Orchestrating Yahoo! FireEagle location based service for carpooling

Dejan Dimitrijević, Ivan Luković, Vladimir Dimitrieski, Ivan Vasiljević
University of Novi Sad, Faculty of Technical Sciences

ABSTRACT

This paper proposes orchestrating Yahoo!'s FireEagle public location based service to build web and mobile applications intended to be used jointly for a near real-time capable carpooling service.

1. INTRODUCTION

The number of cars on the roads today is ever-increasing, increasing congestion, putting strain on the available infrastructure, as well as diminishing resources available for new car production. Instead of producing ever more cars, and expanding the road infrastructure, there are be better solutions for optimal uses of all existing resources. One such already existing solution is carpooling, allowing for more than one person to use a single transportation vehicle, but carpooling usually demands for the driver of the carpooling vehicle to make prior arrangements with all of the carpool requesting users, agreeing at least upon a convenient pickup place.

The idea of this paper is to propose a carpooling solution allowing carpool requesting users to make requests and expect a response in near real-time once at least one carpooling vehicle driver accepts the incoming request. However, to achieve such a thing, the precursor must be the existence of some location-based service which would first allow for real-time location sharing between the drivers and carpool requesting users. Even though it is quite possible to build such a location based service, this paper proposes exploring orchestration of public location based services (LBS) instead, incorporating their APIs into the proposed carpooling solution.

The fact that the largest social network and search engine providers such as: *Facebook, Foursquare, Google, Yahoo!* already have built LBS solutions, and have made them publically available along with their corresponding APIs gives us confidence in their use. Unfortunately, Facebook (Places) and Foursquare API LBS solutions are not viable for carpooling system envisioned, since they focus mostly on places, primarily sharing textual venue descriptions of user location's and not exact GPS locations. Google Latitude, on the other hand, uses exact GPS coordinates, but currently limits the number of daily location updates and thus real-time location sharing.

Yahoo!'s public LBS called FireEagle (FE) is a free, service-offering website, that collects information about its user's location updates. With explicit user's permission, other previously registered, and thus FE trusted web and mobile applications can also easily either update that information or access it. That way, FireEagle, exposed to third-party use via methods of its API, is designed for helping other applications respond to its users' locations, using their location data to power games, local information services, friend-finders, and potentially for friend vehicle tracking also. Similar vehicle location data has been used already for building taxi service automatic vehicle location and dispatch systems (AVLDS) [1].

FireEagle allows for sharing users' locations with other sites and services safely through a secure server and a standardized authorization protocol – OAuth. All users can themselves decide what to share about their location with any other site or application that uses FireEagle as its location provider, choosing how much detail to share with those applications (exact point – GPS data, neighborhood, city, state, country).

Having all of the above in mind, this paper proposes building a web and mobile application, both using Yahoo!'s FireEagle (FE) as their underlying LBS. The two applications (web and mobile) work in sync, jointly providing their users with a single service, allowing for public carpooling requests and acknowledgement messages to be exchanged between users. All users of those applications, providing user consent is previously explicitly given to FE, can track each other's statuses (willing to carpool, currently busy, etc.) and possibly location. Unfortunately, FE disallows tracking and locating multiple users simultaneously by mobile applications, so a web application which is privy to a general-purpose public/secret token must be used instead. Such a token allows FE registered web applications to issue the, so called, general-purpose FE API method calls, best suited for locating multiple carpool driver users with a recent certain (non-busy) status update, located within the given radius of a given geo-location centered area. Those non-busy users, located within a given area can then in turn notified of the incoming carpooling requests. Once one of them accepts such a request, the carpool requesting users will be notified back which user accepted their carpooling request. Afterwards, just by accessing FE, both user's mobile applications can track each other.

Because relying upon a public LBS foregoes the need to develop one's own location-based service allowing for real-time location sharing between the carpool vehicle drivers and carpool requesting users, in this position paper the case is made for an exploratory use of the public LBS solution as an alternative to an in-house LBS solution. Description of the principle design alongside some of the issues stemming from having to comply with the terms of use policies covering the use of an already built LBS have been addressed in the next section. After that section, various other components of the solution are identified with their infrastructure requirements. And finally, the current state of application development is given along with the proposed evaluation strategy to be undertaken to test real-world solution viability.

2. PRINCIPLE DESIGN AND POLICY ISSUES

As said, all web applications, registered as such with Yahoo!'s FireEagle (FE), can make general-purpose FE API method calls. This means that, besides being able to call distinct user-specific tracking and location updating FE API methods (such as *user* and *update*), every FE registered web application can also call API methods for accessing information of all users of the web application in question (e.g. finding all users who recently updated locations, and all users within a certain location – by, respectively, using *recent*, *within*, and *lookup* API method calls – the last one also being a general-purpose API method call, but not necessarily returning user's locations, being used instead only for reverse geocoding parameters).

The point of the given is, that by using a FE-issued web application's general-purpose access tokens, one can potentially intersect the results returned by both *within* and *recent* API method calls (forming a resulting list of user-specific access tokens, which identify those web application specific users who updated their locations recently in a particular area of interest, or in a location within a given lookup area).

For a workflow diagram, which graphically illustrates the carpool seeking and related actions, please notice the figure 1 diagram. In application's simplified use-case scenario, this would mean that a user seeking a carpool ride (1) could potentially rely upon the proposed FE registered web application and its FE API method calls (2,3) to find carpool drivers (4) which have only recently "checked-in" to/or near a location/area in which the carpool requesting users are currently located. Also, driver mobile application will update that user's location with FE only when that user is willing to accept another carpool request.

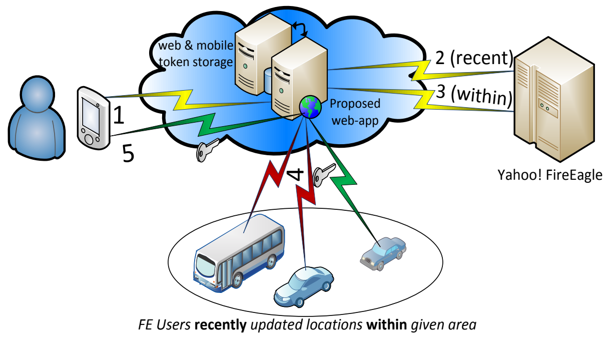
web & mobile token storage
Proposed web-app
2 (recent)
3 (within)
Yahoo! FireEagle
1
5
4
FE Users **recently** updated locations **within** given area

Figure 1 *– Simplified application use-case scenario*

Please notice also that the resulting set of FE users who recently updated their locations within a given area is located at the bottom of the diagram. That set of carpool drivers corresponds to a list of the user-specific access tokens previously returned in an intersection of the *within* and *recent* FE API method call results. The web application uses those same tokens to identify and sequentially "call out" individual carpool drivers, which in turn either agree to a pickup of a carpool requester or disallow the request (illustrated by a green or red link colors).

Once a "near-by" recently checked-in user of the proposed web application willing to accept a carpool request is found, the web application then sends back the data to the original carpool requesting user about that agreeing user, but sends *related* user-identifying data to that user's FE registered web application user-specific access token (5). Related and not identical data, because use of a web application user-specific and particularly general-purpose API method calls by FE registered mobile applications is not allowed by FE terms of use [2]. The original resulting user-specific access tokens are then internally used to find corresponding mobile application user-specific access tokens (represented by key symbols). That token will then in turn be passed along by the web application back to the original carpool requesting user.

Beside for ensuring the FE policy compliance, this process is done for another rather important reason. Once both carpool driver and the carpool requesting web application users are informed of their mobile application's user-specific access tokens, they could then potentially use just those tokens and apps to track each other via FE registered mobile application user-specific API method calls. Both users can then track each other by just querying FE and not the web application, reducing proposed web application's bandwidth costs. Since both users now know each other's user-specific access tokens, location tracking can be done by just calling the user-specific FE API method named *user*, sufficiently passing in just the aforementioned user-specific access token parameter.

Thus, little or almost no traffic is being exchanged or directed to no other service than FE, which means that there should be no network traffic directed towards the proposed web application. Since FE is intended and used for world-wide web-scale use, the proposed web and mobile applications should also be scalable to a near-equivalent world-wide web-scale use status just as large as Yahoo!'s LBS would allow.

However, so that this could be more easily achieved, the proposed web and mobile applications should allow for some non-periodic/ajax polling techniques also. Besides periodic polling, for communication with the web application/server, using HTTP, the users' clients can now use technologies such as forever frames, long polling and server-sent events. A new development in modern HTML5 clients is the websocket support, which provides full-duplex communication channels over a single TCP connection. The latter of the listed technologies gets used, the lesser amount of resources get allocated on the web server for the same communication task. But, even using HTML5 (TCP) websockets and offsetting much of the traffic to FE LBS, doesn't guarantee that at a certain point, a single instance web server wouldn't be overrun by a large number of concurrent users, C10K problem [3]. This is why the "elasticity" of the cloud-deployed architecture, where multiple load-balanced web server instances are allocated, comes in handy. However, since multiple web server instances are load-balanced in the cloud, such servers need a preferably fast message backplane, which will be discussed more in the next section.

3. IMPLEMENTATION AND RELATED ISSUES

As said, the proposed web and mobile applications could be scalable to the near-equivalent of the FireEagle's world-wide web-scale use status. This is in large part due to the fact that, even though the backbone of the carpooling modern request/response mechanisms is performed by the logic of the proposed web and mobile application, the most network resource demanding operations, location updating and tracking are almost completely offset to FE. Since FE is a Yahoo! developed product, its resources are considered plentiful for world-wide web-scaled operations within this paper. So, what remains is to implement the backbone logic of the web and mobile applications used during the process of searching for and accepting of carpool driver users in line with previously outlined basic design features.

Since the FE platform uses an OAuth (version 1.0a) implementation as its authorization protocol, it is necessary to first understand and then implement OAuth clients for the web and mobile application, both of which differ slightly.

The main preposition behind OAuth is that to use a resource of a third-party (in this case location data stored in FE), one need not be forced to implement their own authorization, if that third-party (FE) has implemented their own (OAuth) authentication and authorization already. If the application relying on the aforementioned third-party data is then willing to "trust" that third-party's authentication data, and the third-party is also willing to extend their end-user authorization to allow for authorized access to the end-user's data by relying application, OAuth, in essence, offers a standardized approach for implementing exactly such a thing.

Yahoo!'s FireEagle LBS also allows for exactly that, be it for a relying web or mobile (or even desktop) applications, previously registered with that LBS. Once a relying web application is registered with FE (by its Yahoo! registered developer) it will be issued a set of two token pairs, one for a user-specific and the other for general-purpose FE API method calls, the latter pair being only available to applications designated as web and not mobile. The initially issued token pair to all application types, usually referred to as consumer tokens, consists of a key and a secret token – the key being used to unanimously identify the "consumer" i.e. the application requesting access to a specific user's location data and the latter token being used to sign that request. One such request by a web application is illustrated in the diagram figure 2 bellow.

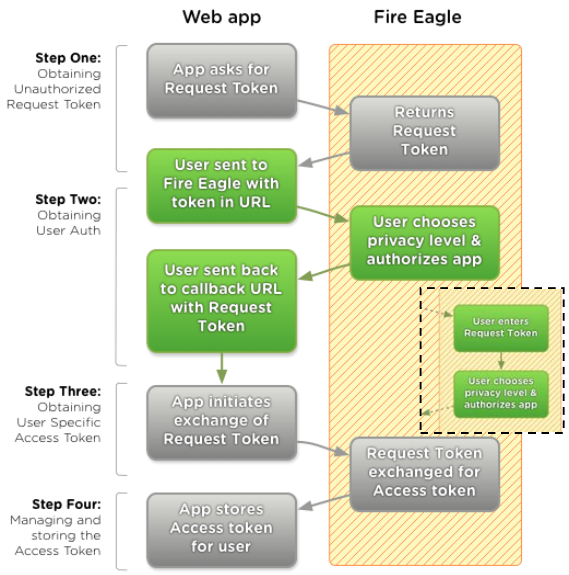
Web app
Fire Eagle
Step One: Obtaining Unauthorized Request Token
App asks for Request Token
Returns Request Token
User sent to Fire Eagle with token in URL
Step Two: Obtaining User Auth
User chooses privacy level & authorizes app
User sent back to callback URL with Request Token
User enters Request Token
User chooses privacy level & authorizes app
Step Three: Obtaining User Specific Access Token
App initiates exchange of Request Token
Request Token exchanged for Access token
Step Four: Managing and storing the Access Token
App stores Access token for user

***Figure 2** – OAuth web and mobile authentication workflow diagrams taken from FireEagle API documents*

As of OAuth version 1.0a all requests coming from the users of a relying (web) applications must contain a non-null callback URL, to which the FE redirects back any user who has set about to authorize access to their location data by the relying web application. As the proposed web application, or better say, its previously registered FE user starts the location data access authorization process, FE receives an initial request formed using the consumer token pair issued back when the application was initially registered by its developer with FE. If the consumer token signed request is recognized by FE as being properly signed, coming with a consumer token from a previously registered web application, then the web application will receive back a so-called temporary request token, which will be only valid for a limited time. If then another request by the proposed web application is sent back to FE, containing the previously issued request token, once the request token is checked for its validity the user will be rendered a FE web form to either confirm or deny access to their location data by the initially requesting relying web application. In case the user disallows the access, the authorization process is stopped there, however if they allow access the request is sent back to a callback URL location, presumably a URL belonging to the relying web application, which can then strip that callback request of a query parameter, the so-called access token. At that point the access token and its secret counterpart used for signing future requests is all that is needed by the relying web application to access the user-specific location data of the user to whom the access token belongs.

The whole process for a mobile OAuth client is quite similar with the exception that the callback URL should be specified as “oob” (out of band). This is done because legacy mobile devices may not all be able to receive back and then even interpret callback URLs and for that reason instead of using the callback URL the authorization process yields an access token which is rendered on a FE web page, just after the user on a web form authorizes access to their user-specific location data by the relying mobile application, which could then be re-entered manually. The reason for this OAuth client implementation difference for mobile devices lies mostly in the fact that FE insists that the authorization for mobile applications be done in the context of a (mobile device’s) web browser (allowing all FE users to recognize the familiar interface along with the address) and not in an embedded web browser control. This essentially causes disconnect in the flow seen for web applications, where the users needs to enter the access tokens manually via browser, shown in figure 2 again outlined in non-solid dashed lines.

For the newer generation mobile devices one might be able to relieve the situation somewhat by registering a custom protocol on a mobile device, which refers to the mobile application, as a callback URL sent initially along with the request for the temporary request token. If the mobile device’s web browser is integrated into the mobile OS and capable of interpreting the callback URL request coming in with a custom protocol, that request can be made to re-open / activate the mobile application which could again strip the request for a query parameter representing the user’s access token automatically.

As that explains most of the logic needed for both web and mobile OAuth client implementation, there remains the need to further explain the logic needed for the server portion of the proposed web application which will be responsible for storing of access token pairs. Since the deployment platform of choice is the cloud, chosen primarily due to its “elasticity”, which will be crucial in case if the proposed web and mobile application use becomes very wide-spread (C10K), currently considered solution for chaining access token pairs issued for web and mobile applications is Access Control Service (ACS), part of Microsoft Windows Azure cloud platform. ACS is responsible for allowing uniformed access to claims issued by various identity providers (in our case Yahoo!’s identity store) to relying party applications (in this case the web and mobile applications). Due to the fact that the relying party application is in fact two applications, mostly due to the FE terms of use, each user interested in being a carpool driver will need to have two access tokens (one for updating their location which is searchable by the web application and another for mobile application use). This overhead of having to track pairs of access tokens would be easily solved by the ACS, since each identity provider’s claims always contain an unambiguously identity-determining piece of data, in case of the Yahoo! identity provider (i.e. user’s Yahoo! registered email address). So, once the user is unambiguously identified in both web and mobile applications, by their email address easily accessible via ACS, their location and status will be easily transferable to and from the context of the web and mobile applications, which is essentially most of the authentication logic that is needed.

As for logic determining the carpool drivers’ statuses, the logic for that can be as easy as this: if a certain carpool driver has "recently" (*FireEagle allows for maximum sequential updates once each 10 seconds or 6 times per minute*) updated their location within the web application’s domain, they will be considered as a user willing to take on passengers by the proposed web application, thus their web application access tokens will be subject to being listed by the *recent* and intersected with *within* FE API method results.

Recapping, once a mobile application carpool requesting user issues a request for a pickup, their mobile device transmitted location will be used to start the web applications *within* FE API call, allowing the web application to locate all carpool drivers within the area containing the previously given location. If that search yields results, another API call is issued, this time for the *recent* method. The resulting user-specific access token list will represent all the users who are recently updating their location, thus they are willing to accept carpool requests. Once the intersection of recurring *within* and *recent* API calls yields some results, each intersecting user-specific access token will be used to find the corresponding ACS identity, using which the proposed web application will be capable of notifying all those users of the incoming carpool pickup requests on a separate channel (preferably using at least a long pooling one, if the TCP websocket is not supported on their mobile devices). If and when those users accept a carpooling request and the proposed web application is notified of that on the separate channel, which could be easily implemented using open-source websocket compliant SignalR [4] library, that user's corresponding user-specific mobile application access token will be transferred back to the original carpool requesting mobile application user, as will that user's user-specific mobile application token be transferred to the carpool driver accepting user. At that point, both users know of each other's mobile application user-specific access tokens, and in conjunction with the consumer key and consumer secret mobile application tokens which will come baked into the mobile application itself, they will be capable of tracking each other via just FE queries as explained before.

Even though most, if not all, the traffic used for mutual locating and tracking of carpool agreed users is offset to FE, there's still a chance of our proposed web application's web server being saturated by a large simultaneous connections made by carpooling and requesting users (C10K). To overcome such a potential, but possible problem, especially in world-wide web-scale use, multiple load balanced web servers must be introduced, along with a backplane messaging mechanism for their mutual synchronization. To achieve this, one might store all incoming requests and their states in a relational database, but since the proposed solution tends to be near real-time, the issue of storing incoming request states becomes a possible bottleneck when joining data from large data sets. As this is a possibility due to an unpredictable large number of concurrent users which may attempt access at any moment, it is for that reason that a NoSQL memory-caching data store could be used instead. Since Redis [5] is an in-memory persistent NoSQL database, using a custom data model with it is proposed for building multiple pub/sub state-differentiated memory-caches. This also ensures that transformation needed to store states in the database is reduced to just a choice of which cache to store the incoming request in (incoming fresh, broadcasted, timed out…) with each Redis instance cache being subscribable to and having inbuilt time-to-live (TTL) which could implicitly incur state transitions once timeouts occur. All of the described implementation details given are again represented at high abstraction level using the diagram given in the figure 3.

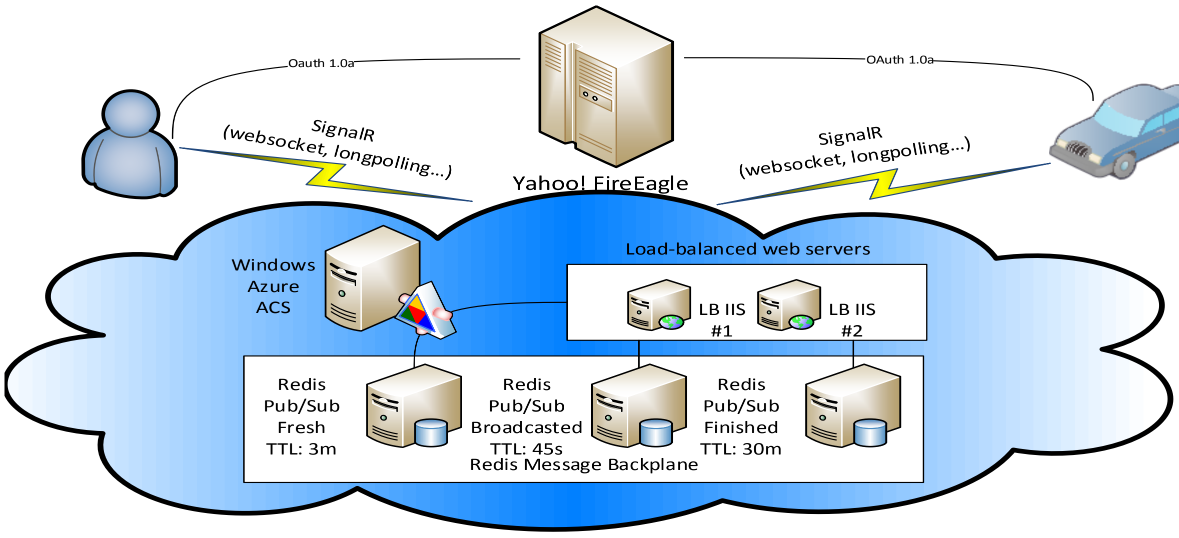
Oauth 1.0a
OAuth 1.0a
SignalR (websocket, longpolling...)
SignalR (websocket, longpolling...)
Yahoo! FireEagle
Windows Azure ACS
Load-balanced web servers
LB IIS #1
LB IIS #2
Redis Pub/Sub Fresh TTL: 3m
Redis Pub/Sub Broadcasted TTL: 45s
Redis Pub/Sub Finished TTL: 30m
Redis Message Backplane

Figure 3 *– High abstraction level implementation diagram*

4. CURRENT APPLICATION DEVELOPMENT

Currently, both web and mobile applications are proof-of-concept and not of public staging or production quality thus no real-world evaluation has been done yet. Based on few open source projects, none of which however were up-to-date, both applications' source codes had to be previously updated for current FireEagle compatibility, adding OAuth 1.0a support. OAuth version 1.0 protocol was found to be subject to an attack vector in 2009, so version 1.0a includes an obligatory non-null callback URL to be always passed along when authorizing users. The common open-source library behind both applications is FireEagleNet [6], which unfortunately supported only OAuth version 1.0. Future web and mobile application development direction includes converging both mobile and web client application presentation layers into HTML5, supported by web socket real-time communication features if the used device browser is compatible.

5. EVALUATION STRATEGY

As this solution not yet fully implemented and deployed, to assess the viability of the proposed web solution some unknowns should be evaluated more thoroughly proposing this strategic order: 1) network latency – during the process of locating and notifying individual carpool drivers FE LBS must be queried using its API method calls, the unknown there is what amount of latency, short of infinite (unreachable LBS), would be deemed acceptable. The proposed metric for this would be to ascertain the processing time needed for current state of the art commercial solutions, calculating the latency needed for break-even performance. Alongside break-even performance figures, the evaluation could also judge the performance figures for performance with exceptions and best case ones, allowing for relative performance to be judged afterwards. 2) NoSQL performance comparison to traditional SQL, relational model, databases – having presumed the fact that the NoSQL solution using a custom data model would over perform its relational model counterpart, it is essential to judge how much faster exactly such a solution could be. The metrics for this that is yet to be determined, however, a possibility would be to construct both relational and non-relational data models needed for the proposed solution to function. Once both data models are functional one could precisely estimate execution times for the most frequent data queries, comparing NoSQL to SQL easy. 3) reduction of network bandwidth – even though it is quite evident that offsetting the bandwidth traffic used for location tracking to public FE LBS would decrease the amount of bandwidth used by the proposed web and mobile applications, it is unclear exactly what amount, both percentage and exact-wise would be saved. To assess the percentage-wise figure one should have real-world estimates to compare with first, estimating the average bandwidth figure needed for location of a first carpool driver willing to accept the carpooling request. This evaluation could only be made once sufficiently high enough number of real-world carpool requests were made and been answered within the solution.

6. CONCLUSION & FUTURE WORK

The expected benefit of the proposed carpooling solution would be significantly reduced total cost of ownership (TCO) figures for implementing and maintaining it, compared to a baseline TCO figure needed to develop and implement one's own LBS from scratch, not factoring in even the additional costs of having to run it. However, by basing it upon Yahoo!'s FireEagle and its resources, we would still keep the proposed solution world-wide web-scalable, which should keep it on par compared to the current state of the art solutions used commercially by recent ride sharing startup services such as Lyft or Sidecar.

Our next step would be to complete the implementation prototype and evaluate it in the real-world, comparing various design options along the way. If a near real-time solution is deemed not yet possible using the public FE LBS due to network latency cloud-server issues only, a custom public open-source FE API compliant replacement could be implemented in-house separately and integrated, still providing location tracking as a service. By logging GPS data from participating carpooling vehicle drivers and carpool requesting users, we could use that data to provide some fee-based recommender systems [7] allowing for further development and refinement of our proposed carpooling solution's automatic dispatch algorithms, as well as offsetting the maintenance cost.

REFERENCES

[1] Z. Liao. "Real-time Taxi Dispatching using Global Positioning Systems". Communications of the ACM, 46(5):81-83, 2003

[2] Yahoo! FireEagle API Documentation - http://fireeagle.yahoo.net/developer/documentation

[3] C10K - http://en.wikipedia.org/wiki/C10k_problem

[4] SignalR - https://github.com/SignalR/SignalR

[5] Redis - http://redis.io

[6] FireEagleNet - http://code.google.com/p/fireeaglenet/

[7] Jing Yuan, Yu Zheng, Liuhang Zhang, Xing Xie. T-Finder: A Recommender System for Finding Passengers and Vacant Taxis. Submitted to TKDE, under second round review, 2012