CabFriendly: A Cloud-based Mobile Web App

Adam Roberts, Harold Pimentel, Sergey Karayev
Computer Science Division
University of California, Berkeley
{adarob,pimentel,sergeyk}@cs.berkeley.edu

ABSTRACT

We present a cloud-based mobile web application to match users who request similar trips and would like to share a cab. The application is hosted on Amazon Web Services and combines several open-source frameworks (Django, PostgreSQL, Redis, Node.js) with social networking and mapping APIs. The modularity of our design allows the service to easily scale in the cloud as the user base grows. We discuss our architectural choices and evaluate scaling performance.

1. MOTIVATION

The goal of our work is to allow users to effortlessly find others to share cab rides with. Reasons to do so are many: it splits the cost, reduces individual environmental impact, and may even be a fun way to meet new people.

We want the system to be intuitive, fast, and usable on-the-go. You shouldn't have to plan a day in advance when you want to leave the bar on Friday night. Additionally, we want users to feel safe about sharing rides by providing useful information and photographs of other riders.

Further extensions to our system would allow cab drivers to see which rides are requested. This may eventually eliminate the inefficiency of non-overlapping dispatch services.

To introduce the components of the system, which we will then describe in detail, we present a typical use case scenario.

1.1 Use Case Scenario

A typical use case scenario, presented in screenshots in Figure 2, sees a user searching for a ride that they want to split cab fare on.

The first user, Adam, goes to the website in the iPhone Safari browser. As described in section 7, our application's UI is built on top of a robust open-source framework and looks the same in all modern mobile browsers. The HTTP request

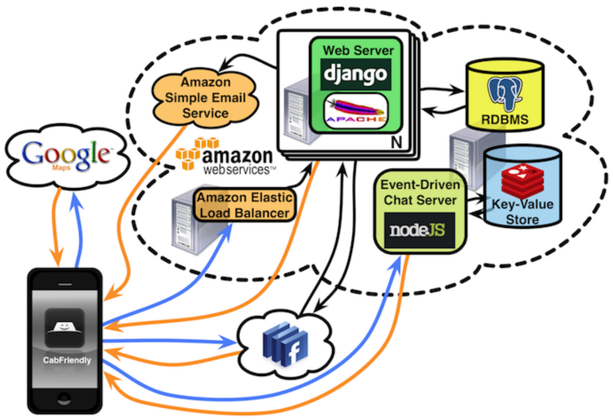
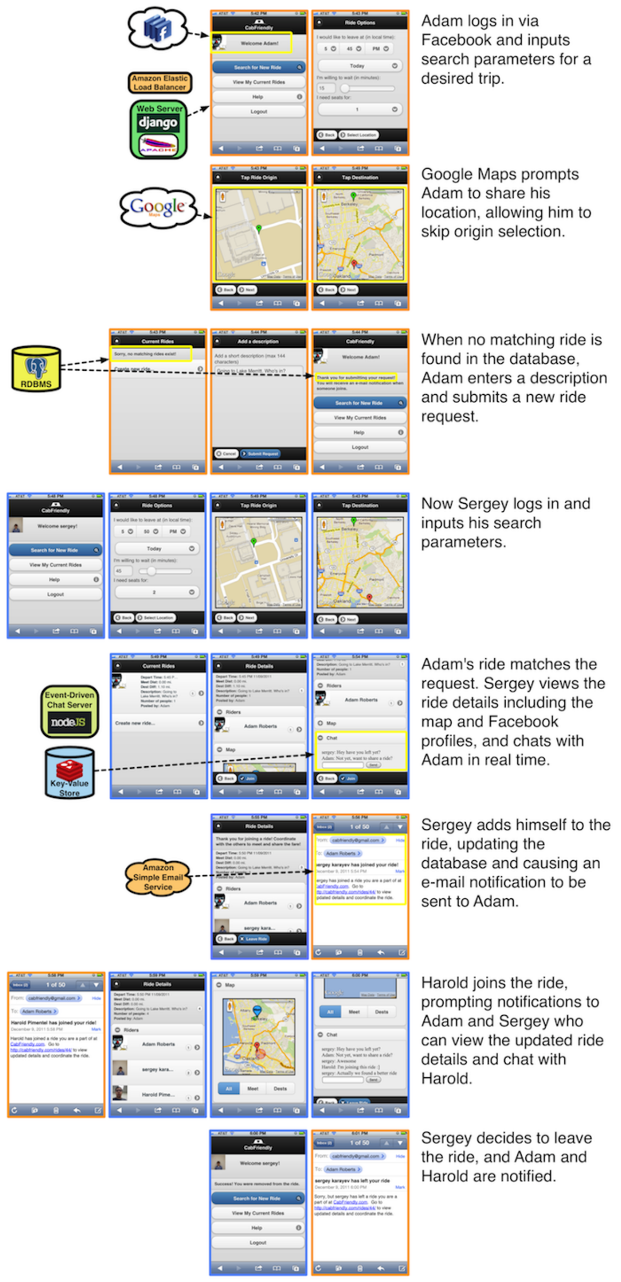


Figure 1: Our application relies on a few open-source components. The server runs on Amazon EC2 instances behind a load balancer. The Apache web servers run Django processes to serve content, which interact with a PostgresQL database (either on one of the web server instances or on a separate instance). Amazon Simple Email Service is used for client notification. A real-time chat server running on the event-driven web server Node.js connects to the client through an update stream. The client is coded in HTML and JavaScript using the jQuery Mobile framework and runs in a web browser on a mobile device.

for the home page goes to a load balancer in the Amazon Web Services cloud, which routes it to one of the EC2 instances running the Django web server on top of Apache, as described in section 3.

Because he has previously given the CabFriendly application permission to do so, Adam is automatically logged in via Facebook, as described in section 4. He goes to search for a new ride, filling out a multi-page form consisting of the desired pick-up time, start location, and the final destination.

The destinations are selected via a Google Maps interface, which automatically gets the user's location if such permission is granted. The form submission hits the database with a complex query and finds no matching rides. Adam then submits his request with a short description, hoping that



Adam logs in via Facebook and inputs search parameters for a desired trip.

Google Maps prompts Adam to share his location, allowing him to skip origin selection.

When no matching ride is found in the database, Adam enters a description and submits a new ride request.

Now Sergey logs in and inputs his search parameters.

Adam's ride matches the request. Sergey views the ride details including the map and Facebook profiles, and chats with Adam in real time.

Sergey adds himself to the ride, updating the database and causing an e-mail notification to be sent to Adam.

Harold joins the ride, prompting notifications to Adam and Sergey who can view the updated ride details and chat with Harold.

Sergey decides to leave the ride, and Adam and Harold are notified.

Figure 2: A typical use case scenario of our application. These are actual screenshots taken on the iPhone 4S.

people will soon join his proposed ride.

A few minutes later, Sergey goes to the website and searches for a very similar ride, which shows him Adam's recent submission. Sergey opens the ride details page and sees the pickup and drop-off points Adam specified.

In the chat tab, Sergey sends a message. Since Adam still has the window open, he sees Sergey's messages in real time—this is described in section 6. Sergey decides to join the ride, which automatically sends Adam an email notification, as described in section 5.

Later, the process is repeated when Harold joins the ride, prompting another email notification to Adam and also to Sergey. For an undisclosed reason, Sergey decides to leave the ride at this point, which results in another email notification.

1.2 Related Work

Several similar shared ride solutions exist, but none combine a mobile interface with quick, local cab ride searches from any location.

Craigslist (`http://www.craigslist.org`) has a "rideshare" section where users can post requests for drivers and passengers. However, the search uses a text-based query instead of a location and time-based one, making it unnecessarily difficult to find matching rides, especially on-the-go. The focus is also on rides with a driver (not a taxi) that are usually several days in the future. There is also no reputation system to provide trust in those you will be riding with and the third-party mobile interfaces are difficult to use.

ZimRide (`http://www.zimride.com`) allows users to post and search for shared rides using location and time, but similar to the Craigslist "rideshare" section, the focus is on long-distance rides posted several days in advance by a driver who wants to split gas money. Reputation is provided by searching within trusted networks such as companies or universities, which limits potential ride matches. ZimRide also lacks a mobile interface.

Fare/Share (`http://www.faresharenyc.com`) has the most similar functionality to the service we propose. The app allows for on-the-go postings of shared cab ride requests based on location and time from a mobile device. Trust is provided by user ratings generated from feedback of previous co-passengers.

However, Fare/Share location search is specific to landmarks and neighborhoods in New York City, whereas our service is agnostic of geography. Users must select their destination from a drop-down list instead of simply touching it on a map. Furthermore, only minimal information is given about a user (name, age, sex, current apparel for identification), which would make it difficult to create trust for new users who have not yet generated sufficient feedback.

CabFriendly combines the best features of these services with Facebook integration to provide reputation (even for new users), global usability, and a real-time intuitive mobile interface that allows the creation and fulfillment of ride requests on-the-go.

2. ARCHITECTURE OVERVIEW

Our goals for the development of the application were to use components that allowed both developer efficiency (high-level, well-tested) and scaling capability. A schematic representation of the various components of our system is shown in Figure 1.

We used the popular Django web framework, written in Python, and the open-source database PostgresQL for the back-end. User authentication has many potential security pitfalls. We chose to rely on a widely-used third-party authentication platform: Facebook OAuth. This requires our users to have Facebook accounts, but we consider this to be a positive side effect in that it provides a reputation mechanism for our service.

Recently there has been much interest in NoSQL data stores. However, we decided that not using a SQL database would be a premature optimization for an application of uncertain popularity. In addition, PostgresQL can be scaled to a considerable extent simply by running it on a dedicated fast machine with a large amount of RAM, and sharding if necessary.

For the underlying server hardware, we took advantage of the Amazon Web Services Free Tier, which allows continuous deployment of a single Micro instance and an Elastic Load Balancer. Although we evaluated various solutions, we eventually used an AWS product for our notification system as well.

The client is implemented as an HTML and JavaScript web site, optimized for mobile device access and using the open-source framework jQuery Mobile.

All of these components will now be described in detail.

3. WEB SERVER IN THE CLOUD

Most of our application logic is built atop Django, a popular model-view-controller web application framework written in Python. Django provides robust routing, session management, and an object relational model that is compatible with several database implementations. We run Django on top of the Apache HTTP server, a highly configurable multi-threaded web server.

In addition to the concurrency provided by Apache processes, a robust web application should also be able to scale by adding machine instances. We rely on Amazon Web Services for the hardware, and run the web server on multiple micro instances, all joined to an Elastic Load Balancer. The Load Balancer monitors the "health" of the server instances, and routes requests to the healthiest instances first [1]. As we show in section 8, this setup allows us to scale gracefully with increasing numbers of concurrent requests.

We also configured the Load Balancer with SSL to reduce the likelihood of session hijacking. With this configuration, the load balancer handles the encryption and decryption and forwards the traffic to the web servers without encryption.

We additionally 'rely on Amazon to resolve our domain name, `http://cabfriendly.com` to the load balancer, using the Route 53 Domain Name System.

4. PERSISTENCE LAYER

Our main backing store is PostgresQL [11, 6]. The storage has two main purposes: authentication and session storage, and persistence of ride information. The database is accessed indirectly through the object-relational mapping (ORM) layer provided by Django. Every model class represents a table in the database and objects represent entries in the corresponding tables.

4.1 Authentication and Sessions

Our website requires Facebook authentication [3]. Facebook authentication allows the user to have one less password to keep track of, as well as the ability to interact socially through the app. Since a user's Facebook profile is linked in a ride, it also allows potential riders to decide whether or not they would like to share a ride with other passengers based on the information provided in their profile. To integrate Facebook authentication, we used the Django Facebook plugin [10].

Authentication works as follows. We first start a cookie session with the user's browser, and assign it a session ID. If the session ID is marked as authenticated in our local database, the user login can proceed. If not, the user is prompted to log in via Facebook:

1. The user is sent to a Facebook OAuth page with our Facebook App ID.
2. The user authenticates with Facebook and is prompted to accept permissions that our service requests.
3. If they accept access from the application, Facebook sends a token to our server.
4. Our server replies with token and our Facebook App Secret Key (via SSL).
5. Facebook responds with an access token which we can then use to access the user's profile and post to their wall.

Once the access token is received from Facebook, it is logged in the `FacebookProfile` table, and the user is authenticated via the standard Django authentication system. This entails mapping a session ID from the cookie we set to a table with when the session expires, a foreign key to the `User` model, etc.

After the initial time the user accepts our app permissions on Facebook, the user no longer has to go through the entire procedure. Also, if the user is already logged in on Facebook (and has authorized our app), the entire procedure is done as a series of redirects and the user is not prompted.

4.1.1 Session storage

In addition to authentication information, we cache ride request form submissions in the session table. This allows the user to browse through rides and hit "back" on their browser without having to resubmit the request form.

4.2 Ride Representation and Manipulation

Rides are represented in two models: `SearchRequest` and `Ride`, both of which are sorted by a unique ID. The `SearchRequest` model (Table 1) stores all of the necessary information to match requests based on input from users. In addition to matching requests, its purpose is to help determine a meet-up location if the original creator of the ride leaves.

A `Ride` object simply stores an optional description of the ride posted by the initial user. The purpose of the ride object is to have a key on which matched `SearchRequests` can be joined. To check if a set of `SearchRequests` are part of the same ride, you simply check if their `ride` ID is the same.

Given our design, looking up information about a `Ride` requires finding all `SearchRequests` that list its id. We need to do this every time we display anything about a ride, such as the number of people currently in it—a frequent operation.

To optimize this part of the application, we could denormalize the database and store computed information such as the number of people in the ride table the first time we compute it, and update the values as relevant `SearchRequests` are added or removed. We considered implementing this denormalization a premature optimization, but it is one venue in which to improve scaling performance of the application.

We now discuss how the database is accessed and manipulated through the ORM.

4.2.1 Ride creation

When a user creates a new ride, both a new `SearchRequest` and a new `Ride` are stored in the database. Each object is assigned a unique ID and the `ride` field in `SearchRequest` is set to match the corresponding `Ride` ID.

4.2.2 Ride search

When a user searches for a ride, we perform a query that selects rides that are compatible with the search. Compatibility is defined by,

- Proximity - is the origin within a quarter mile and the destination within two miles?
- Departure time - do the users want to leave at approximately the same time?
- Number of riders - will this rider's party fit in the taxi? We assume a maximum of four riders.

Results are reported to the user so they can browse matches.

We initially discussed other search methods, such as storing each search request and iterating over all requests on regular intervals reporting matches as they happen. This method is similar to polling. We decided against this method since it potentially leaves a user waiting for a long period of time. This method also makes merging `SearchRequests` into a `Ride` more complicated since before you perform an action, you wait for a response from users who may have submitted a request long ago.

4.2.3 Joining a ride

Joining a ride has the most complex application level concurrency control. Since a user might look at numerous different rides, and many users might consider these same rides (but only select at most one), it does not make sense to lock the rides a user is looking at, as this would not allow high concurrency. For this reason, we implemented *optimistic concurrency control* (OCC) [8].

Using OCC allows multiple users to look at different rides simultaneously without any locks. At the same time, since there no locks, this introduces the potential for a race condition every time a user attempts to join a ride. To avoid the race condition, we use the following commit algorithm:

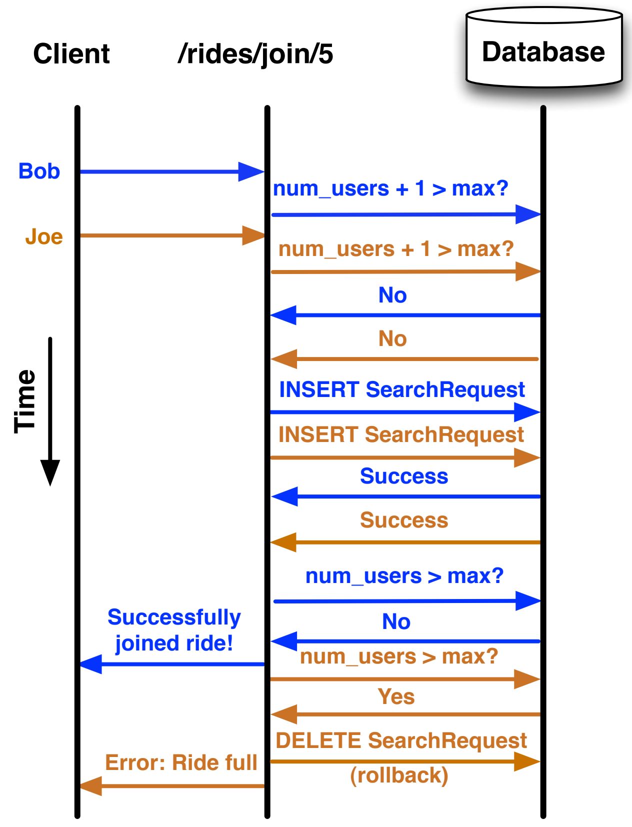


Figure 3: Example of optimistic concurrency control. Two users attempt to join the same ride at the same time but there is only space for one of them. Since Bob was able to join the ride before Joe, Joe has to roll back while Bob successfully commits.

1. **Begin**: Check whether a user can still join this ride. If the ride is full, abort.
2. **Commit**: Write the `SearchRequest` to the database. This is done using a transaction, such that the web server waits until the entry is completely written to the database before proceeding.
3. **Validate**: Check whether or not the ride is overfilled: does the number of riders now exceed the maximum?
4. **Roll back**: If the ride is not overfilled, then we are done. If the the ride is overfilled, we check whether the number of users due to all transactions that occurred earlier than us, plus us, is less than maximum. If not, then we remove the entry (roll back) and notify the user to choose another ride. If so, we know that other transactions will be rolled back, so we are fine.

A user has overfilled the ride when the number of people in the ride exceed the maximum and their row comes after another row (with the same `Ride` ID) where the number of people does not exceed the maximum. The ordering is maintained by the database table primary key.

Thus, if two users insert their `SearchRequest` at approximately the same time and they check if the number of users exceeds the maximum, if the second row that was inserted is the one who overfilled the ride, then only the second user will roll back.

Upon a successful commit, the user's `SearchRequest` will point to the primary key of the `Ride` they joined. An example of two users trying to join the same ride where only one will fit can be seen in Figure 3.

We expect that under normal circumstances there will be few rides that are compatible. Thus, only a small amount of collisions are likely and we will rarely have to roll back any transactions.

4.2.4 Leaving a ride

Terminating a ride is as simple as removing the `SearchRequest` entry from the table. If the owner (initially the creator) of the ride leaves, then it is assigned to the next user who joined the ride. To avoid making an additional query to when a user leaves to check if there are no more users in a ride, we simply leave the `Ride` object intact. In other words, once a `Ride` object is created, it is never deleted.

5. NOTIFICATIONS

When a user joins or leaves a ride, it is important that all other passengers are notified so that they can update their plans and coordinate accordingly. Several options exist for sending such notifications, including device-specific push notifications, text messages (SMS), and email.

5.1 Notification Options

For a native app, device-specific push notifications are ideal due to the fact that they are free and users would receive a high-priority notification on their device including a visual and auditory signal (depending on user settings). Unfortunately, there is currently no solution for sending push notifications from a web app.

SMS messages are another excellent option since the signal is similar in nature to the native app push notification. Nevertheless, this would require knowledge of the users' phone numbers, which are rarely included in Facebook profiles. This could make users hesitant to use our service due to privacy concerns. Furthermore, SMS messages are a relatively expensive form of communication for both the sending and receiving parties.

Therefore, we implemented email notifications, which have the benefits of being free and minimally invasive (Facebook profiles must include email). Moreover, although they do not lead to as high-priority of a signal on most mobile devices (there is no pop-up for example), the notification will be accessible from all of the user's internet-enabled devices, including their personal computer.

5.2 Email Notification Implementation

Given the choice of email notifications, there were multiple ways to implement the service. We initially developed a system inspired by SEDA [12] whereby the web server would place notification requests on a queue, which would be popped by one of a number of worker threads–running in a separate process–with open connections to the GMail SMTP server. These threads would then send the message and attempt to read the next message off the queue, blocking if empty. A thread pool manager would monitor the size of the queue and add or remove threads as needed.

While this system worked very nicely, our tests showed that GMail would begin closing connections and blocking new ones after approximately 200 emails sent in rapid succession. Furthermore, the overall sending was capped at 1,000 messages a day.

We looked for other SMTP servers with more generous limits and considered hosting our own, but ultimately found that Amazon's Simple Email Service (SES) [2] was the best solution for our needs. It allows us to send 5 emails per second, and because we are running our server on EC2, allows for very fast transactions without using SMTP. Moreover, an interface is available for Django (Django-SES [9]) that enables asynchronous sending of emails directly from the web server.

Using this service, we have found the receipt of notifications to be nearly instantaneous and have not experienced any lost messages during testing.

6. CHAT SERVER

The goal of the chat section of the Ride Details view is to allow real-time communication between passengers. Real-time communication between client and server is a non-trivial problem. A possible method could be frequent GET requests to the server from the client asking if there are additional messages. This is unsatisfactory because it overloads the server with constant requests while still not allowing truly real-time chat.

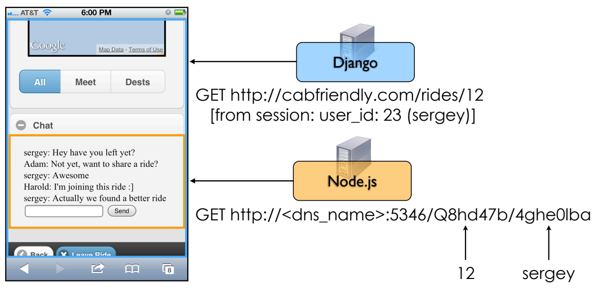
A better way is to use a long-held HTTP request from the client and thus allow the server to push data as it becomes available. In recent years the need for such a web application pattern has emerged, and a collection of techniques for achieving bi-directional client-server communication has become known as Comet.

The problem with using such methods with unmodified Apache web servers is the need for maintaining many concurrent connections to clients. This can quickly overload the thread pool, degrading the performance of HTTP requests that actually want information returned immediately and aren't just waiting for occasional updates from the server.

A solution to this lies in the use of an event-driven web server that would also use non-blocking I/O to allow multiple connections to clients to all be held without the need for a large thread pool. Node.js is a recent implementation of just such a server, built on top of the single-threaded V8 JavaScript virtual machine (originally developed for the Chrome web browser) [5]. Additionally, an open-source framework on top of Node provides implementations of several Comet tech-

Name	Data type	Description
`user`	User	Creator's ID of the `SearchRequest` (foreign key)
`ride`	Ride	Ride ID that the `SearchRequest` is mapped to (foreign key)
`start_{lat,long}`	Float	Start latitude and longitude
`end_{lat,long}`	Float	Destination latitude and longitutde
`submission_time`	DateTime	When request was submitted
`start_time`	DateTime	When user want ride
`expiration_time`	Datetime	Latest the user is willing to wait
`num_people`	Integer	How many people riding with the user

Table 1: `SearchRequest` class model. The model is stored in the database by Django and accessed via the ORM.



sergey: Hey have you left yet?
Adam: Not yet, want to share a ride?
sergey: Awesome
Harold: I'm joining this ride :]
sergey: Actually we found a better ride

GET http://cabfriendly.com/rides/12
[from session: user_id: 23 (sergey)]

GET http://<dns_name>:5346/Q8hd47b/4ghe0lba

Figure 4: The chat portion of the Ride Details page is served from a separate HTTP connection to a Node.js server, using an <iframe>. To provide security, encryption is used in the URL request to the chat server.

niques, and deploys them in order of preference and availability on the client browser[1].

Because our Django web servers are not event-driven, a separate HTTP request is required to load the chat section of the ride details window. We achieve this with the <iframe> HTML tag that makes a GET request to our Node.js server, running on a special port on one of the EC2 instances in our configuration. The GET request includes the ride id of the details page that the chat should belong to, and the name of the user making the request.

We do not want arbitrary users to spoof requests and change their names or access rides that they are not a part of; yet we also don't want to duplicate the authentication functionality of Django on the Node server, or make an authentication HTTP request to it. Our solution is to encrypt the parameters of the GET request, using RSA encryption. This is presented in Figure 4.

6.1 Persistence

The chat history of every ride should be persisted, such that users can look back on rides they were part of and have the complete picture. For this simple functionality, a key-value store is perfect: under each <ride id> key, we store a list of messages of form <user name>: <message>.

[1]The methods are: Websockets, Flash Socket, AJAX long polling, AJAX multipart streaming, Forever iframe, JSONP Polling

We chose the Redis datastore [7] for the task. Redis is an in-memory (with periodic journaling durability) key-value store that is easy to distribute across multiple machines. In addition, it offers data structures more advanced than simple strings for both keys and values. We use the list type for values and string type for keys.

Another useful feature of Redis is an implementation of the Publish-Subscribe messaging pattern, which allows clients to subscribe and publish to named keys. A chat server could therefore be implemented purely in Redis. We did not take this path, because the Socket.io architecture already allowed us to have instant communication between clients in the same way.

Since maintaining chat history is not a critical feature of our application, the absence of hard guarantees on durability are an acceptable trade-off for fast performance and ease of implementation.

Redis runs as a server on a special port on the same machine instance as the Node.js chat server. While we haven't yet found it necessary, it is trivial to distribute Redis across a local network.

7. MOBILE CLIENT

The client-facing portion of the system has several requirements. A user needs to be able to

- search for available rides matching her criteria,
- create a new one if none are available,
- view the details of a ride (current passengers, origins and destinations, departure time),
- and chat with other passengers to coordinate the ride.

Also, all of these requirements need to be met from an internet-enabled mobile device such as a smartphone. To create a simple interface that enabled this functionality, we leveraged HTML and JavaScript combined with the jQuery Mobile framework and APIs from Facebook and Google Maps.

jQuery Mobile provided us with a framework for designing intuitive, mobile web-compatible touch interfaces. Page transitions are triggered by interactions with navigation buttons and occur with a smooth sliding animation similar to native iOS apps. Selections from lists also have a native feel and include thumbnails and condensed information to reduce the number of page requests.

Location information input and visualization is aided by the Google Maps JavaScript API. When a user is selecting her origin and destination, she can simply touch where she wants to leave from and where she wants to go on the map with reference to her current position. When viewing the details of a potential ride, the user can see markers for the origins and destinations of the other passengers in reference to her desired origin and destination. This helps the user evaluate the desirability of a potential ride in terms of how well it matches her location criteria.

The Facebook Graph API makes it easy for a user to see basic information about others a she may potentially share a ride with, increasing safety. The name and Facebook profile picture of each passenger is shown in the ride details and a link provided to the passenger's profile. A user can get a sense of the passenger's "reputation" based on friend count and other information provided by Facebook, and can also select rides the user may feel more comfortable with.

Aside from adding a native-feeling interface through jQuery and for interacting with Google Maps, JavaScript is also used to aid with displaying time information. Timezones are displayed in the user's local time and converted to UTC before being transmitted to the server. The search criteria page also defaults to the next time slot based on the current time using JavaScript.

8. EVALUATION

To test the system, we simulated clients making varying number of requests per second and evaluated the average response time of a single-node server. Then, we tested the scaling performance of our system by increasing the number of server nodes, holding the number of requests per second at a constant high rate. All tests were performed on Amazon EC2 high-CPU medium instances with 1.7 GB of RAM and 5 EC2 compute units.

8.1 Implementation

The tests were implemented in Python using the Multi-Mechanize framework [4]. We set four custom timers: pulling the request page, form submission and search request, downloading the new ride form page (to add a description), and creating a new ride.

Each test used 50 threads, all acting as independent clients. The total time for a simulation was 300 seconds, with a "ramp-up" of 250 seconds. In other words, every five seconds a new client thread was created, until 250 seconds, at which point all 50 threads were making concurrent requests.

Search requests were implemented to be the "best" case from the server end: All start and end locations were completely random, resulting in few to no matches in the database and thus no contention when joining rides.

Each "transaction" involves a user (1) requesting the search page, (2) submitting a search request and loading the list of results, (3) requesting the new ride page, and (4) submitting a new ride.

8.2 Results

The detailed results for one node running both the database and the web server (**configuration one**) can be seen in Figure 5. The plot represents statistics (average and percentiles) of the pooled epoch time, that is, the sum of time spent in all the page requests and submissions during one transaction. This gives an idea of how much total time a user will spend waiting for one entire transaction. Note that the figure shows that during some epochs, the tail latency can become unpredictably high even though the average case and 80th percentiles are relatively consistent. Upon further inspection, we noticed that many search request queries were taking abnormally long, some as long as 10 seconds near peak load.

The same evaluation regime but using separate web and database servers is shown in Figure 6 (**configuration two**). Note that the tail latency is much closer to the average case and there is much less burstiness with this new configuration. Note that simply by making the database server independent, the maximum response time is consistently decreased by about half.

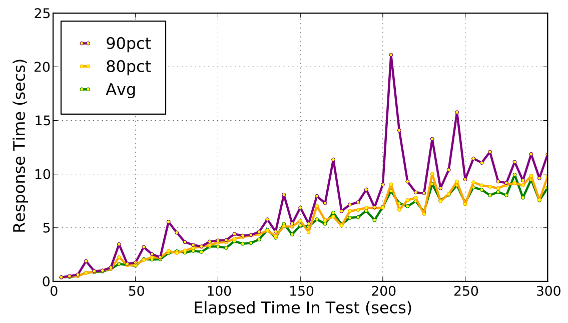


Figure 5: Server response times for configuration one as the number of concurrent client requests increases.

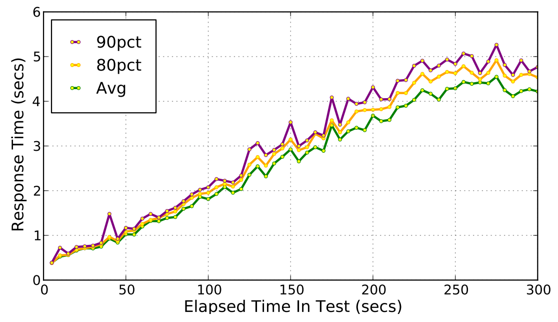


Figure 6: Server response times for configuration two as the number of concurrent client requests increases using one web server and one database server.

We also compared the throughput in terms of transactions per second of both configurations. Figure 7 shows the throughput for configuration one. The plot indicates that while the response time increases as the number of concurrent transactions increases, the throughput stays relatively consistent.

Figure 8 shows the throughput for configuration two. While it also shows a similar trend as Figure 7, the number of transactions per second is higher on average. This behavior is expected as the response time is much lower in configuration two.

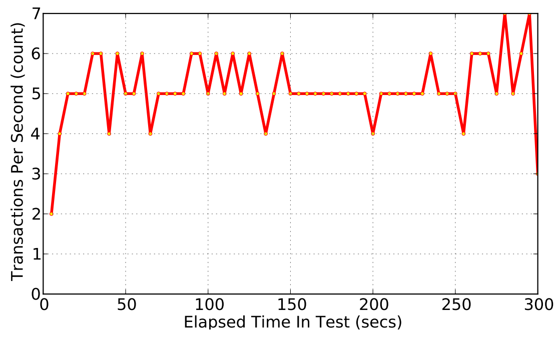


Figure 7: The number of transactions per second for configuration one.

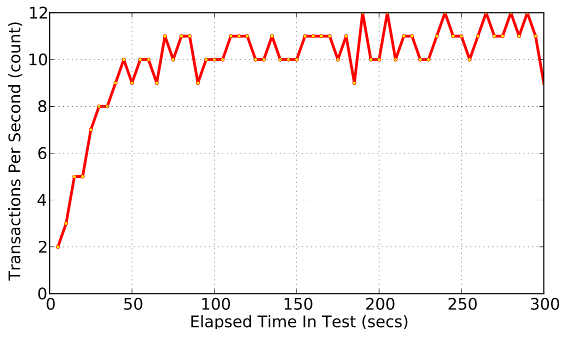


Figure 8: The number of transactions per second for configuration two.

To further evaluate the difference between adding more machines or adding a dedicated database server, we tested the response times at a constant high request load as a function of the number of web servers in both configurations.

As Figure 9 shows, the tail latency when using an independent database is fairly consistent, regardless of the number of web servers, whereas when sharing a machine for the web server and database, the response time continues to decrease.

Nevertheless, configuration one with total of five machines results in a response time that is still worse than configuration two with only two machines. We also note that in configuration two, the response time asymptotes quickly, implying that for a total of 50 concurrent users one or two web servers may be sufficient. This also implies that the bottleneck lies in the database access or the network.

9. FUTURE WORK

The database performance of our service leaves room for improvement. The lowest hanging fruit is definitely in-memory caching of common database lookups. In particular, our architecture requires a lot of session and user authentication

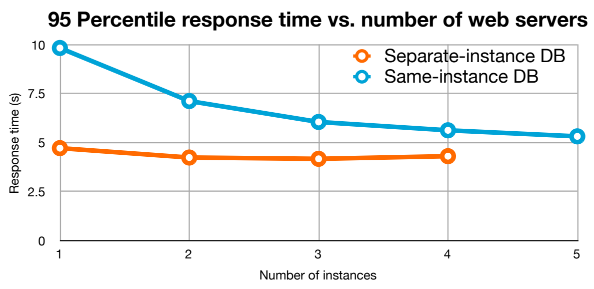


Figure 9: Scaling the number of instances the server is running on.

checks, each one hitting the database with a SQL query. We could easily cache such checks for a period of time, such that user authentication would only hit the database once per a reasonable-length session. It also happens to be the case that our queries are very focused on a ride's start time. It may be beneficial to sort on this field, for example by having keys be the concatenation of the start time with a unique identifier. Moreover, we may find that a NoSQL solution is more efficient for handling our specific queries.

Additionally, we have found that database performance is highly dependent on settings such as paging behavior and maximum available memory. Extensive tuning of these settings to an expected or actual user load and behavior would certainly be in order should our service become popular.

We would also like to develop an interface for cab drivers to be able to find nearby ride requests and "claim" them, thus replacing the current inefficient and expensive dispatch service. The addition of this feature would require relatively few modifications to the client and server models and provide a method for monetization.

10. CONCLUSIONS

We have implemented a mobile web application that enables users to find shared cab rides on-the-go. The application is hosted on the cloud (Amazon EC2), is scalable to many simultaneous users, and integrates Facebook for authentication and social networking. CabFriendly is free to use and is available at http://cabfriendly.com.

11. ACKNOWLEDGMENTS

We would like to thank Lior Pachter for helpful ideas and advice throughout the project.

12. REFERENCES

[1] Amazon elastic load balancing. http://aws.amazon.com/elasticloadbalancing/, December 2011.
[2] Amazon simple email service. http://aws.amazon.com/ses/, December 2011.
[3] Facebook developers. http://developers.facebook.com/, December 2011.
[4] Multi-mechanize. http://code.google.com/p/multi-mechanize/, December 2011.

[5] node.js. http://nodejs.org/, December 2011.

[6] Postgresql. http://www.postgresql.org/, December 2011.

[7] Redis. http://redis.io/, December 2011.

[8] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12:609–654, 1987.

[9] H. Marr. Django ses plugin. https://github.com/hmarr/django-ses, December 2011.

[10] T. Schellenbach. Django facebook plugin. https://github.com/tschellenbach/Django-facebook, December 2011.

[11] M. Stonebraker and L. A. Rowe. The design of postgres. *SIGMOD Rec.*, 15:340–355, June 1986.

[12] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35:230–243, October 2001.