Performance Evaluation of UNSWBook

Callum Bannister   Jerome Samir   Peter Bishay   Aaron Quan
Waseem Sajeev

October 22, 2017

Contents

1. Introduction

Maintaining quality of service under heavy usage is essential for any web application that expects a large amount of concurrent users. As there is currently no demand for an asocial media platform, we have performed a series of experiments on a selection of functions of our UNSWBook application to obtain data on performance degradation as the number of simultaneous users increases. We present the methodology; what functions were tested and what hardware were they tested on. Then, we present results of the experiment, an analysis of the results, and the conclusions to be drawn.

2. Application Details

We looked at four basic functionalities of UNSWBook for performance under pressure. They are as follows:

1. User login
2. Searching for friends
3. Adding a friend
4. Posting a message

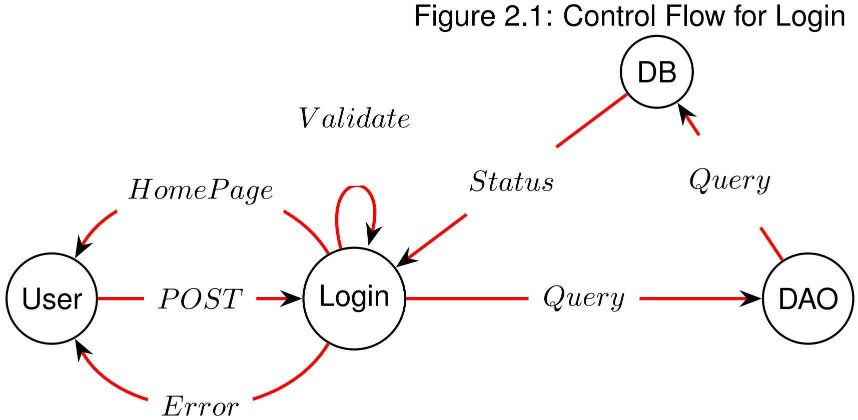
For each of these functions, we will explain the implementation and what database interactions were required, in order to give context to the performance data in Chapter 3.

2.1 User Login

The user submits their username and password with a POST request. The request body is passed to the *Login* servlet's doPost method, where we perform initial validation of the input, checking that the required fields exist.

Then, we construct the DAO object for a User. We make a single query to the database to check that a non-admin user with the name and password exists, and we retrieve their unique id and whether or not they are banned. Then, we route the user to either the login page, or their home page, returning an error message where necessary.

Figure 2.1: Control Flow for Login

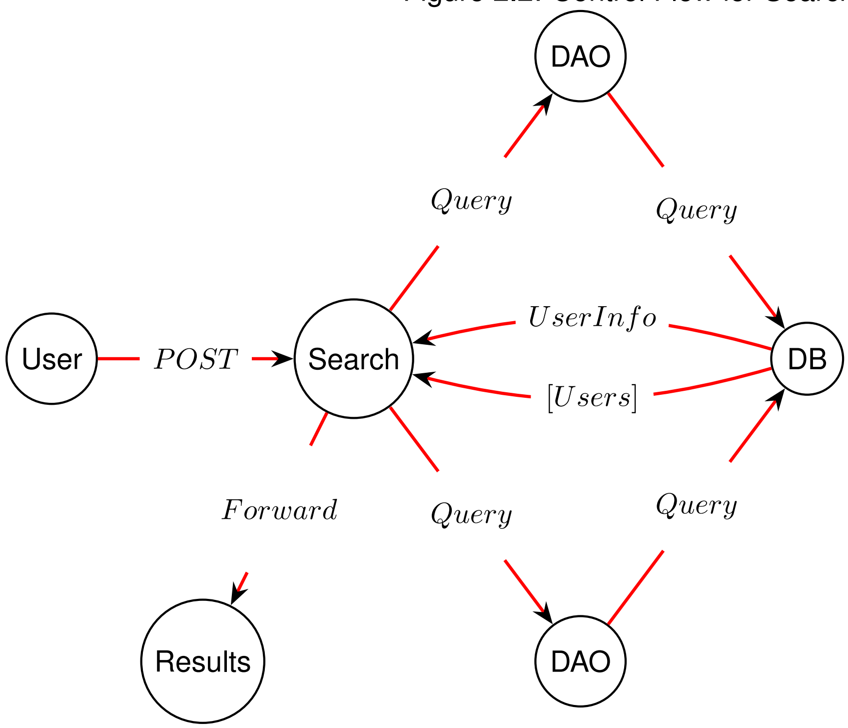


2.2 Search

In this section we describe only the basic search functionality. We expect that the advanced search to be used relatively infrequently, so optimisation of that function would be of secondary priority.

As before, the user submits the search query information via POST, getting routed to the appropriate method in the *Search* servlet. First, we construct a DAO object for the current logged in user, querying the database to obtain their information. Then, we construct a second DAO object, and use it to retrieve a list of users whose username contains the submitted string. We set this as an attribute, and forward the request onto $results.jsp$.

Figure 2.2: Control Flow for Search

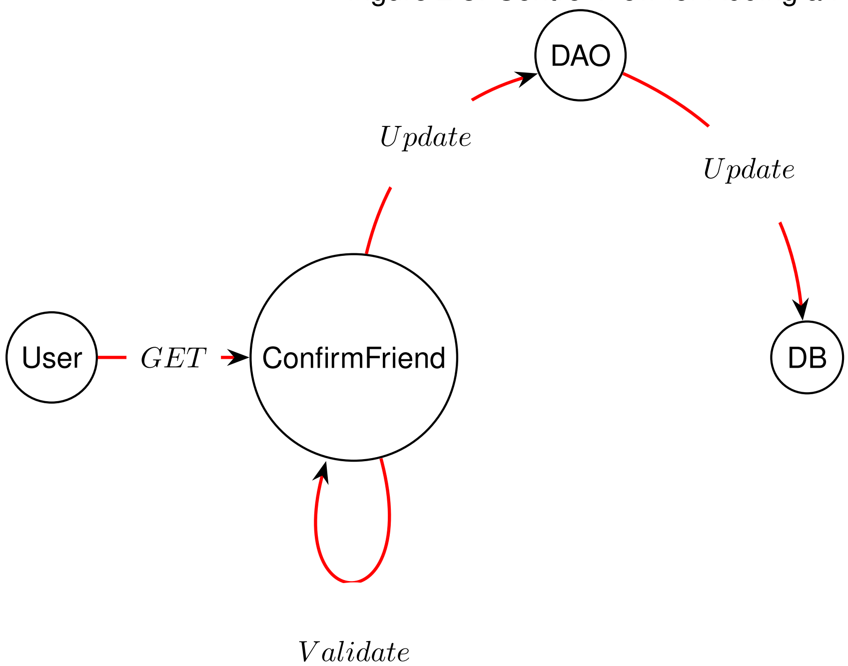


2.3 Adding a friend

Although adding a friend consists of two operations, the need for a confirmation email to link the two makes measuring them as a conjoined unit unrealistic. Instead, we present only the accepting of a friend request.

The friend acceptance submits the user id for the requester and the requestee, dispatched to the *ConfirmFriend* servlet. As was the case for *Login*, we perform simple validation of the parameters. Then, we update a table in the *friends* database, setting the confirmed flag to $true$, requiring a second query.

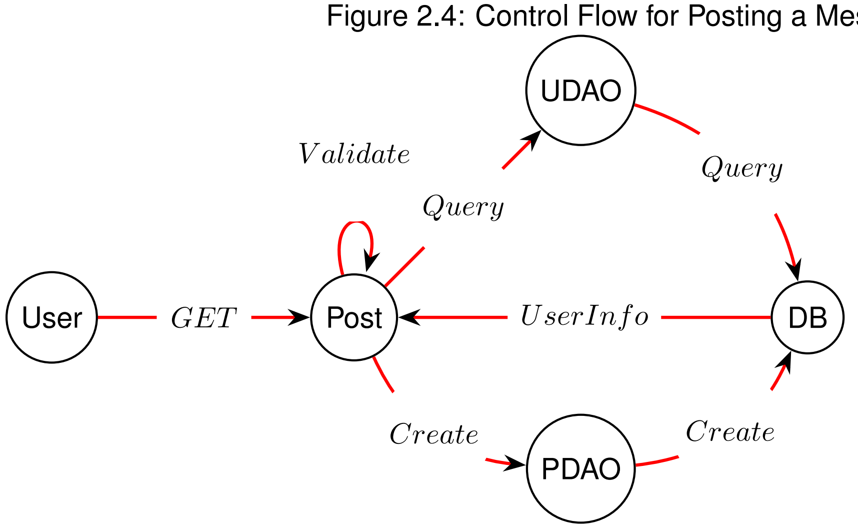
Figure 2.3: Control Flow for Adding a Friend



2.4 Posting a message

Unlike the other operations, the user posts a message by sending a GET request. This is routed to the *Post* servlet, where we once again perform simple validation on the input. We construct the DAO object and look up the currently logged in user as before. We then create a *post* DAO object, and use it to create the appropriate table in the database.

Figure 2.4: Control Flow for Posting a Message



3. Test Setup

In this chapter we present the hardware, setup, and experimental procedure that we used to obtain the data.

3.1 Server

The application server had the following hardware specifications:

Table 3.1: Server Hardware

Hardware	Specification
HDD	128GB
RAM	8GB
CPU	2.2 GHz Intel Core i7

The machine was running Windows 10, with JVM Version 1.7.0. We used Tomcat 7.0.42 as our server, and Apache Derby 10.9.1 as our database. To maximise the accuracy of our results the only user processes on the machine were the Tomcat and Derby servers.

The connection pool size was kept at the default for the jdbc driver (a *maximum* of 100 and *minimum* of 10). We did not predict that manipulating this value would lead to significantly different results.

3.2 Client

The client had the following hardware specifications:

Table 3.2: Client Hardware

Hardware	Specification
HDD	128GB
RAM	8GB
CPU	2.2 GHz Intel Core i7

As with the server machine, extraneous processes were kept to a minimum. Both the client and server were on the same LAN, minimising factors unrelated to the performance of UNSWBook from affecting the results.

3.3 Experimental Method

We constructed our experiment to simulate the following scenario:

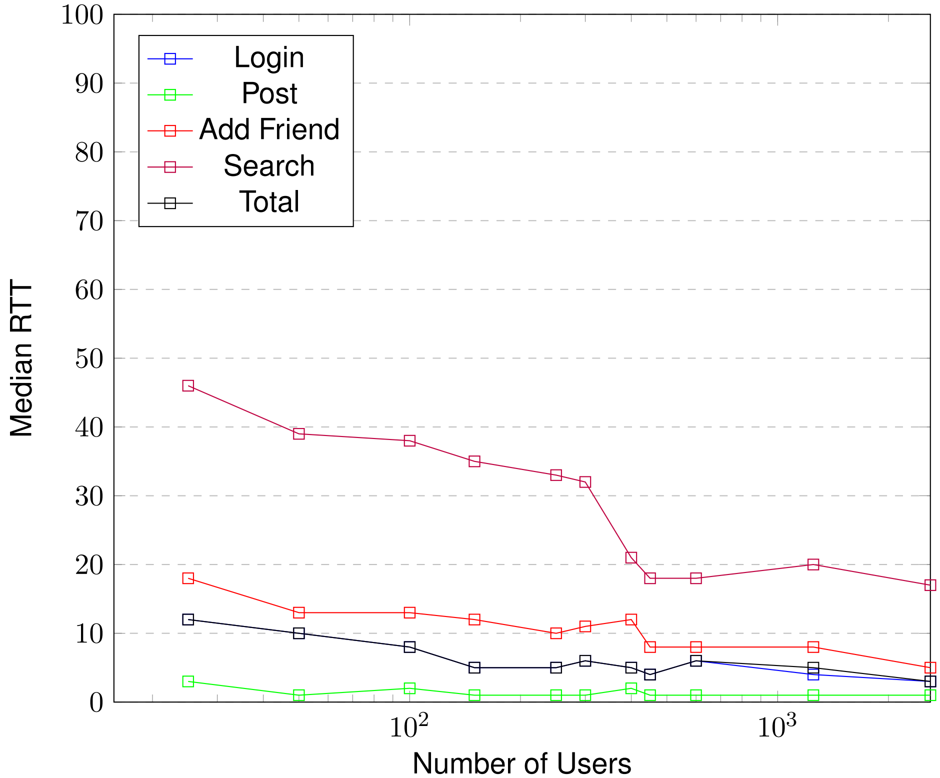
1. A user logs in, searches for a new friend, adds them, and posts a message to acknowledge their new friendship

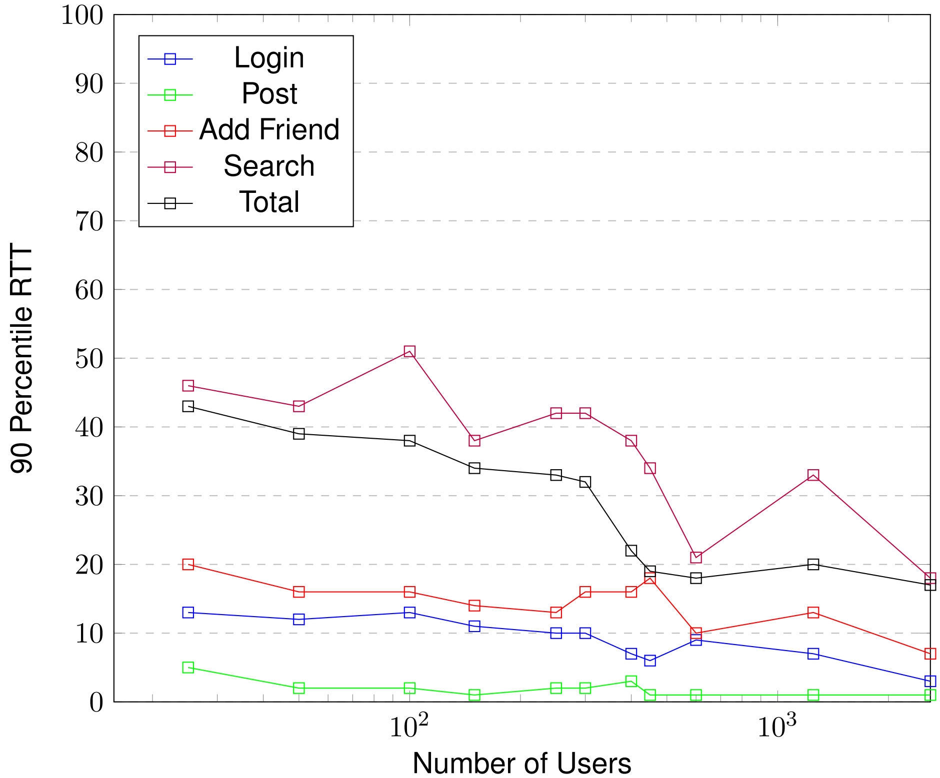
We added a delay of five seconds between each user action. We tested this scenario for increasing numbers of concurrent users: 25, 50, 100, 150, 250, 300, 400, 450, 600, 1250, 2600, 3000. Before each test attempt, we cleared the database of any existing posts and/or friends, in order to keep those factors constant.

However, at 3000 users, we started to receive a large number of errors when attempting to search. We were unable to determine the cause of these errors.

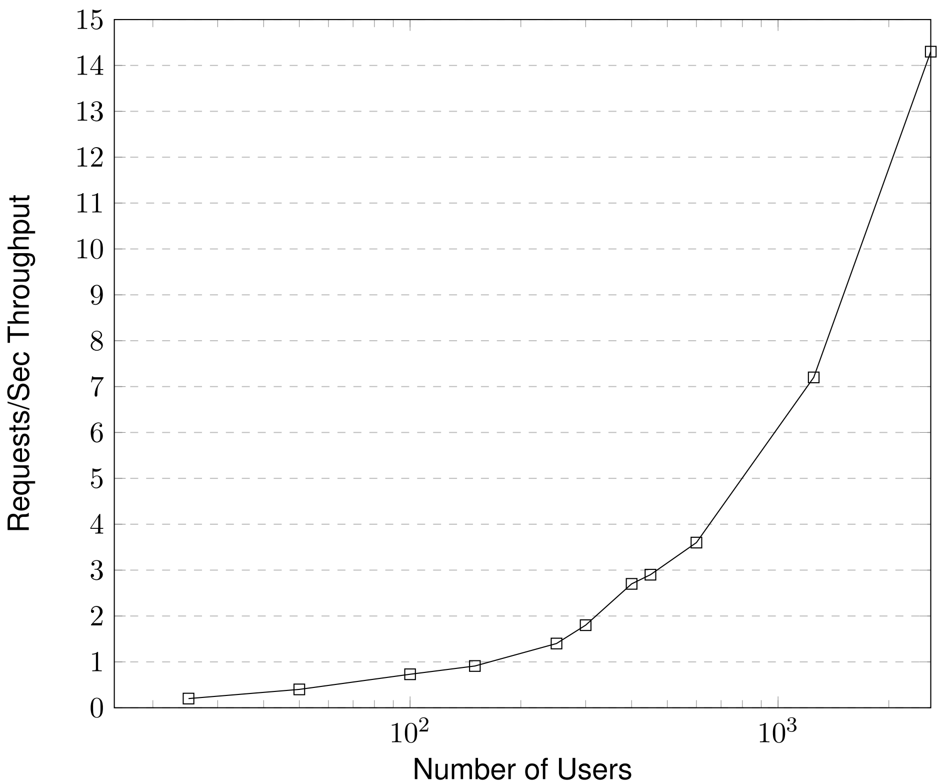
4. Results

4.1 Data





Login
Post
Add Friend
Search
Total
90 Percentile RTT
Number of Users



Requests/Sec Throughput
Number of Users

Table 4.1: Number of Requests by Little's Law

Users	Average RTT (ms)	Throughput (requests/sec)	Requests
25	83	0.2	0.0166
50	13	0.4	0.005
100	13	0.73	0.01
150	13	0.91	0.153
250	10	1.4	0.014
300	11	1.8	0.019
400	9	2.7	0.0243
450	8	2.9	0.028
600	7	3.6	0.0252
1250	7	3.6	0.0252
2600	5	14.3	0.0715

4.2 Analysis

For the bounds we were able to test, we noticed no degradation of the application performance. Up until 3000 users, the RTT for all operations maintained relatively constant. The throughput behaves as expected.

We observed that searching was by far the slowest operation out of those we tested. This is unsurprising, as all other functions only need to reference a single table at a time, whereas searching returns a list of results. It would also run into performance issues as the size of the data stored increased (although we did not test it here).

We found adding a friend to be the next most expensive function. This, again, is unsuprising, as it requires an update, which is slightly slower than a simple select. We believe this suggests that if we were to run into performance issues, optimising the data access layer would achieve the best results.

We suggest that part of the performance of the application is due to the relatively small amount of data stored in UNSWBook currently. As we said before, the data access layer appears to be a relative bottleneck, so performance would degrade faster if the database calls were more expensive either due to relatively larger amounts of data being stored, or, in a realistic scenario, where the server and database were not hosted on the same machine.

5. Conclusion

We were able to gather some useful data on deploying UNSWBook under a realistic load environment. While the performance of our application held up, we ran into errors at 3000 users. We discovered that the best place for optimisation of our web application was data access.

`https://www.youtube.com/watch?v=9SMNyWMRqjU&feature=youtu.be`