Prototype Implementation of a Scalable Real-Time Dynamic Carpooling and Ride-Sharing Application

Dejan Dimitrijević, Vladimir Dimitrieski, and Nemanja Nedić
University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000, Novi Sad, Serbia
E-mail: {dimitrijevic, dimitrieski, nemanja.nedic}@uns.ac.rs

Keywords: real-time, carpooling, ride-sharing

Received: September 29, 2014

Setting out to build a real-time carpool and ride-sharing solution, which would be able to attain a global user base and is initially designed as highly-scalable, this paper describes some of the selected designed concepts, distribution and cloud computing strategies needed to do so. Our selections were based on experiences of others with same or similar-purpose solutions which were developed in the previous decade. Some of these solutions were either outdated, mostly by leaving its users with a subpar user experiences as their user base grew, or outgrown by having limited client reach, leaving them available only to a small portion of mobile client devices or desktop browsers and users. This paper presents an implementation of a ridesharing application prototype that follows all of aforementioned strategies. The goal of this prototype is to show that it is possible to make very scalable and ubiquitous ridesharing application, which is able to successfully reach and serve a global user base.

Povzetek: Članek predstavlja nov prototipa aplikacije za delitev stroškov prevoza z avtomobilom.

1 Introduction

This paper is a follow-up to a paper presented at the FedCSIS'13 conference [1] which extends further upon it with some experimental data and even more prototype implementation details. The aforementioned paper explored some currently available positions concerning an implementation of any present and future carpooling and ride-sharing applications. These applications should be real-time, dynamic (more about meaning of dynamic in following sections) and scalable enough to reach a worldwide audience. The explored positions were chosen so that any such application could and should work without either omitting support for any future mobile platform or leaving its clients with deteriorating quality of service as it client base grows. Because of the prior of the two just previously noted requirements, we identified some novel client web technologies which are or will be available on all modern mobile platforms. Therefore, we felt confident that any real-time dynamic carpool and ride-sharing application built upon these technologies could be ubiquitous enough. By ubiquitous we mean making it available across both various mobile and desktop platforms, current and future ones. Also, because of the latter requirement, which was also explored and outlined in our previous paper, some effort was also put in identifying server side technologies capable of producing a low-cost development and maintenance real-time dynamic carpooling and ridesharing mobile and web application. Considering existing solution experiences, primary idea was to build a scalable solution that could start out small but allow easy growth later. This is to accommodate the fact that the ride-sharing industry has only recently started becoming globally interesting. However, carpooling formally appeared in the US in the mid-1970s, after the 1973 oil crisis [2]. At that time the rising costs of using a personal vehicle for transportation of only one passenger made it prudent to drive more than one person, usually co-workers commuting to and from same workplace, splitting transportation costs. However, the reduction of oil and gas costs in the 1980s and the breakdown of a typical 9AM to 5PM workday in the 1990s led to a spiral down trend in carpooling popularity. Federal government in the US tried to counter such a trend by incentivising carpooling drivers, growing the number of no-toll carpool lanes across many highways. Those lanes were also allowing for relief from ever growing traffic jams and gridlock, as the number of vehicles on the roads increases. In 2000 it exceeded 740 million globally [3] and projections are there will be over 2 billion motorized vehicles by 2030 [4]. Large number of vehicles creates many well-documented problems for urban areas, such as increased traffic, increased pollution, parking congestion, and the need for expensive infrastructure maintenance. To reduce all of those and personal transportation costs also, we set out to create a low-cost, ubiquitous and global audience capable real-time carpool and ride-sharing solution.

1.1 Problem

The expenses, both environmental and fiscal, of single occupancy vehicles can be reduced by utilizing more of the empty seats in personal transportation vehicles. Carpooling and ride-sharing targets those empty seats: taking additional vehicles off the road thus reducing traffic and pollution, whilst providing opportunities for social interaction. However, historically carpool scheduling often limited users to consistent schedules and fixed rider groups–carpooling to the same place at

the same time with a set person or a group of people. To make that problem worse, the leading problem concerns, given in a 2009 survey about why people don't carpool, were difficulty to organize carpools and the inconvenience of organization far in advance [5].

Due to aforementioned reasons, this paper proposes some main guidelines and cloud distribution strategies that we fell will bring best value for any future global carpool and ride-sharing solution.

The rest of the paper is organized as follows: Section 2 provides an overview of related work. In Section 3 we present overall design concepts and our objectives. Section 4 elaborates on our proof-of-concept prototype system implementation choices, with subsections focusing on several specifics. Section 5 concludes the paper and discusses our future work.

2 Related work

In this section we present existing ridesharing and carpool solutions and research. We have divided this section into two main parts: Subsection 2.1 reviews carpool and ride-sharing related solutions currently available and Subsection 2.2 surveys some of the literature and papers on the subject.

2.1 Current carpool and ride-sharing solutions

Entering carpool and ridesharing search terms in some of the largest mobile app store and internet search engines returns a great deal of mobile apps and internet websites offering either classic or dynamic carpooling and ride-sharing. By the notion of classic carpool mobile app or website, we denote applications within which users just schedule and advertise their plans for a trip well in advance. This is accomplished effectively through an, in essence, a searchable electronic bulletin board, seeking other users travelling in the same direction at the same time. Although some of those apps and websites, such as carpooling.com [6] and its mobile client apps have a large user bases, the static routing problems they help solve makes their usage fairly limited.

The inconvenience of having to search though large carpools or even smaller but fixed choice driver groups, hoping to amongst them find a pre-scheduled and advertised trip adequately consistent with one's schedule, makes classic carpooling apps or websites non-practical for relatively short and near-immediate on-the-go carpool and ride sharing trip plans. It is for that reason that even a solution presented on [6] and its large network of European subsidiary websites, added advanced time constrained search features. These features are used to "find a lift", which to a certain extent alleviate some of the inconveniences for their on-the-go passenger users. However, the added hourly time-constrained advanced searching still inconveniences their vehicle driving users to be mindful of their advertised pre-trip given schedules, even though that may not always be objectively possible, due to unforeseen events such as: road accidents, gridlocks, etc.

Thus, a new form of dynamic carpool and ride-sharing mobile apps and websites is emerging. They are indicated by their use of real-time passenger requests along with real-time vehicle driving users' location data, foregoing the need for well in advance pre-scheduled and advertised trips. Amongst some of the most known and pioneering mobile apps and websites offering dynamic carpooling and ride-sharing are Lyft [7] and SideCar [8]. Both mobile apps are available for iOS as well as for Android, but neither app currently has a web browser user interface. This may be intentional since both are fully natively written, and by our observations both use TCP sockets to communicate with their respective backend services. Therefore both would need changes to make them web browser-friendly. Unfortunately, those code changes could include some rather tedious transformations. This is due to the fact that native TCP socket traffic has not been well suited for consumption in web browsers relying mostly on HTTP, until recently. Another popular application and website is Waze [9], which isn't intended for carpool and ride-sharing. Waze is used for gridlock traffic reporting and avoidance, but accommodates for pickup requests also, and it seems to have taken another approach in comparison to the two aforementioned applications. Although Waze mobile apps are also fully natively written, their website presents a "live map" user interface which maps events reported by their users. Such events include pickup requests and replies of passenger and vehicle driving Waze users who are otherwise linked either via a popular social network, or through email and SMS. Fortunately, access to those real-time events has now been extended from its native app users to even non-Waze users through private URLs which lead to a live map connected to the Waze backend via a GeoRSS [10] feed through HTTPS (Figure 1).

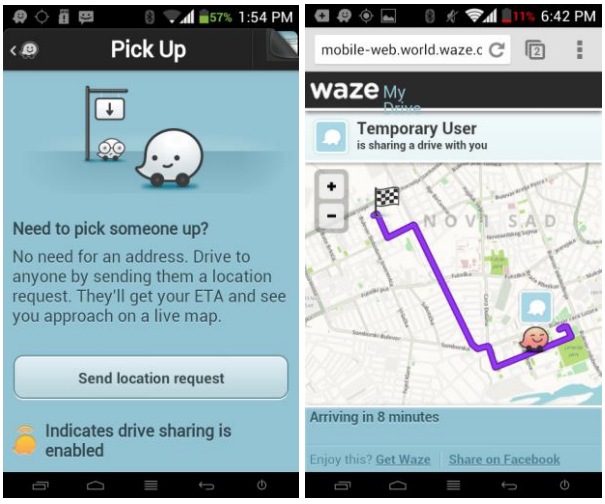


Figure 1: Waze "pick up" and mobile "live map" UI.

The original GeoRSS XML format is however transformed to JavaScript Object Notation (JSON) presumably for easier (JavaScript) client-side consumption and traffic overhead reduction. However it is still limited by a set update interval time. To our knowledge, there are no other globally popular websites

and mobile apps that currently allow for carpool and ride-sharing uses, excluding commercial taxi dispatchers.

2.2 Current carpool and ride-sharing papers

Because static carpool still represents the majority of existing solutions, almost all of the available papers and literature on carpool and ridesharing mainly deal with the static ridesharing issues. In the static carpooling users must pre-schedule their trips, neglecting the dynamic aspect. Despite much of the progress experimented on dynamic carpooling and ride-sharing concepts, it still remains in the early stages regarding publicly available works and literature. In order to make up for that shortfall, some of the papers which mention carpooling and ride-sharing, and even consider the dynamic aspect, like [11], also considered other issues at the same time. Some papers are especially involved in the concepts of traceability, communication and security services. Their authors feel that none of the current solutions evoked these concepts, identifying the security issues as one of the main reasons hindering their success [12]. All cited papers provided us with a lot of beneficial ideas and food for thought transferred onto this paper. They also influenced this paper's findings and conclusions, and out of that still quite disorganized literature, we have identified some yet non-tackled issues, laid out in the following sections. Mainly, we take issue with web browser user interfaces and standardized web technologies which seem to be the unifying way forward, putting ubiquity in the grasp of every hybrid web and mobile application.

3 Design considerations

In the previous section we presented some of the carpooling and ride-sharing solutions, ideas and issues tackled recently. In this section we are building up on those solutions and ideas, proposing some of our own design concepts for a global dynamic real-time carpooling and ride-sharing solution. In Subsection 3.1 we describe some of design concepts that are suitable for a real-time dynamic carpooling and ride-sharing solution. Subsection 3.2 further describes our ideas, allowing for the proposed real-time solution to tackle the problem of being able to serve up to a global user base, adding cloud and distribution design concepts. Finally, in Subsection 3.3 we deal with the ubiquity problem, considering the client user interface technology we feel will be future-proof and available on almost all new mobile and desktop platforms.

3.1 Real-time dynamic solution design concepts

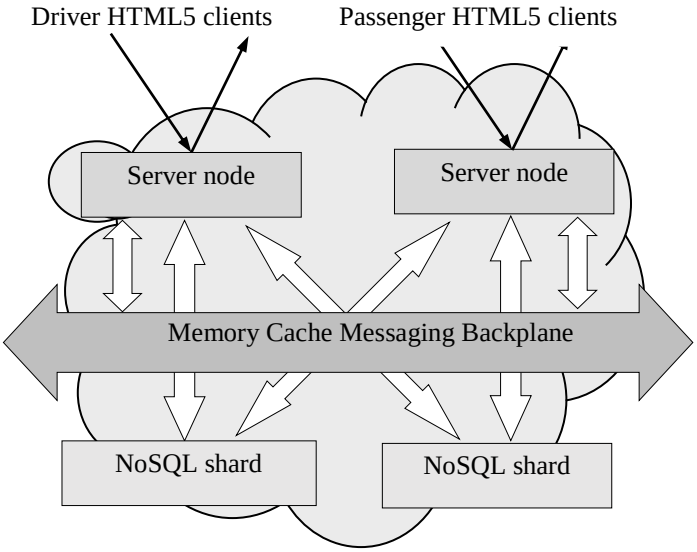
As it was noted in Section 2, real-time dynamic carpooling and ride-sharing solutions are becoming more common. However it takes more designing effort to achieve real-time dynamic capabilities than for mere static carpooling and ride-sharing. The reason for the recent increase is obviously because real-time dynamic solutions are more convenient, and thus more likely to be used in greater numbers by end users. Additionally, some technologies previously used for seemingly real-time communication on the web, have only recently matured and have been standardized.

In the beginning of a so called Web 2.0, at the time when real-time updating websites were only just starting to appear, most of those websites used Asynchronous JavaScript and XML (AJAX) [13]. AJAX is a group of interrelated web development techniques used on the client-side to create asynchronous, seemingly real-time web applications. Most of those techniques relied upon regular HTTP, a simple request-response and stateless protocol. Having to achieve what was usually two-way communication took some effort for websites and web applications. Developers were using various workarounds, techniques involving the use of the browser XmlHttpRequest object or some other web browser plugins.

The first workarounds developed into techniques known as: frequent polling, long-polling and the so called forever-frames. Although all of those techniques were, and still are, very much usable for seemingly real-time web page updates without requiring full page refreshes, they had drawbacks. Their primary drawback was the amount of server-side and network resources they consume. The server is either forced to respond to a large number of frequent requests, or it opens up a number of long running responses, which additionally occupy its hardware resources. On the other hand, using workarounds such as various browser plugins, although they used less network and server-side resources, turned out to be non-practical, because of the lack of plugin support on current mobile devices. For such reasons, new techniques were developed, and recently standardized by the World Wide Web Consortium (W3C). As part of the HTML5 specification Server-Sent DOM Events (SSE) were standardized in 2011 [14], but have not yet been implemented by all desktop browsers, namely, Internet Explorer. However, Web Sockets API [15] was drafted back in 2009 and it is currently supported by all major web browsers. Web Sockets provide a full-duplex communication channel over a single TCP connection, thus allowing for a lower network latency time due to less traffic overhead compared to HTTP. Compared to SSE and other polling techniques, Web Sockets provide the best option for building real-time communication on the web. Due to aforementioned reasons, this protocol is an integral part of our proposed design.

3.2 Distribution and cloud design concepts

Having chosen Web Sockets (WS) as a preferred means of communication, although helping solve latency issues, left another issue unsolved. WS based communication, as all others, supports clients' connections to a single server node. Even though a number of users may be greater when using WS, it still depends on available server hardware resources. Since vertical scaling of server hardware resources can be expensive and still ultimately limiting, the best solution to the near-infinite scaling



problem is horizontal distribution, across multiple server nodes. Ideally, any global real-time solution would be best served in one's own server farm, but given hardware and its maintenance costs, renting cloud resources is a more realistic option. However, for horizontal scaling, one needs to be able to scale data also. Since traditional, i.e. relational data scaling is much harder [16], we have turned to non-relational data (NoSQL). NoSQL databases allow easier scaling and offer better performance in data writes. Additionally, they offer a possibility of scaling reads onto multiple database nodes, combining so called sharding and some parallelism approaches. The notion of sharding denotes horizontal distribution of data across multiple servers. Utilizing aforementioned advantages in a document oriented NoSQL data store that supports geospatial data indexing would make it a perfect fit for our proposed solution and storing our users' location based data. Also, a key-value memory caching NoSQL

Figure 2: Basic architecture design.

data store could be used as a messaging backplane for communication between our individual server nodes, but that use and setup is trivial in the case of our chosen real-time library.

3.3 User interface architecture design concepts

To make a website or mobile app truly ubiquitous, one needs to support as many different desktop and mobile platforms as possible, ideally all of them. Although client applications can be natively written for each platform, there are some unifying user interface technologies for almost all current desktop and smartphone mobile platforms.

So, to achieve ubiquity, we propose the usage of combined HTML5/CSS3 canvas map and form elements for user interface (UI) rendering. Building the UI around streamed real-time data flows of state changes created by passenger and vehicle driving user events is also why the paradigm of reactive programming seems to be the perfect choice. Reactive programming should not be confused with responsive web design, which is also utilized, for same UI reuse across various device screen resolution sizes. Any changes in state registered by user client applications are asynchronously processed when transferred preferably by a Web Socket (as it offers lowest latency compared to other real-time web transport mechanisms) to cloud server nodes and back again to either desktop or mobile clients which are schematically displayed in Figure 2.

4 Prototype implementation

This section describes in more detail implementation choices we made to build the prototype of our distributable cloud-based dynamic real-time carpooling and ride-sharing solution. Subsection 4.1 describes our prototype's real-time communication transport library. Subsection 4.2 deals with our use of geospatial indexed data and gives a comparison of the previously used relational database solution, along with future used NoSQL database and a fast memory caching messaging backplane solution. In Subsection 4.3 we present UI implementation details.

4.1 Real-time communication

During our previous research, we have helped develop the first online taxi dispatching solution in Serbia, named TaxiProxy [17]. This solution was fully realized in .NET and is cloud-hosted on Microsoft Azure. Our participation on this project influenced a lot of our primary technology choices for the prototype of a real-time dynamic carpooling and ride-sharing solution described in this paper.

As noted in the design section, the need to have a real-time dynamic carpooling and ride-sharing solution is imperative, since those solutions are what most users currently wish to use. To make our prototype solution real-time capable, the choice to implement it using a library capable of WS protocol communication in .NET came down to a library named SignalR [18]. SignalR is an open-source library for ASP.NET adding real-time web functionality to .NET applications. It also allows for server-side code to push content to the connected clients

as it happens, in real-time. SignalR server is capable of supporting clients written in several programing languages including .NET and JavaScript. The server-side code can push content to those connected clients by a number of transport techniques, most suitable being bi-directional WS, if available. For a server-side the WS transport requirements are either a self-hosted ASP.NET 4.0+ application or one hosted within Internet Information Services (IIS) 8. Server-side hosted on earlier versions of IIS falls back to other means of message transports. For clients, WS requirement issue is a bit lengthier to describe, so it is listed in better detail in the client UI subsection.

SignalR library allows for two types of implementation approaches. First, less abstract, uses persistent connection, a low level class option, which offers basic real-time mechanisms. It just provides mechanisms to notify the server-side backend of client connection and disconnection, receiving and sending asynchronous individual and bulk messages to connected clients. However, second option, named SignalR hubs, provides an easier to use development interface. It allows abstracting away the need to serialize and deserialize request and response messages manually. The power and ease of use of the second option made it a prudent choice for the speedy development of our prototype application. Even though SignalR hubs allow for the collective segregation of the real-time connected clients into groups, such as a driver and passenger group, the connected clients' membership in those groups is unfortunately not automatically persisted. Also, a dependency resolving GlobalHost server-side object, which is just an implementation of the service locator pattern, can also be easily used to communicate real-time messages from one hub onto another. It is for that reason that we decided to split drivers and passenger clients onto two distinct SignalR hubs: a Passenger and a Driver SignalR hub. That choice allows easy implementation of membership persistence later on, which is important when scaling the application onto multiple server nodes. Persisted group membership allows for a single server node instance to fail without losing all of the membership data of our perspective clients connected to the failing node. As each server node replicates the original driver and passenger SignalR hub architecture, any passenger or driver clients connected to a failing node server should be unaware of any server-side problems, this due to the fact that the load balancer would then just instruct them to another fully working server node and its SignalR hubs.

4.2 Database implementation and performance considerations

For this prototype implementation we have identified two main entities:

- Driver entity which comprises main identifier, position and availability. For each driver, SignalR updates drivers' geospatial coordinates, i.e. position, based on a signal it receives from drivers' devices. When a driver picks up enough passengers, to utilize desired number of empty seats, it becomes unavailable for other passengers to join.
- Passenger entity comprises main identifier and position. Passengers' position is also updated through SignalR and data received from passengers' device.

We have chosen two databases with different data models for this prototype implementation. As we have already implemented our previous project TaxiProxy on Windows Azure platform, we have chosen Windows Azure SQL Database (SQL Azure) as relational database to test in this research. Additionally, we have decided to use one NoSQL database as we plan to make our solution scalable and deploy it in a distributed environment. There is plethora of available NoSQL databases, but we have opted for MongoDB [19] due to our previous experience with this database. Since IIS 8 was our prototype's hosting platform of choice, it should be noted that the server node could then only have been hosted within the Windows 8 / Server 2012 OS platforms. Fortunately, Windows Server 2012 was made available to end-users on the Windows Azure cloud platform as a virtual machine operating system choice since late 2012, and it is deployable onto an Extra Small low-cost machine instance. Extra Small Windows Azure instance, which entails a shared core processor with only 768MB RAM, may not be the first choice from a performance standpoint. However, it is sufficient for proof-of-concept deployments and for limiting cloud-hosting costs.

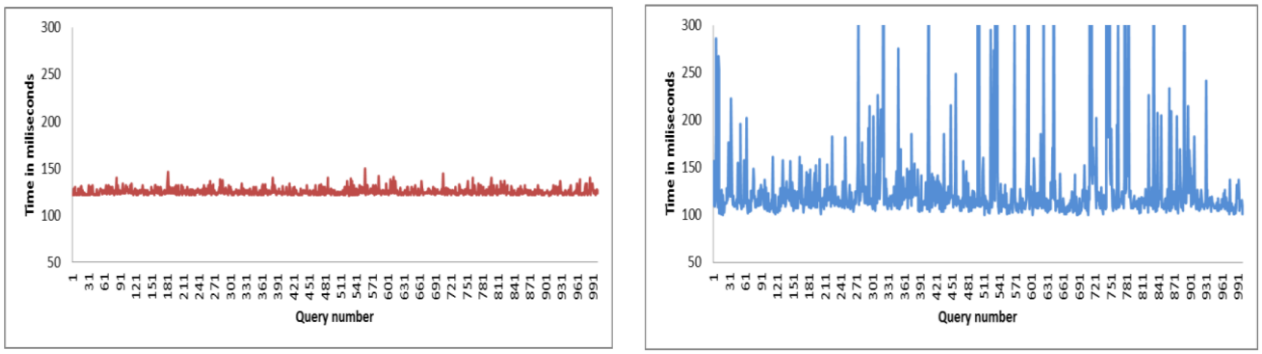
SQL Azure was our first choice to store data as we have used Windows Azure cloud to implement other parts of this prototype. In order to support faster and easier distance calculations between drivers' and passengers' positions, we have used SQLGeography data type for columns containing geospatial data. This data type offers a number of methods for creating and manipulating geospatial data. In order to speed up query execution over geospatial data eve further, we have created indexes on these SQLGeography columns.

One of main motivations to choose MongoDB database came from the fact that MongoLab [20] service offers up to 500MB of free storage for a single-machine instance of MongoDB. This is a major advantage as we want to keep our ridesharing system's costs at minimum to allow cost-free usage of our application for end users. The only drawback is that it is deployed on a single machine which doesn't fully utilize MongoDB's functions as it is supposed to run on a distributed environment. MongoDB is a document oriented NoSQL database and thus we have created driver and passenger documents to store appropriate data. For each driver and passenger data is stored in JSON. Driver's and passenger's positions are stored as longitude and latitude pairs. Similarly to SQL Azure, we have created geospatial indexes to speed up execution of queries.

In order to see which database suits our needs better, we have decided to test these databases based on two criteria:

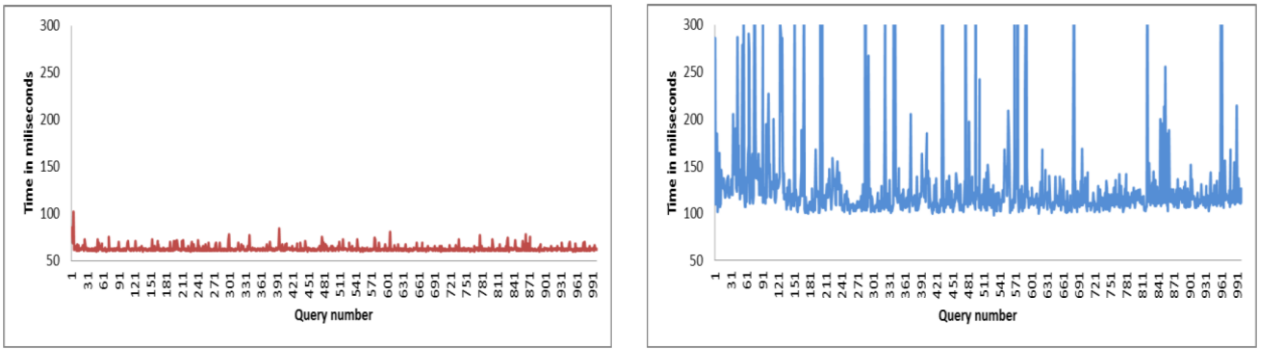
1. *Time needed to execute geospatial query*. This is tested with the most used query in our system that finds five nearest drivers for a passenger.
2. *Amount of storage space needed to store all passengers and drivers*. As the cost of renting cloud based storage is proportional to size of data, we want to keep cost and thus storage size as low as possible.

To prepare a test environment for testing these two database instances, we have set up as similar instances as it is possible with two different data models. We have populated both databases with the same amount of data: 10000 drivers and 20000 passengers. Locations are randomly assigned to all entities. All locations are distributed uniformly within the city area of Novi Sad, Serbia. Both servers, Mongo DB and SQL Azure, are located in EU-West region (geographically closest to us) and are deployed on a single machine. Single machine deployment was chosen as we want to keep service costs as low as possible. Both databases were accessed from a computer with 8GB of DDR3 RAM, 2.67GHz quad core CPU and with 10Mbs/1.5Mbs bandwidth. We have executed 1000 geospatial queries, querying for five nearest drivers to a random location in Novi Sad. Each query was executed using an index on geospatial data. We have taken into consideration only the queries that needed between 50ms and 300ms to execute.



a) MongoDB *b) Windows Azure SQL Database*

Figure 3: Query execution time including time needed to open connection to a database



a) MongoDB *b) Windows Azure SQL Database*

Figure 4: Query execution time without time needed to open connection to a database

In Figures 3 and 4, query execution times are depicted for aforementioned databases. In Figure 3 we have presented query execution times that include time needed to establish and close connections to appropriate database. Both databases performed similarly if we take average times into consideration. MongoDB needed 124.22ms to establish connection, execute query and iterate over results. For the same process, SQL Azure needed 124.83ms. The only difference in these tests is the fact that all query execution times on MongoDB had a small dispersion unlike SQL Server query times. Of all queries on SQL Azure, 33 queries took over 300ms to complete, while all queries on MongoDB were completed in the required time frame between 50ms and 300ms.

In Figure 4 we have presented query execution times without taking into consideration time needed to establish and close connections databases. MongoDB was twice as fast as SQL Azure in these tests. MongoDB needed 62.95ms to execute query and iterate over results. For the same process, SQL Azure needed 122.43ms. Similarly to the test that included time needed to establish and close a connection, MongoDB again had a small dispersion unlike SQL Azure queries. While querying SQL Azure, 20 test cases took over 300ms to complete, while all queries on MongoDB were completed in the projected time frame.

In addition to previously described query time considerations, other main factor in choosing database for final product is the size its data are occupying. Both Windows Azure and MongoLab have limitations on the amount of data their cheapest options can store. As it is already stated, we have populated both databases with the same amount of data: 10000 drivers and 20000 passengers. This data, stored in SQL Azure occupies 1598 KB. At the same time, data stored in MongoDB occupies 3727KB which is twice as many as SQL Azure. This is a direct consequence of the overhead that storing data in JSON format has.

As this amount of data is our prediction for a Serbia-based application, storage size issues are not much of a problem as there is more than 500MB left in both databases to keep them in the same cost range. Even if this application is intended for worldwide use, there is still much space for user base growth. Of greater significance is the speed of query execution. If a connection pool is created to keep connections open, MongoDB is the obvious choice as it is two times faster than SQL Azure. Even in the case of opening one connection per query, MongoDB has more stable execution times. Furthermore, as this is single-machine instance, we expect even greater improvements by deploying MongoDB instance in distributed environment thus allowing Map-Reduce algorithm to fully show its potential. Therefore, we will use MongoDB for a further development of our prototype.

In addition to databases that store driver and passenger data, we have used a NoSQL database to support SignalR server nodes. SignalR server-side code may be deployed alongside a NoSQL key-value memory cache data store named Redis [21]. With the minimum amount of 250MB of RAM allocated to Redis, a fully functioning server node could be produced. Such a node is capable of serving initially large enough number of simultaneous users on its own. Since a new server node can be cloned, and any cloned node's Redis object instance can then be easily subscribed to an existing Redis instance node, we can easily increase the number of new server nodes to meet our future scaling needs. Scale out is easily achieved in part due to SignalR's in-built scaling mechanisms, which uses Redis pub/sub features for a messaging backplane. Each SignalR server node could then be notified of any new WS or other real-time connection channels opened on any SignalR server node through its Redis instance. The load balancer of connected computing cloud instances, which is built into Windows Azure, takes care of diverting traffic to a most appropriate SignalR server node. Such node is chosen based on its current traffic, and it is able to process any incoming new or reoccurring real-time request. But since each node has by then been notified, by its Redis instance each connection should be replicable by any other SignalR node. Therefore each node is capable of replying to any previously opened real-time connection request on another SignalR node.

Table 1: Web browser support for Web Sockets.

Web browser	**Supported since version**	**Supported**
Internet Explorer	10.0 (fully)	Yes
Firefox	4.0 (partially) 6.0 (fully)	Yes
Chrome	4.0 (partially) 14.0 (fully)	Yes
Safari	5.0 (partially)	Yes
	6.0 (fully)	
Opera	11.0 (partially)	Yes
	12.1 (fully)	
iOS Safari	11.0 (partially)	Yes
	12.1 (fully)	
Opera Mini	-	No
Android Browser	-	No
BlackBerry Browser	7.0 (fully)	Yes
Opera Mobile	11.0 (partially)	Yes
	12.1 (fully)	
Crome for Android	25.0 (fully)	Yes
Firefox for Android	19.0 (fully)	Yes
Firefox OS Boot2Gecko	1.0.0-prerelease (fully)	Yes
Tizen OS	2.0.0a-emulator (fully)	Yes

4.3 User interface

Finally, for ubiquity reasons, our choice of prototype client's UI rendering was LeafletJS [22]. LeafletJS is an open-source library that provides HTML5 Canvas [23] mapping. Encouraged by the results of our previous project, TaxiProxy, we felt confident that HTML5/PhoneGap was a right choice. PhoneGap [24] allows a developer to develop a fully functional HTML5/CSS UI and then generate native mobile applications. Therefore, this client could be used on both desktop and mobile devices and have an ubiquitous UI. In both desktop and mobile web client, sharing the same JavaScript logic codebase offers a unified access to a geolocation [25] feature of the device clients are running on. That is a necessary feature for a dynamic carpooling and ride-sharing applications, but also although the look and feel across smaller resolutions changes accordingly, it is not drastically changed. The learning curve for using clients across multiple platforms is thereby reduced by utilizing a responsive CSS3 web design incorporated in Twitter's Bootstrap [26] library.

As we have previously mentioned in Subsection 4.1, real-time request and response messages used by clients to communicate with the SignalR server-side backend are in JSON format. LeafletJS utilizes this format for encoding a variety of geographic data structures named GeoJSON [27]. JSON and its derivatives tend to be lightweight, compared to XML, in an attempt to reduce the latency caused by the need to parse out data from server requests and responses. Additionally, reducing any network latency is also achieved by an attempt to support Web Socket transport, as it is data just sent at the TCP instead of HTTP level but still accessible by the browser.

However, support for WS as a mean of communication transport, depends primarily on a platform web browser's capabilities which is for current desktop and mobile web browsers given in Table 1.

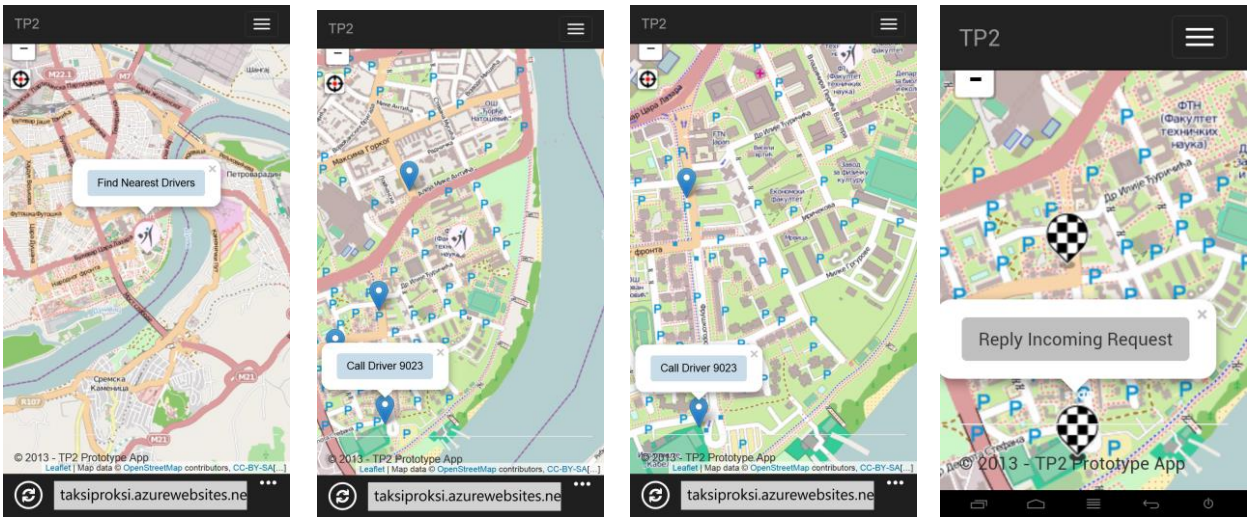


Figure 5: Prototype app screenshot on WP8 and Android platforms, respectfully showing passenger and driver UI

5 Future work and conclusion

Having described our prototype application, of which the early-development stage UI is depicted in Figure 5, our future work and plans envision for it to be deployed and further tested in the real-world. Since the prototype clients were based on previous work done for a commercial online taxi dispatcher, it will initially be tested and deployed as part of that solution to a limited number of taxi drivers. Early adopters of the online taxi dispatching service will then get the benefit of being able to track a few assigned taxis in real-time. The drivers of those taxis will be either issued mobile devices with pre-installed HTML5/PhoneGap web clients or those client apps will be installed on their own devices. Such real-world tests will hopefully lead to identifying problems not yet foreseen. Once a stable solution is reached, the prototype application could and will become a standalone service, open for public use and not just for taxi dispatching and the cost of its operational maintenance could then also be better estimated. If deemed low enough to be offset by ad support according to [28], its use could be free for end users unlike currently popular services like Lyft and SideCar [7, 8].

To reach that point however, some other issues, such as security and privacy, will also need to be tackled. In [29] the solution for the security and privacy issue was implied by use of a 3rd party location based service (LBS). This LBS used claims based authentication protocol OAuth to authenticate and subsequently authorize which exact set of users would be allowed access to the authorizing user's location. Unfortunately, from February 2013 the 3rd party LBS was shut down, and an alternative solution should either be found or developed a standalone service.

Trying to avoid the repeat of having to find alternatives to a 3rd party components not being operational any more, the focus in this paper was on starting to build up our own LBS features respective of privacy using NoSQL. Additionally, our goal in the future would be to also rely on information which can be provided from popular social networks for user authentication. To aid us in that endeavour, instrumental part of the puzzle could be Windows Azure built-in Access Control Service (ACS), allowing for users to

single sign-on to the proposed carpool and ride-sharing service just as if they were signing into the selected social networks. If those users comply, their location data could then only be made accessible to a subset of their social network friends, a widely acceptable solution from a current privacy standpoint.

This paper tried to underscore the need for developing dynamic real-time carpool and ride-sharing solutions, instead of already outdated static ones. Our approach comprises novel web technologies and approaches. Since a prototype has been successfully developed following the outlined design concepts, distribution and cloud strategies, it is obviously possible to build other such solutions using the same approaches. Especially interesting is the possibility to develop a web platform application that runs across multiple devices and their web browsers, be they mobile or desktop. Using an open-source Bootstrap library and Apache Cordova [30] mobile developer platform, which was derived from PhoneGap, is our main topic of interest. We feel this approach could be the unifying tool for any future service supposedly usable across multiple operating systems, current and future. Combining those with some other frameworks which use the HTML5 UI elements such as the canvas element thus adding the ability to render graphical data such as street level maps for carpool should, by our position, be the leading way forward.

References

[1] Dmiitrijevic, D., Nedic, N., & Dimitrieski, V. (2013, September). Real-time carpooling and ride-sharing: Position paper on design concepts, distribution and cloud computing strategies. In Computer Science and Information Systems (FedCSIS) 2013 Federated Conference on (pp. 781-786). IEEE.

[2] Ozanne, L., & Mollenkopf, D. (1999). "Understanding consumer intentions to carpool: a test of alternative models." In Proceedings of the 1999 annual meeting of the Australian & New Zealand Marketing Academy. smib.vuw.ac.nz (Vol. 8081).

[3] Fraichard, T. (2005). "Cybercar: l'alternative à la voiture particulière." Navigation (Paris), 53(1), 53-74.

[4] Dargay, J., & Hanly, M. (2007). "Volatility of car ownership, commuting mode and time in the UK." Transportation Research Part A: Policy and Practice, 41(10), 934-948.

[5] Massaro, Dominic W., et al. (2009) "CARPOOLNOW: Just-in-time carpooling without elaborate preplanning." the 5th International Conference on Web Information Systems and Technologies. Lisbon, Portugal. 2009.

[6] The largest car sharing network for cheap, green travel in Europe. Web – www.carpooling.com

[7] Lyft. Web – www.lyft.me

[8] SideCar. Web – www.side.cr

[9] Outsmarting traffic, together. Web – www.waze.com

[10] GeoRSS. Web – georss.org

[11] Sghaier, M., Zgaya, H., Hammadi, S., & Tahon, C. (2011). A Distributed Optimized Approach based on the Multi Agent Concept for the Implementation of a Real Time Carpooling Service with an Optimization Aspect on Siblings. International Journal of Engineering (IJE), 5(2), 217.

[12] Sghaier, M., Zgaya, H., Hammadi, S., & Tahon, C. (2010, September). A distributed dijkstra's algorithm for the implementation of a Real Time Carpooling Service with an optimized aspect on siblings. In Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on (pp. 795-800). IEEE.

[13] Garrett, J. J. (2005). Ajax: A new approach to web applications.

[14] Hickson, I. Server-Sent Events, W3C Working Draft 20 October 2011.

[15] Hickson, I. (2010). The Web Sockets API, W3C Working Draft 29 October 2009.

[16] Cattell, R. (2011). Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39(4), 12-27.

[17] Najbrži put do slobodnog vozila. Web – taxiproxy.com

[18] ASP.NET SignalR: Incredibly simple real-time web for .NET. Web – www.signalr.net

[19] MongoDB. Web – www.mongodb.org

[20] MongoLab. Web MongoDB. Web – www.mongolab.org

[21] Redis. Web – www.redis.io

[22] LeafletJS. Web – www.leafletjs.com

[23] HTML5 Canvas. Web – www.w3.org/TR/2009/WD-html5-20090825/the-canvas-element.html

[24] PhoneGap. Web – www.phonegap.com

[25] Popescu, A. (2010). Geolocation api specification. World Wide Web Consortium, Candidate Recommendation CR-geolocation-API-20100907.

[26] Bootstrap. Web – www.getbootstrap.com

[27] GeoJSON. Web – www.geojson.org

[28] Goldstein, D. G., McAfee, R. P., & Suri, S. (2013, May). The cost of annoying ads. In Proceedings of the 22nd international conference on World Wide Web (pp. 459-470). International World Wide Web Conferences Steering Committee.

[29] Dimitrijević, D., & Luković, I., & Dimitrieski, V., & Vasiljević, I. (2013) "Orchestrating Yahoo! FireEagle location based service for carpooling" 3rd International Conference on Information Society Technology and Management, Kopaonik, Serbia, 2013.

[30] Apache Codrova. Web – cordova.apache.org