split of the space whereas the thinner road entering on the western border could be handled with only a $\frac{1}{4}$'th split and the busy interchange required a $\frac{1}{64}$'th split. The figure on the right shows a tree representation of how Gadfly maps locations to servers using longest prefix match. Each node in the tree has a server associated with it. There are four servers corresponding to each of the $\frac{1}{16}$'th sized grids that the thick road goes through and one each for the thin road and busy intersection. Grids that have not been expanded are illustrated with grayed edges. To lookup a label, start at the root and follow along the edges with characters in the label until you can go no further, thereby finding the server that is responsible for the smallest grid that contains this location.

Some features of Gadfly are worth noting. First, the time complexity of performing a longest prefix match is O(length of the label), which is logarithmic in the area but constant for all practical purposes (15 in the case of MGRS). Second, rather than run per-vehicle queries which become complex when the vehicle is near boundary, Gadfly runs per-grid queries. The prediction layer forwards vehicular information to all of the grids that the vehicle may pass through. Gadfly chooses 10×10m as the finest granularity grid. Since the phone app sends updates every 100ms, vehicles traveling slower than 100m/s or 223mph rarely pass through more than two grids between updates. Finally, longest prefix match allows a server to be responsible for any of the smaller regions within its region that are not dense enough to require their own server. This leads to a more compact division of work. In the above example, Gadfly has to assign servers for only 7 grids; many of the sparse regions (labels $0, 11, 12, 22, 23$) are handled by the root node.

Existing systems find it hard to support distributed spatial partitions. MongoDB [7], for example, supports spatial indexes on keys but will not shard (i.e., distribute) on such keys. This means that to gain the benefits of a spatial index one has to pay the cost of a spatial query having to touch all of the servers. Gadfly is similar in spirit to a quad-tree [17], but since it has to cope with potentially all of the land area on earth,

it does a 10×10 split at each level unlike quad-tree's 2×2 split. This keeps labels compact and reduces the lookup time. Further, quadtrees do not use longest prefix match to lookup. Hence, they have to assign all of the grids that are created to some server. In the above example, a quad-tree will have to manage 13 grids, many of which were created because of overload elsewhere in their region. This is almost double the number of grids that Gadfly has to manage. A few other spatial indexes, such as R-tree partition the space with rectangles that envelope data items. Doing so results in fewer partitions and can avoid partitioning in the middle of a dense cluster of items [18]. Such schemes cannot cope with high frequency of updates since they recalculate appropriate partitions whenever data items arrive or leave. Gadfly presents an effective way to manage and query distributed spatial partitions.

3.5 Supporting queries on continuous data

Gadfly executes queries per-grid that perform continuous math on the predicted location of vehicles. For example, checking for colissions in a grid translates to:

$$\exists \text{ vehicles } v_1, v_2, \text{time } t^*,\ s.t.\ L_{v_1}(t^*) - L_{v_2}(t^*) < \epsilon. \quad (2)$$

Here L_v is the location function from Eqn. 1, ϵ is some small distance value and the $-$ operation computes the euclidean distance between the two locations With this equation, Gadfly checks whether two vehicles in the grid come very close to each other at some time. The corresponding check for lane (or roadside) departures is:

$$\exists \text{ vehicle } v, \text{time } t^*\ s.t.\ \min\left(L_v(t^*) - \text{left edge}\right) > d\ \wedge \\ \min\left(L_v(t^*) - \text{right edge}\right) > d, \quad (3)$$

where the edges of the lane (or road) are represented as curves and d is the maximum amount of acceptable drift over the edge. This equation checks that the shortest distance between the vehicle and both the edges of the lane (or road) are above d which would only happen when the vehicle has drifted off one of the edges.

Gadfly solves these inequalities as follows. Eqn 2 wants the euclidean distance between two vehicle locations to be smaller than ϵ. This would only happen if both $|x_{v_1} - x_{v_2}|$ and $|y_{v_1} - y_{v_2}|$ are smaller than ϵ; here x_v, y_v represent the x and y coordinates of location L_v. Notice from Eqn 1 that, if the yaw (γ) is small, then both the x and y components of the location are 2^{nd} degree polynomials over the time variable t. Hence the difference between two values of x (or y) has the same degree and checking that its value is small can be done by quadratic factorization. Equation 3 can also be solved in a similar manner. When the yaw is large, we use the taylor approximation for cos and sin which increases the degree of the polynomial but is still solvable. In this way, Gadfly can check whether the differences in distance are small at any time before the next update from these vehicles (100ms) with only a few numeric operations.

3.6 Maintaining the architecture

The master role is performed by a small number of servers, in a Paxos ring, that adapt the Gadfly architecture to failures and load. The master controls the mapping from grids to servers via the label lookup tree that it pushes to all servers. It also determines how vehicle IDs are mapped to servers at the prediction layer through a hash function that maps vehicles to *buckets*, which rarely changes, and a function that assigns buckets to servers. Gadfly's servers exchange heartbeats with the master once every 100ms. Three consecutive missed heartbeats are treated as a sign of server failure. When a server in the spatial store (or the prediction layer) fails, the master assigns its grids (or buckets) to other servers by pushing an updated label lookup tree (or bucket to server map). Content in the spatial store is not replicated since a new update will arrive within 100ms from the phone app. However, the vehicle prediction layer has state collected over a longer duration for the vehicle. Each bucket is also assigned a backup server, and vehicle state is checkpointed once every 10s to the backup server. Data since the last checkpoint is retrieved from the phone app. The expected period of unavailability upon single server failure is about 500ms, which is okay for an assitive technology. Overload is more common and Gadfly handles it without downtime by treating overload as a non-fatal failure; the lookup tree (or bucket to server map) are changed as in the case of a failure but the identity of the previously responsible server is retained for a short while after the change to facilitate access to past data.

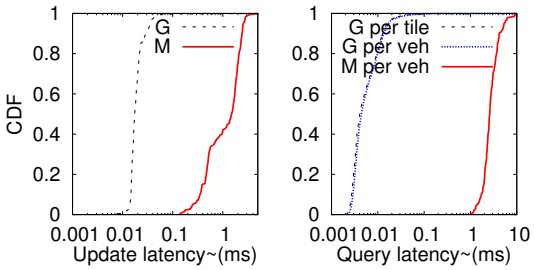


Figure 9: Per item update/ query latency

4. EVALUATION

Our experiments show that Gadfly system can scale to the latency and throughput requirements for driver safety. Comparison with a state-of-the art distributed spatial store (MongoDB) shows that for the same workload and using the same servers, Gadfly achieves between two and three orders of magnitude lower latency for processing updates and queries.

4.1 Methodology

Gadfly **Prototype:** We built a prototype of Gadfly and deployed it as a service on Azure. We also deployed the prototype on servers in the testbed to stress-test it. Unless otherwise specified the experiments here use four servers (or VMs) each for the vehicle prediction layer and the spatial store. Most of the functionality is implemented in C#, so there is room for improvement by switching to native code. We used the SignalR library [11] to maintain persistent connection between the app and the cloud service. It supports punching

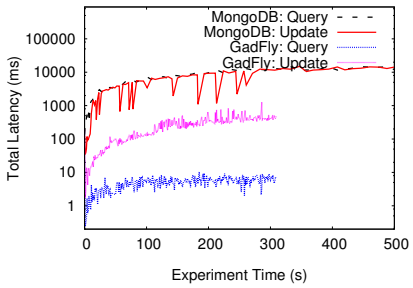


Figure 10: Time lapse of experiment

holes through NATs. We used ZeroMQ [12] for communication between the various layers in the cloud service. In all, the cloud service has over 6K lines of C# code.

Roadways: To stress-test Gadfly, we obtained roadway information of two large metropolitan areas from OpenStreetMaps [9], one 25x25 and the other 25x40 miles in size. OpenStreetMaps is cloud sourced, i.e., depends on volunteers, and by comparing with maps from a major commercial vendor we realize that it misses a small number of roads. We chose to use them because they have no usage restrictions.

Traffic Modeling: Our deployment of the Gadfly app allowed us to measure the quality of the smartphone sensors but 8 users is not enough to stress Gadfly's cloud service. Hence we rely on a widely used and sophisticated traffic emulator SUMO [15]. We pass the above roadways to SUMO and have it generate traffic. SUMO mimics vehicles moving along these roadways with varying speeds and different driver behavior. It also mimics activity at traffic lights and congestion induced slow-down.

Accidents: We extended SUMO to create collisions between vehicles. The extension is simple; it randomly picks vehicles and imparts to them acceleration that is much larger than warranted based on the roadway and the traffic surrounding the car. This leads to rear-ends, which our analysis showed to be the most common accident type in the US. We have not yet evaluated with other accident types such drifting off lanes, vehicles in blind-spot and aggressive driving.

Compared alternatives: We present results from Gadfly and an idealized version of Gadfly that assumes perfect prediction of vehicle trajectory. To compare, we implement the same queries over MongoDB. MongoDB is a leading no-SQL datastore with support for spatial indexes, which is supposedly used in production at FourSquare [1].

MongoDB configuration: We configure MongoDB to maintain a spatial index on the vehicle's location. This means that vehicles with nearby locations can be accessed more quickly than others. MongoDB lacks explicit support for data that is updated frequently and cannot store items that are functions over time (the predicted vehicle trajectory). Hence, with each vehicle's record, we attach its predicted trajectory as a list of {time, location} pairs. MongoDB supports sharding, i.e., distributing disjoint portions of the dataset to multiple servers. However, it is known to not support sharding over the key that has a spatial index [8]. We configure MongoDB to use the same number of servers as Gadfly and distribute data (i.e., shard) on vehicle ID. This means that a spatial query will fetch data stored in more than one shard. Finally, to identify collisions, we issue a periodic query that for each vehicle v_1, obtains all vehicles whose known location is close enough to the trajectory of v_1, that they may be able to collide with v_1 if moving at speed. This is a spatial query. Specifying constraints on the speed or trajectory of other vehicles would not use the spatial index, which is built only on node location. The results are post-processed to remove vehicles whose trajectory rules out a potential intersection with v_1.

4.2 Results

We performed a simple experiment that mimics a prototypical use-case of Gadfly. We generated five minutes worth of traces from SUMO, for the 25×40Km map of Seattle, which had 20K cars moving through the map. We sent updates from each car every 100ms to the store. The update carried the current location of the vehicle and enough metadata to support trajectory prediction. We found that MongoDB simply could not keep up with this rate of updates. So, to get some reasonable numbers we dialed down the update frequency to once every 1s, which reduced the aggregate update rate from 200Kps to 20Kps. In both MongoDB and Gadfly, a standing query ran every 100ms to check for collisions.

Fig 9 shows the time to update the store and to query the store for collisions normalized by the number of vehicles. The prediction layer batches updates that arrive close to each other before sending them to Gadfly's spatial store. We see that for Gadfly, amortized per-vehicle update latency and collision query latency is under 20μs over 95% of the time. The corresponding values for MongoDB are about 2ms for updates and over 6ms for queries. Since each query in MongoDB touches all shards and fetches all vehicles that could potentially intersect a vehicle, we would expect the query latency to be high. However, we were surprised by the high update latency. It appears that MongoDB's implementation of the spatial store cannot handle high update rate, perhaps because they use a data structure similar to R trees which needs rebalancing whenever data moves.

Figure 10 shows the total time to finish the collision query vs. experiment time. At the beginning of the experiment, there are only a few vehicles but soon the number of vehicles reaches steady state. The y axes is on log scale. The figure shows that while Gadfly and MongoDB are comparable early on, when the number of vehicles is small, MongoDB scales very poorly. In fact, the experiment generates load in a closed loop, that is no new updates are sent to the system until the earlier updates (and queries) are processed. This lets MongoDB drag on; it is able to crunch through five minutes worth of load only after 4970s. In practice, traffic will be generated in an open loop, that is updates and queries would be generated independent of whether the service can keep up.

Make	Features	Cost	
		Base Price	Feature Cost
Subaru Outback	1, 4	30K	4K
Acura MDX	4, 5	43K	9.5K
Toyota Camry	5	26K	5.5K
Mercedes C250	2, 5	35K	6K
BMW 328i	1, 3, 5	40K	5K
Volvo S60	1, 3, 4, 5	33K	3K

1=Lane Departure Warning, 2=Lane Keeping, 3=Collission Detection, 4=Auto Braking, 5=Blind Spot Monitor, 6=Back-Up Sensors

Table 5: Survey of the availability and cost of safety-related features in new cars.

Hence, we believe that MongoDB in practice would drop a substantial fraction of updates and queries and could even lead to server livelock due to thrashing.

Gadfly easily handles this level of load within the 50ms budget allocated for querying. Here we used four servers for 20K vehicles. We expect Gadfly's resource requirements to scale linearly with the number of vehicles.

5. DISCUSSION

With this paper, our goal is to present a feasible alternative architecture for driver assistive technology. We acknowledge the privacy concerns associated with sharing user location and driving characteristics. Delivering similar functionality in a privacy-preserving manner remains an open problem. However, we are aware of two instances of driver monitoring systems that are seeing widespread adoption. First, Progressive insurance offers discounts on auto insurance to drivers who install their Snapshot monitoring system [10]. Progressive reports that almost one million users have signed up. Specifics on what data is collected, where it is stored and who owns the data are unclear. But, presumably, the insurance company benefits from improved risk assessment of its drivers while users are incented both by the upfront discount and the likelihood of lower premiums when identified as safer drivers. Second, systems such as Google Latitude collect GPS readings from the phone once every few minutes and offer users a visual perspective of their drives and a breakdown of how their day is spent. The data is stored on Google's servers. It is estimated that Google Latitude has 10 million users [4]. Together, these systems give us hope that drivers may be willing to trade-off privacy for functionality.

6. RELATED WORK

Table 5 presents a survey of the safety related features currently offered by auto manufacturers. We picked a few popular car makes and in each case searched for the lowest price model that had some safety-related features. The table presents the base price for the car model and the additional cost of the safety technology. We see that most manufacturers offer a subset of the desired safety features as *packages* which increase the base price of the car by between 9% to 22%. The technology behind the features involved one or more cameras, lasers and radars mounted on the car and on-board software that combined information from these sensors. To the best of our knowledge, Gadfly is the first system to deliver similar functionality by using GPS and other sensors on the smartphone.

MARVEL [23] equips cars with four radios to enable relative localization of vehicles using antenna diversity. CarSpeak [22] detects humans entering the road from a hidden sidewalk by allowing the vehicle to communicate with roadside infrastructure over WiFi. They design an improved MAC and a content dissemination architecture. In comparison, Gadfly requires no new infrastructure, does not have to innovate at the link layer and requires no standardization across manufacturers. CTrack [31] uses roadway information and hints from other sensors to accurately estimate the path taken by a vehicle. Balan et. al. [13] describe a service that uses GPS readings to estimate taxi fares. Zhou et. al. [34] describe a service that matches updates from users traveling on buses with users who want an estimate of the arrival time of the bus. Gadfly is similar to these systems in its use of GPS on the smartphone, however the services it delivers to enhance safety and other aspects of the driving experience are novel and require Gadfly to solve different challenges.

Database and graphics communities have considered spatial datastructures (e.g., octrees [25] and R-trees [18]). More recently work on supporting high update rate and queries on moving data [29] concludes that more sophisticated datastructures are unsuitable for high update rates induced by data movement. Gadfly reaches the same conclusion and improves in three significant ways. First, prior work focuses on nearest-neighbor queries whereas Gadfly supports a wider set of queries. It shows that per-grid queries are a better match for the driver safety use-case. Second, Gadfly supports much stricter requirements on latency, below 100ms, and high throughput. To do so, it builds a distributed spatial store with the guarantee that queries rarely touch more than one server. Prior work is mostly single server (except for MOIST [20]) and obtains query latency on the order of seconds. Third, Gadfly incorporates correcting sensor signals and prediction of trajectory as first class entities. It speeds up queries over predicted trajectory by solving continuous equations.

7. FINAL THOUGHTS

In this work, we explored the question of whether driver safety technology such as collision detection can be built from sensors that are available on commodity smartphones and a cloud service that processes the data from nearby vehicles. We surveyed existing sensors in smartphones for feasibility, and also built and designed a cloud service to reason about future events involving one or more vehicles based on reports of vehicle state in the past. We also designed algorithms that compensate for missing or inaccurate sensor readings by combining information from multiple sensors, historical trip information and readings from other vehicles. We hope this serves as the first step towards building a low-latency spatio-temporal system for driver safety technology.

References

[1] http://bit.ly/DkEXr.

[2] Ford Lane Assist. http://bit.ly/sQy2gk.

[3] GoGPS. http://www.gogps.org/.

[4] Google Latitude. http://tcrn.ch/eO1PBg.

[5] Jeep Blindspot Detection. http://bit.ly/VzTKJL.

[6] MaxqData. http://www.maxqdata.com/.

[7] MongoDB. http://www.mongodb.org/.

[8] MongoDB spatial index. http://bit.ly/eYlmSS.

[9] OpenStreetMap. http://www.openstreetmap.org/.

[10] Progressive Snapshot Monitoring. http://pgrs.in/zPyOyb.

[11] SignalR. http://signalr.net/.

[12] ZeroMQ. http://zeromq.org/.

[13] R. Balan, K. Nguyen, and L. Jiang. Real-time trip information service for a large taxi fleet. In *Mobisys*, pages 99–112. ACM, 2011.

[14] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, New York, NY, USA, 1990. ACM.

[15] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz. Sumo - simulation of urban mobility: An overview. In *SIMUL*, pages 71–84, 2011.

[16] M. P. et al. CRAWDAD data set epfl/mobility (v. 2009-02-24). Downloaded from http://crawdad.cs.dartmouth.edu/epfl/mobility, Feb. 2009.

[17] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[18] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, New York, NY, USA, 1984. ACM.

[19] J. Huang, F. Qian, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *MobiSys*, pages 225–238. ACM, 2012.

[20] J. Jiang, H. Bao, E. Chang, and Y. Li. Moist: a scalable and parallel moving object indexer with school tracking. *VLDB Endowment*, 5(12):1838–1849, 2012.

[21] J. Krumm and A. J. Brush. MSR gps privacy data set 2009. Retrieved from http://research.microsoft.com/ jckrumm/GPSData2009, 2009.

[22] S. Kumar, L. Shi, N. Ahmed, S. Gil, D. Katabi, and D. Rus. Carspeak: a content-centric network for autonomous driving. In *SIGCOMM*, pages 259–270. ACM, 2012.

[23] D. Li, T. Bansal, Z. Lu, and P. Sinha. Marvel: Multiple antenna based relative vehicle localizer. In *ACM Mobicom*, 2012.

[24] J. Manweiler, S. Agarwal, M. Zhang, R. Roy Choudhury, and P. Bahl. Switchboard: a matchmaking system for multiplayer mobile games. In *Mobisys*, pages 71–84. ACM, 2011.

[25] D. Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. In *Rensselaer Polytechnic Institute (Technical Report IPL-TR-80-111)*.

[26] O. Mezentsev, Y. Lu, G. Lachapelle, and R. Klukas. Vehicular navigation in urban canyons using a high sensitivity gps receiver augmented with a low cost rate gyro. In *Institute of Navigation GPS Conference*, volume 15, pages 263–369, 2002.

[27] MSR. Testmynet. http://research.microsoft.com/en-us/projects/testmynet/.

[28] RoyalTek. RBT2300. http://www.royaltek.com/index.php/rbt-2300.

[29] D. Sidlauskas, S. Saltenis, and C. S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *SIGMOD Conference*, pages 37–48, 2012.

[30] A. e. a. Thiagarajan. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *Sensys*, pages 85–98. ACM, 2009.

[31] A. e. a. Thiagarajan. Accurate, low-energy trajectory mapping for mobile devices. In *NSDI*, pages 20–20. USENIX Association, 2011.

[32] uBlox. uBlox. http://www.u-blox.com/.

[33] Y. Zheng, L. Zhang, X. Xie, and W. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, pages 791–800. ACM, 2009.

[34] P. Zhou, Y. Zheng, and M. Li. How long to wait?: predicting bus arrival time with mobile phone based participatory sensing. In *Mobisys*, pages 379–392, New York, NY, USA, 2012. ACM.