Project 4 - Secure Facebook API and Simulator
COP5615, Fall 2015

Grant Hernandez and Chelsea Metcalf

December 17, 2015

Security Model

The entire goal of this project to remove the implicit trust in the server. To do this, clients never store plaintext data object on the server. They first encrypt all data objects with a unique key, encrypt the key with the target user's public key, and then transmit the (Key, Object) pair to the server.

Trust Model

From the client's perspective, we trust the security of the machine that the client is running on. We also trust that the client is able to keep its private key a secret from everyone. The client also trusts that the client itself is free of backdoors and remotely exploitable bugs.

The client trusts received public keys for users on first fetch. This public key is hashed and stored locally on the client in order to prevent public key man in the middle attacks orchestrated by the server.

The client also assumes that the transport mechanism for packets to and from the server is secure and not being eavesdropped upon. This project does not require use of HTTPS, so it is possible for secret tokens/cookies to be captured in the clear. In a real project HTTPS would have to be enabled to ensure this trust.

Threat Model

From the client, the primary threat in this security model is a full server database compromise. The server is not to be entrusted with the plaintext data for any object. It is assumed the server can and will modify and attempt to view all data stored on its system.

Security Objects

The notation is as follows:

$$
\begin{aligned}
P_k &\rightarrow \text{Asymmetric Public Key} \\
P_k^{-1} &\rightarrow \text{Asymmetric Private Key} \\
D_k &\rightarrow \text{Symmetric Data Object Key} \\
E(Key, Data) &\rightarrow \text{Encrypts } Data \text{ using } Key \\
D(Key, Data) &\rightarrow \text{Decrypts } Data \text{ using } Key \\
Sign(AsymKey, Data) &\rightarrow \text{Signs } Data \text{ using an Asymmetric Private Key, } AsymKey
\end{aligned}
$$

Client Objects

- Each and every client has a unique private key P_k^{-1} and public key P_k. This key pair is generated on the client using RSA-2048 with secure random number generators. The private key never leaves the client's machine, but the public key is freely shared to the server.

- The client uses P_k^{-1} to authenticate the data objects it creates and posts to the server through signing: $Sign(P_k^{-1}, Hash(Obj))$, where $Hash$ is a cryptographically secure random hash function.
- It uses P_k to encrypt symmetric keys, D_k, destined for a target user, P_k, or itself:

$$\begin{aligned} TargetPubKey &= P_k \\ ObjKey &= D_k \\ EncObjKey &= E(TargetPubKey, ObjKey) \end{aligned}$$

- It uses P_k^{-1} to decrypt data objects received from the server and destined for it:

$$\begin{aligned} EncObjKey &= E(P_k, D_k) \\ ObjKey &= D(P_k^{-1}, EncObjKey) \\ Obj &= D(ObjKey, EncObj) \end{aligned}$$

Server Objects

The server must maintain a `Keychain` containing a Map of $Identifier \rightarrow KeyData$, where Identifier is a unique user ID and KeyData is an object containing the key material for that user (if authorized to view the object). `KeyData` is an object that stores the created time, expiry time, and the encrypted data key.

In order to enforce basic object permission, each object has an Access Control List (ACL). The `FBACL` maintains information on the object owner and the permissions granted to other consumers of the object. For example, if the object permissions are ONLY_ME then only the object owner is able to view and interact with the object.

FAQ

How are we validating a user's data without ever being able to read it? If a certain field cannot contain commas, we are trusting the client to not put commas. Conversely, clients who can view stored data objects have to be trusted to correctly validate/display the data correctly, regardless of actual content. Given the security model, there is no possible way to validate the content on the server. The clients must know the format of the data and reject or transform any data not matching the expected format.

How are previously trusted user's stripped of keys once they are no longer trusted (no longer friends)? The only person who can modify the keys for a data object is the object creator. If Alice is initially friends with Bob, who shares a post with her, but then soon after defriends her, Alice's key to the post must be revoked. Revocation is impossible as Alice may have saved the key and post data for later inspection. This is analogous to saving a web page, so it is not of great concern. The server is trusted to perform correct Access Control List checks in order to prevent Alice from getting a fresh key and post data. The only case where this model would fail would be if the server is compromised and a database of online keys was used to decrypt stored data objects. One way to avoid this would be to rotate the objects key on a group membership change, or at a predefined interval, which ever comes first.

What happens if a server is compromised and all data objects are leaked? The security model states that *no key material is stored on the server*. All data objects are encrypted using secret keys available only to the clients. The objects are end-to-end encrypted and a server compromise would not expose their contents. However, all *metadata* would be exposed, including friends lists, content size, content times, and so forth.

What happens if the server is compromised or the server operators maliciously replace data objects? A server could maliciously replace a data object from Alice destined for Bob merely by encrypting a new object with Bob's public key. The server controls the metadata and cannot be trusted to keep accurate records. The defense to this attack would have Alice *sign* all of her data objects using her private key in order to authenticate them.

What happens if the server maliciously replaces public keys for users? In this security model, clients trust the server to maintain an accurate directory of user's public key. If the server operator or other third-party were to replace a stored public key for a user with their own, then they could man in the middle any data being sent. This becomes the TLS certificate validity problem, which is not solved. There are methods to help detect invalid public keys, such as Public Key Pinning on the client or something similar to Certificate Transparency. These solutions would only allow quick detection, but they each have their own advantages and disadvantages.

Code Structure

Client Simulator

The client simulator is structured to create users (with each having their own actor), pages, and to have users make friends. Once these initial tasks are complete, the user actors begin simulating activity. The structure of the simulator is shown in Figure 1. Based off of a certain "load factor" users will post more frequently. This can be thought of as their aggressiveness.

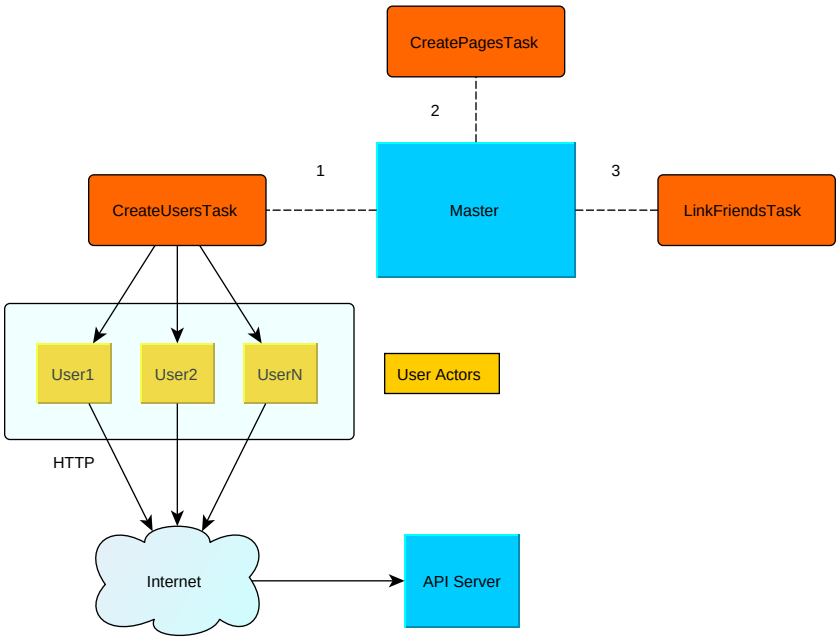

Figure 1: Client structure

API Server

The API server utilizes Spray to build a REST API using the Spray-routing DSL. In each unique REST route either an entity is deserialized using Spray-JSON or parameters are extracted from the route URI. Once the inputs are extracted, they are passed by message, along with the `RequestContext` to a data manager Actor. This actor serializes requests to Facebook data objects and implements all of the business logic for the site.

This includes creating users, pages, posts, albums, and pictures. It also allows these items to be looked up by ID and retrieved. Additionally, auxiliary functions such as adding friends are implemented here due to the easy data object access. This is shown in Figure 2.

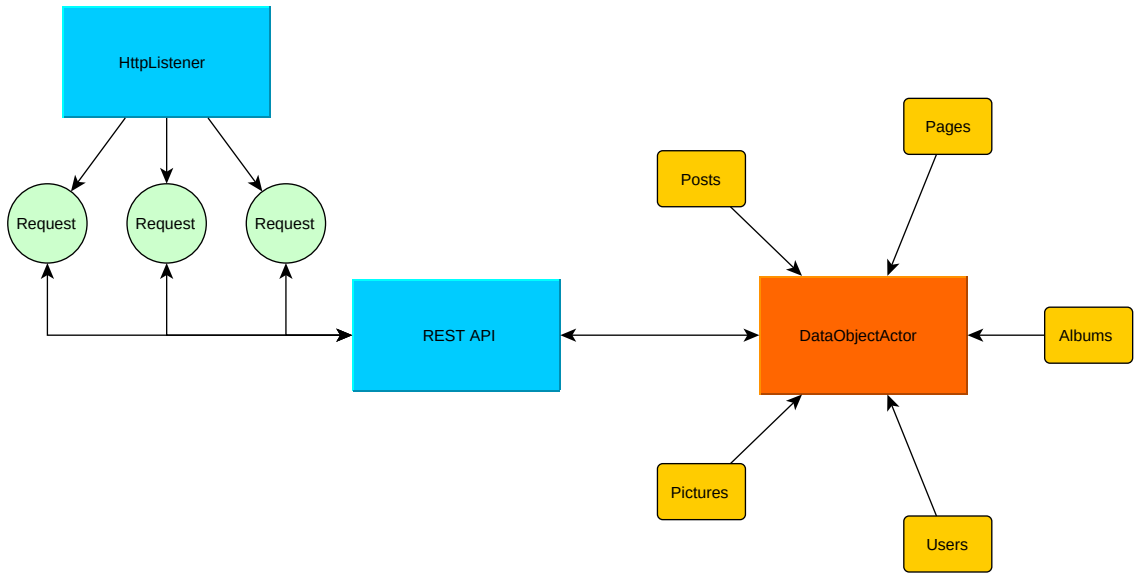

Figure 2: Server structure

One limitation of this approach is that all of the data objects are stored in one Actor instance. This makes programming easy, but since actors are single threaded and can only process one message at a time, this becomes a limitation. A solution to this would be to have a more generic way of storing data such that any Actor could manage a chunk of user data. Then the API frontend would have to message a router to figure out which Actor was responsible for the required objects. This would increase the parallelism of the API server.

For organizing data, all major objects are considered "entities" and all derive from a similar base class (Figure 3). This base class holds common attributes such as identifier and modified_time. For example, UserEnt which holds a user's profile information derives from FacebookEntity. Each specialized Ent has its own JSON serialization and deserialization routines which allow it to be communicated over HTTP.

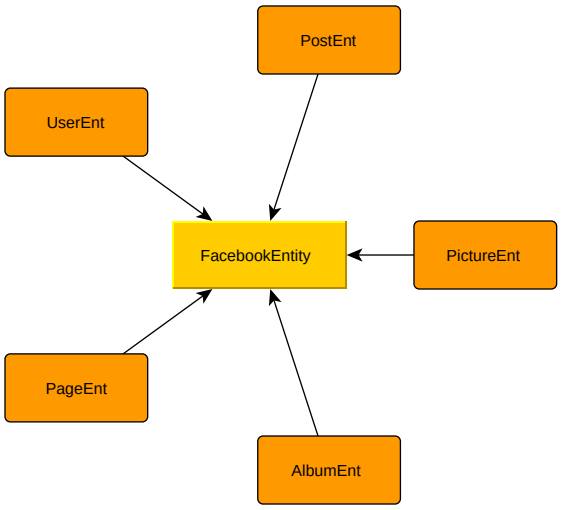

Figure 3: Entity hierarchy

REST API

Below is a simplified table describing the Facebook REST API.

Verb	Route	Description	Body
POST	/user/	Creates a new user with the parameters	UserCreateForm
GET	/user/{user-id}	Get a user by ID	UserEnt
POST	/user/{user-id}/add_friend/{f-id}	Add a friend from user-id to f-id (friend ID)	TextResult
GET	/user/{user-id}/friends	Get user-id's friends list	Array[Identifier]
GET	/post/{post-id}	Retrieve a post by ID	PostEnt
POST	/page/	Creates a new page	PageEnt
GET	/page/page-id	Get page by ID	PageEnt
POST	/album/	Creates a new album	AlbumEnt
GET	/album/album-id	Get album by ID	AlbumEnt
POST	/picture/	Creates a new picture	PictureEnt
GET	/picture/picture-id	Get picture by ID	PictureEnt

Table 1: Facebook REST API

There are more API routes in the actual server, but Table 1 has a good sample of the routes.

User Studies

We generated our users based on these user studies. They are a combination of actual Facebook studies and studies on the general population.

Age

Age	Percentage
13 - 17	0 percent
18 - 24	15 percent
25 - 34	29 percent
35 - 44	24 percent

Table 2: Age Study [1]

Relationship Status

Relationship Status	Percentage
Single	37 percent
Married	31 percent
In a Relationship	24 percent
Engaged	3 percent
It's Complicated	3 percent

Table 3: Relationship Status Study [2]

Political Affiliation

Party Affiliation	Percentage
Republicans	25 percent
Democrats	29 percent
Independents	41 percent

Table 4: Political Affiliation Study [3]

Interested In

Interested In	Percentage
Straight	96.6 percent
Gay/Lesbian	2.4 percent
Bisexual	0.7 percent

Table 5: Political Affiliation Study [4]

Facebook Activity

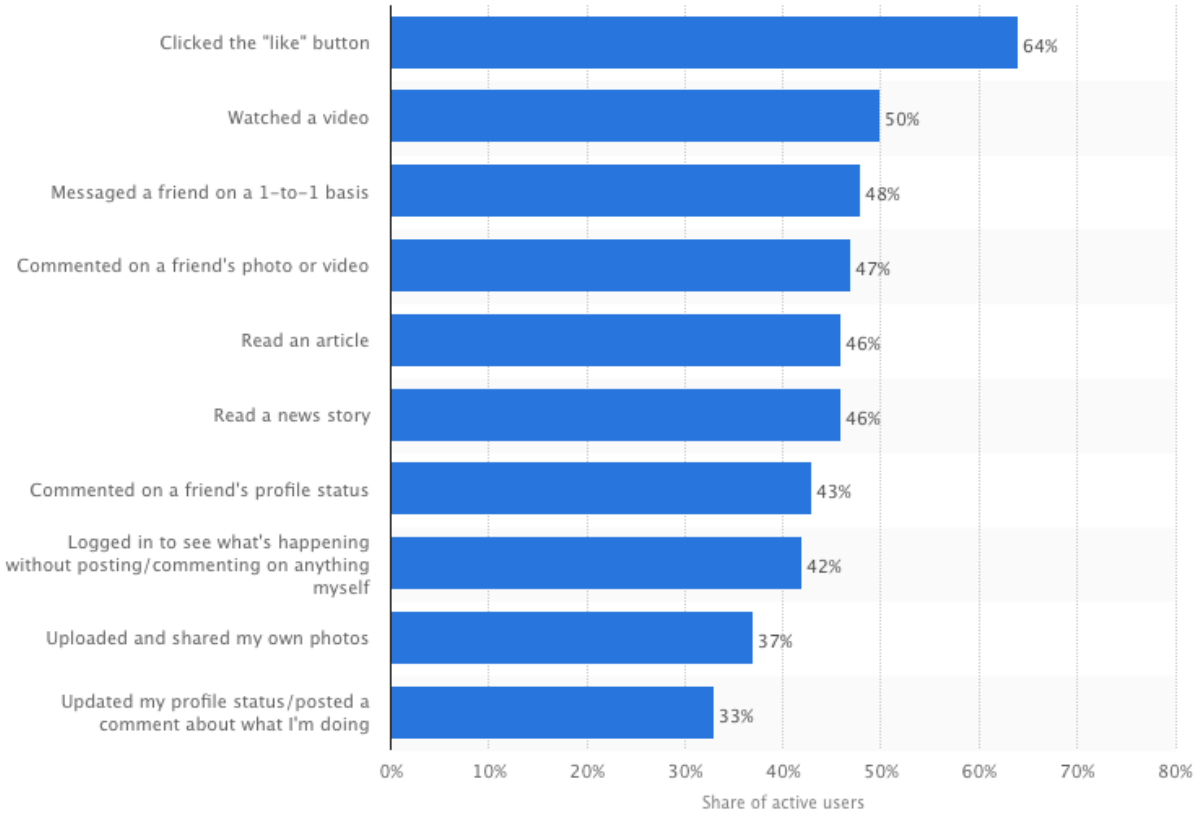

Figure 4: Percentage of Activity for Facebook Users. [8]

Worldwide, there are over 1.49 billion monthly active Facebook users [10]. In Europe, over 307 million people are on Facebook. On average, the Like and Share buttons are viewed on 10 million websites daily. Five new profiles are created every second, with 76 percent of Facebook users female and 66 percent male. With every minute on Facebook, 510 comments are posted, 293,000 statuses are updated, and 136,000 photos are uploaded.

In a study conducted from 2007 to 2010 [11], the three biggest usage spikes of Facebook occurred on weekdays at 11:00 AM, 3:00 PM, and 8:00 PM. Facebook users are less active on Sundays compared to all days in the week.

Testing

For testing, our primary metrics were the number of users simultaneously using the API and the number of sustained API requests per second. To show how the number of requests per second (hereafter r/s) increased, we used 4 clients on separate servers, each managing the same number of users. This means that each client was running the total number of users divided by 4. The plots showing the r/s over time are below.

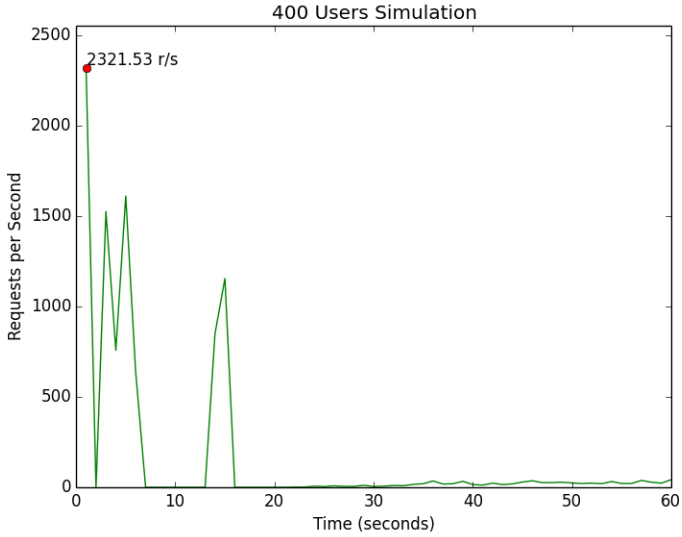

Figure 5: 400 user simulation

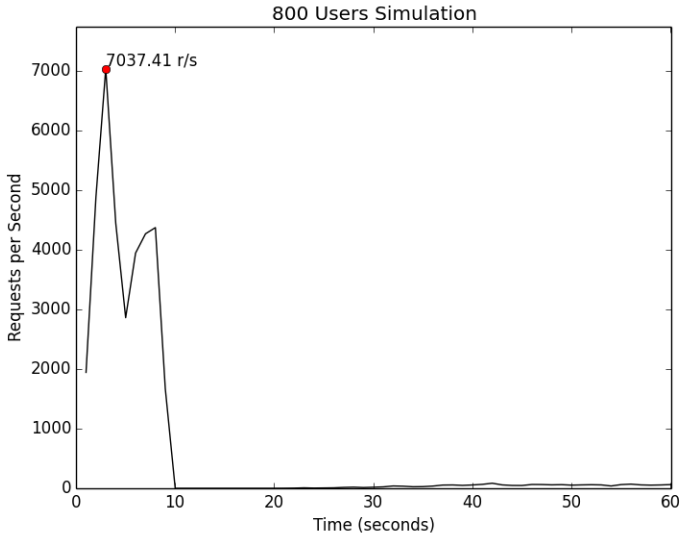

Figure 6: 800 user simulation

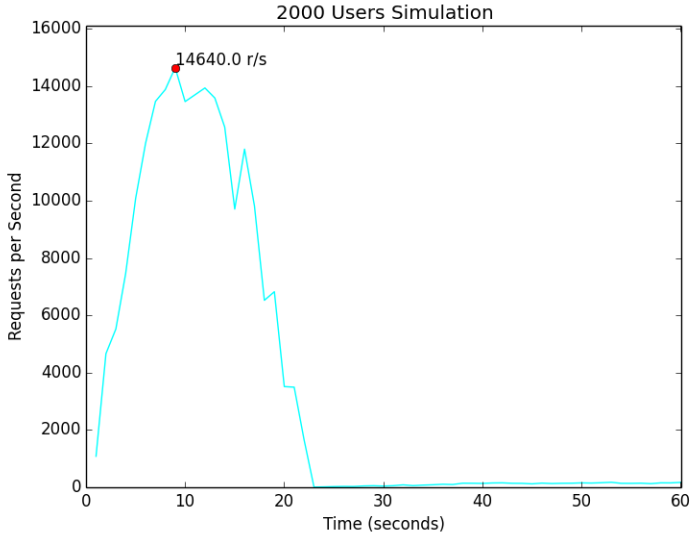

Figure 7: 2000 user simulation

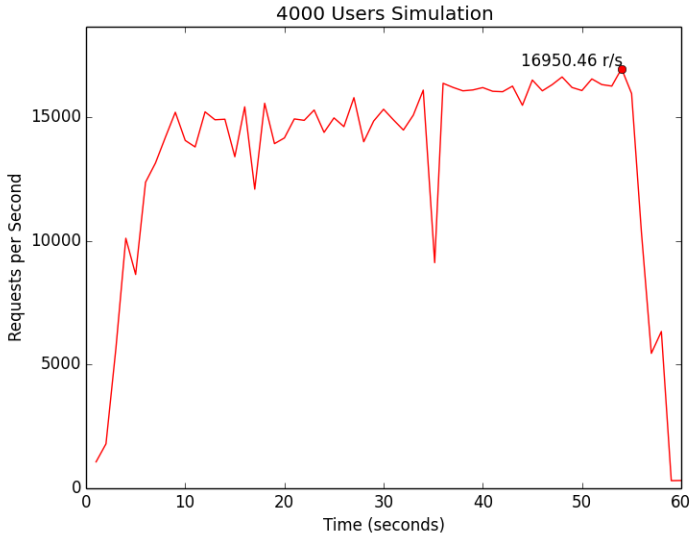

Figure 8: 4000 user simulation

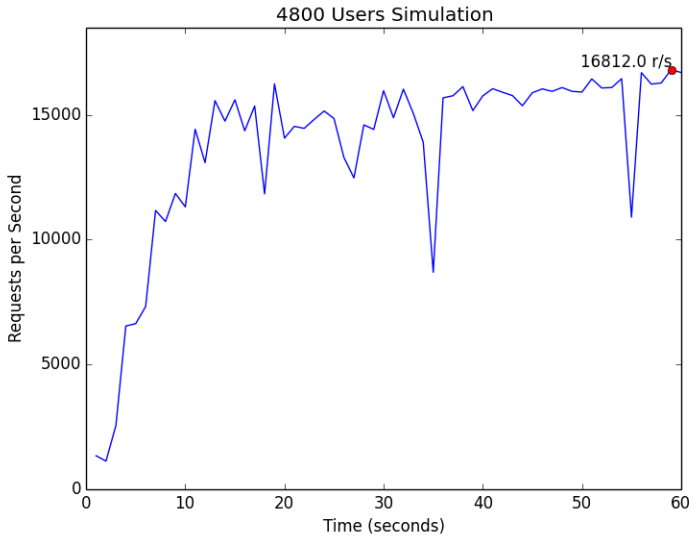

Figure 9: 4800 user simulation

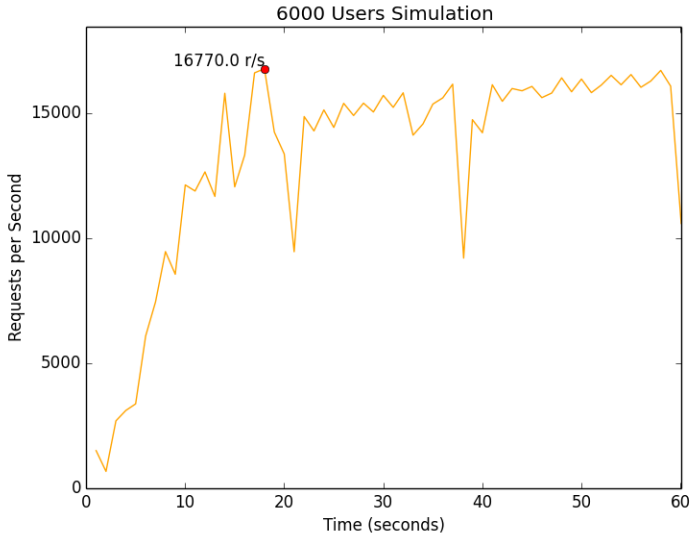

Figure 10: 6000 user simulation

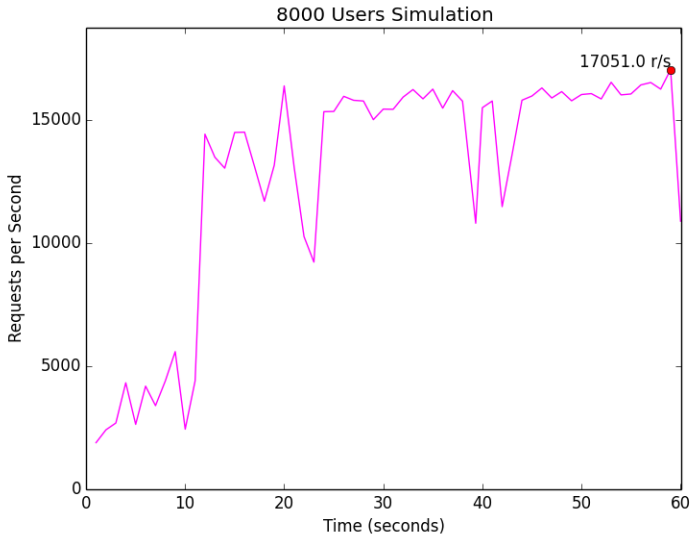

Figure 11: 8000 user simulation

As you can see, the max r/s we achieved was about 17,000. This appears to be a limit of our API server. On an 8 core machine with 2000 users, we were seeing about 370% CPU utilization. This should have been close to 800% if we were fully parallelized. When hitting this limit, we noticed that client connections to the API server were timing out. This means the Spray-HTTP listener was not able to keep up with the connection requests or connections were being buffered. Additionally, the API server used one Actor to manage the Facebook data access. This is limiting due to how Akka messages are handled. For a future project, a different, more scalable architecture would have to be created.

References

[1] Social Media Demographics to Inform a Better Segmentation Strategy,
`http://sproutsocial.com/insights/new-social-media-demographics/`

[2] Facebook Relationship Status Statistics,
`http://www.statisticbrain.com/facebook-relationship-status-statistics/`

[3] Party Affiliation,
`http://www.gallup.com/poll/15370/party-affiliation.aspx`

[4] What percentage of the U.S. population is gay, lesbian, or bisexual?,
`https://www.washingtonpost.com/news/volokh-conspiracy/wp/2014/07/15/what-percentage-of-the-u-s-population-is-gay-lesbian-or-bisexual/`

[5] Most Popular Girl Names in 2014,
`http://www.babycenter.com/popular-baby-girl-names-2014`

[6] Most Popular Boy Names in 2014,
`http://www.babycenter.com/popular-baby-boy-names-2014`

[7] Most Popular Last Names,
`https://en.wikipedia.org/wiki/ListofmostcommonsurnamesinNorthAmerica`

[8] Most popular activities of Facebook users worldwide as of 3rd quarter 2015
`http://www.statista.com/statistics/420714/top-facebook-activities-worldwide/`

[9] Watchout4Snakes,
http://watchout4snakes.com/wo4snakes/Random/RandomPhrase

[10] The Top 20 Valuable Facebook Statistics,
https://zephoria.com/top-15-valuable-facebook-statistics/

[11] When are Facebook Users Most Active?,
http://mashable.com/2010/10/28/facebook-activity-study/7tNySamgEkqt