CS 237 Project Report

Group 16: Zhixuan Jia, Shayang Zang, Hao Ni

Section 1: Key objective and proposed work

Social networking has been a hot topic for many years and there are many influential IT companies focusing on this area. The intrinsic complexity of social networking requires the related applications to be highly scalable and reliable. Therefore, middleware plays an important role in these applications.

In this project, we focused on implementing one of the essential features of social networking applications, which is the message exchange feature. Message exchange feature is essentially a publisher/subscriber system, which commonly appears in distributed systems. Our main objective is to build a web application on which people can anonymously publish real-time messages to a group of people to which himself/herself belongs to. We believe anonymity encourages and assists people, especially the ones who are very shy, to better express their ideas or opinions. For example, lots of people are afraid of asking a “stupid” question. However, by posting anonymously within a group, no one can associate this question to you and because users can only join a group with approval from group leader, all the anonymous message posted within a group will be highly related to this group. Furthermore, our application will also allow users to draft anonymous messages to specific group member. These anonymous messages will be stored in our server and only be sent out when some conditions are met.

Section 2: Related efforts

1. *Heroku: Deployment platform*

Building scalable and highly available infrastructure from scratch is expensive and time-consuming. Therefore, many web application developers choose to build, run and operate their applications entirely on some cloud-based infrastructures. These cloud infrastructures are typically maintained by trustworthy IT companies and delivered as a utility (on-demand, available in second, with pay-as-you-go pricing). With ready-to-use infrastructure, developers only need to focus on the development of their applications. Some widely-used cloud-infrastructure services are Amazon’s AWS, Microsoft’s Azure, Google’s App Engine, etc. These cloud-infrastructure can be further classified into two categories: Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). Our choice of the cloud infrastructure for this project is Heroku, which is maintained by Salesforce and offers PaaS. It provides these attractive features [1]:

- Fast deployment (within minutes) using git push
- Useful add-ons such as database, SSL and activity logs
- Each application component can be scaled up independently
- Provide first 750 computation hours free of charge

However, Heroku has its disadvantages. It does not have the flexibility that IaaS offers. Developers do not have much influence on server-side architecture. For example, load balancing, deployment system and server activity logging are determined and automatically handled by Heroku.

2. *Ruby on Rails (or Simply Rails): Development Framework*

Ruby on Rails is a popular open source server-side web application framework written in Ruby. Many high-volume websites such as Airbnb, Hulu, GitHub, Twitter etc. were initially built on Ruby on Rails. Like many other web frameworks, it uses the model-view-controller (MVC) pattern to organize application architecture [6]. It also emphasizes the use of some well-known software engineering patterns and paradigms [2]:

- Convention over Configuration: This design concept free developers from making decisions which are not worthy of recurrent deliberation, which means more productivity gains.
- The menu is omakase: Rails diminishes developers privilege to choose each tool or library during development. Instead, Rails provide a tool box for all. This makes interaction of different tools/libraries much more stable.

These two practices above are the most important reasons that we chose Ruby on Rails for this project. They make Rails good for rapid application development.

3. *Rack: Web server interface*
Rack provides a modular and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and responses in the simplest way possible, it unifies and distill the API for web servers, web frameworks, and middleware into a single method call [3]. Developers only need to know how to configure Rack and Rack will be responsible for communicating with many different HTTP servers, such as Puma, Unicorn, Nginx, etc. To use Rack in a web application, developers create a Ruby object that responds to the *call* method, which takes the environment hash as a parameter, and returns an array with three elements: The HTTP response code, A hash of headers and the response body [3]. Furthermore, developers can easily add a chain of different middleware components between the application and the web server, which naturally forms a pipeline design pattern. These middleware components can handle tasks like enabling caching, authentication, catching exceptions, etc. Some often-used middleware are:

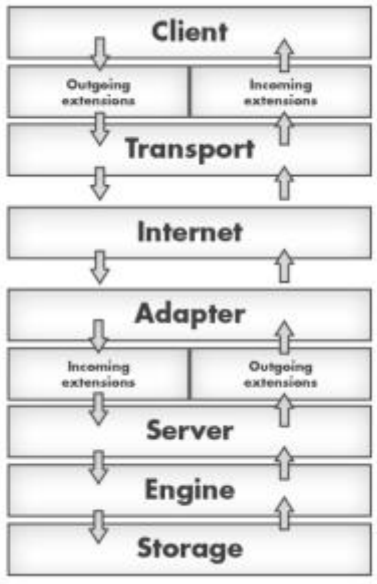
- Rack::URLMap, to route to multiple applications inside the same process.
- Rack::CommonLogger, for creating Apache-style logfiles.
- Rack::ShowException, for catching unhandled exceptions and presenting them in a nice and helpful way with clickable backtrace.
- Rack::File, for serving static files.

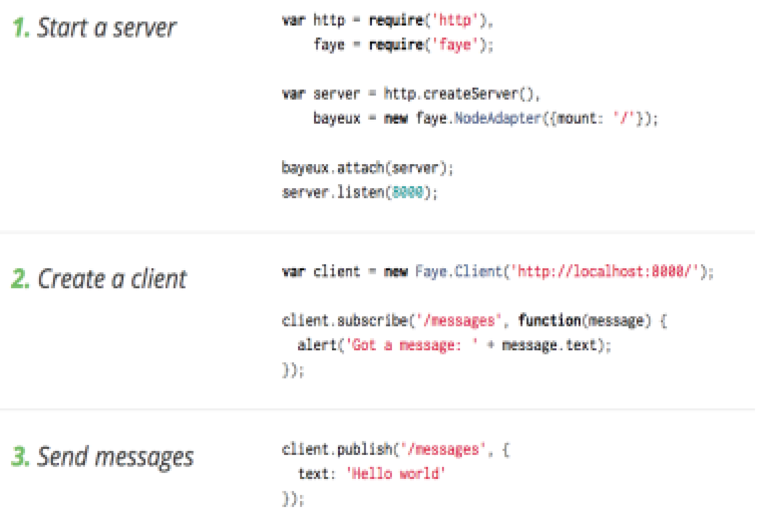
4. *Faye: Real-time pub/sub system*
Faye is a publish-subscribe messaging system created by James Coglan. It is one of the open source implementations of the Bayeux protocol (described in next section). It provides message servers for Ruby, and clients for use on the server and in all major web browsers. Components that make up the Faye's messaging stack are listed below [4]:

- Storage: The storage layer is the base of the whole stack. It stores the state of the Faye service. The state includes a list of active client IDs, which channels each client is subscribed to, and any queued messages waiting to be delivered to clients. This storage can be server's memory or a Redis database.
- Engine: The engine is an object that provides an abstract API on top of the storage layer. It implements the core operations of the Faye service, such as registering new clients, storing subscriptions and routing messages. The engine's operation assume that all necessary validation has been done on incoming data further up the stack.
- Server: The server implements the Bayeux messaging protocol, providing the handshake, connect, disconnect, subscribe and publish operations. The engine executes these operations.
- Server-side extension: This component is written by the developers. They can use this component to intercept messages that flow in an out of the server.

- Adapter: The adapter exposes the server's interface over HTTP. It is responsible for serializing and deserializing messages as JSON and accepting connections over various HTTP transport.
- Transport: Its job is to serialize client's messages and send them to the Server. When responses from server arrive, it deserializes the responses and give them to clients.
- Client-side extensions: These are components written by the user that intercept massages that flow in and out of Client.
- Client: This is the component that developers interact most with. It provides the interface through which subscriptions are registered and messages are published. It also responsible for distributing messages to the right listeners as new messages arrive from the server.
- Clustering: Faye supports clustering. Developers can run a single Faye service across multiple front-end web servers.

Two figures below shows the architecture of Faye and some basic code to use Faye. [4]





1. Start a server

```
var http = require('http'),
    faye = require('faye');

var server = http.createServer(),
    bayeux = new faye.NodeAdapter({mount: '/'});

bayeux.attach(server);
server.listen(8000);
```

2. Create a client

```
var client = new Faye.Client('http://localhost:8000/');

client.subscribe('/messages', function(message) {
  alert('Got a message: ' + message.text);
});
```

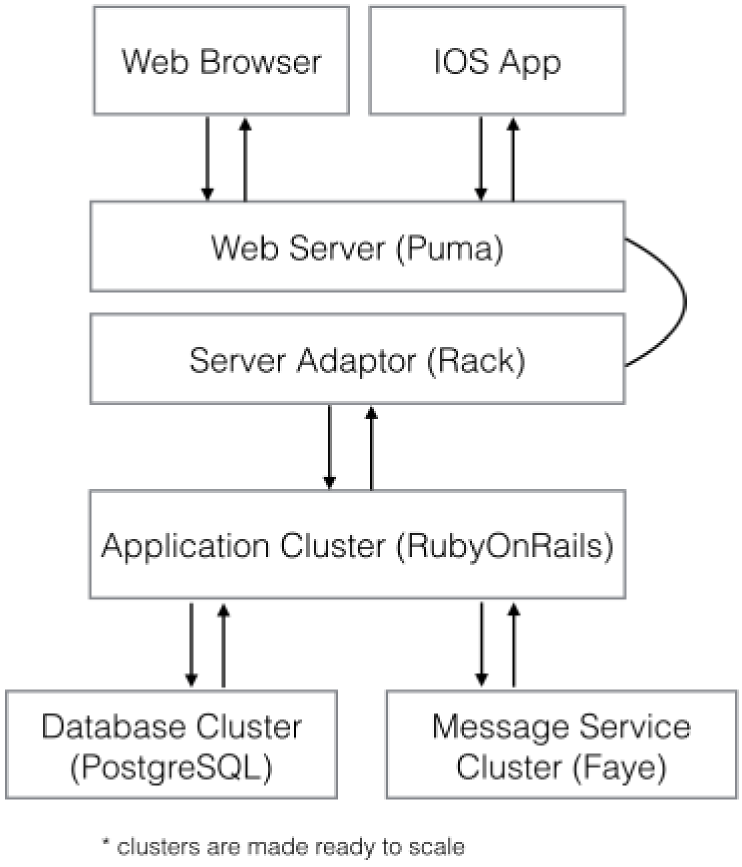
3. Send messages

```
client.publish('/messages', {
  text: 'Hello world'
});
```

In project, we used a Ruby gem called Private Pub which publish and subscribe to messages through Faye. It provides real-time updates through an open socket without tying up a Rails process. All channels are private so users can only listen to events you subscribe them to. [7]

5. *Mobile Friendly: iOS App*
For any messaging application, mobile phone is one of the most important target devices. We developed our own iOS app which made it more convenient for our users to access the application through their mobile phones so that they can send and receive the messages with minimum delays and overhead. Users of the iOS app will directly communicate with our app server through RESTful APIs to publish and receive messages.

Figure below shows how different components interact with each other.


Web Browser
IOS App
Web Server (Puma)
Server Adaptor (Rack)
Application Cluster (RubyOnRails)
Database Cluster (PostgreSQL)
Message Service Cluster (Faye)
* clusters are made ready to scale

Section 3: Approach, methods and techniques

Bayeux Protocol: Foundation of Faye
Delivery of asynchronous messages from a remote server to a web client is often described as *server push*. The combination of *server push* techniques with an AJAX web application has been called Comet. Bayeux seek to reduce the complexity of developing Comet web applications by providing a communication protocol for transporting asynchronous messages (primarily over web protocols such as HTTP and Web Socket), with low latency between a web server and web clients. The messages are routed via named channels and can be delivered: server to client, client to server and client to client (via server) according to t Bayeux protocol. These topic channels are independent of the network transport being used [5].

Section 4: Evaluation plan and implementation issues

1. Overall Performance Evaluation Plan
The performance of an application can be affected by clients' network, web server processing ability and database I/O. The application is currently in testing phase and hosted on Heroku, a cloud-based deployment platform. We plan to invite UCI students to sign up for our web application and see how the growth of the number of users can impact the application performance such as web server response time and database I/O.

2. Web Server
When evaluating or choosing a web server, we mainly considered four factors: First, whether the web server provides buffering for clients' requests. Second, whether the web server supports concurrency. Third, whether it is compatible with Rack. Fourth, how easy it is to be configured. After considering all these three factors, we chose Puma as the web server, which provides concurrency and buffering of clients' request, compatible with Rack and most importantly, very easy to configure. [9] However, the downside is that Puma does not provide more advanced configurations as the other web servers such as Passenger and Unicorn. We may choose web servers with more functionalities in the later phase of the development cycle.

3. Database
Considering the nature of our application, relational database is the go-to option. We evaluated three popular open source relational database: SQLite, MySQL and PostgreSQL. SQLite can be embedded inside the application and store data locally, which enables extremely fast data access. But it hinders the scalability of our application. MySQL is known for distributed deployment, ease of use and high security. But the drawbacks are also obvious: First, it does not ensure ACID compliance. [10] Second, it does not provide good performance in concurrent "write" requests. Finally, we consider PostgreSQL. PostgreSQL is comparable with MySQL in many aspects. Particularly, it alleviates the obvious problems that MySQL has and provides our application with better extensibility because it is also an object-oriented database with an object-oriented programming API for storing and retrieving objects. However, PostgreSQL's performance is not as good as MySQL in concurrent "read" requests, which will become a bottleneck of the application in the long run since the application frequently performs both "read" and "write" operations.

In terms of database migration, Rails framework provides Active Record Module, which is the laying of the system responsible for representing business data and logic. [8] It is a very convenient way to realize the migration of database schema but not the actual data record, which remains an issue to be solved.

4. Future Distribution
The scale of the application has been relatively limited so far. The following high-level distribution strategy will not be used until we can prove there is a market for this application.

- Run the application on a cluster independent of Heroku.
- Use Nginx as a dedicated front-facing server to handle slow clients, load balancing problems.
- Use a dedicated cluster as database to achieve high-availability, high partition tolerance and eventual consistency.

Section 5: Conclusion and possible extensions

Building a functional web application, even within a small scale, involves many components and how to make these components properly interact with each other requires significant amount of work. In this project, we implemented a web application which allows users to publish messages to a group of people in real-time. From web server to message server, from front-end JavaScript and CSS to back-end database logic, each of these components requires careful decisions.

This application can possibly be extended to many other scenarios because being able to exchange messages is essential in most people's lives. One possible scenario we can think of is location-based group messaging. In this scenario, everyone within some specific distance to each other can also form a group. Real-time messages can be published to this group and everyone within a certain distance from the message-sent location is automatically subscribed and will receive a copy of these messages. This scenario is much harder to implement because every user is constantly moving and system has to subscribe and unsubscribe each user very often. This impose great amount of work on the system and most likely will require distributed system combined with NoSQL database.

Code:

- **https://github.com/lishiming9/yipiejihe**
- **https://github.com/zhixuan1120/237_Project**

Reference:

[1] How Heroku Works (https://devcenter.heroku.com/articles/how-heroku-works)
[2] David Heinemeier Hansson: The Rails Doctrine
[3] Rack: a Ruby Webserver Interface (http://rack.github.io)
[4] Faye: Simple pub/sub messaging for the web (https://faye.jcoglan.com/architecture.html)
[5] Alex Russell; Greg Wilkins; David Davis; Mark Nesbitt: The Bayeux Protocol Specification 1.0
[6] Ruby on Rails: A web-application framework that includes everything (http://rubyonrails.org)
[7] Private Pub (https://github.com/ryanb/private_pub)
[8] Active Record Basics (http://guides.rubyonrails.org/active_record_basics.html)
[9] PUMA: A modern, concurrent web server for Ruby (http://puma.io)
[10] Brian Jepson: PostgreSQL vs. MySQL