Erlang Project Report

Gakuo Patrick

April 5, 2019

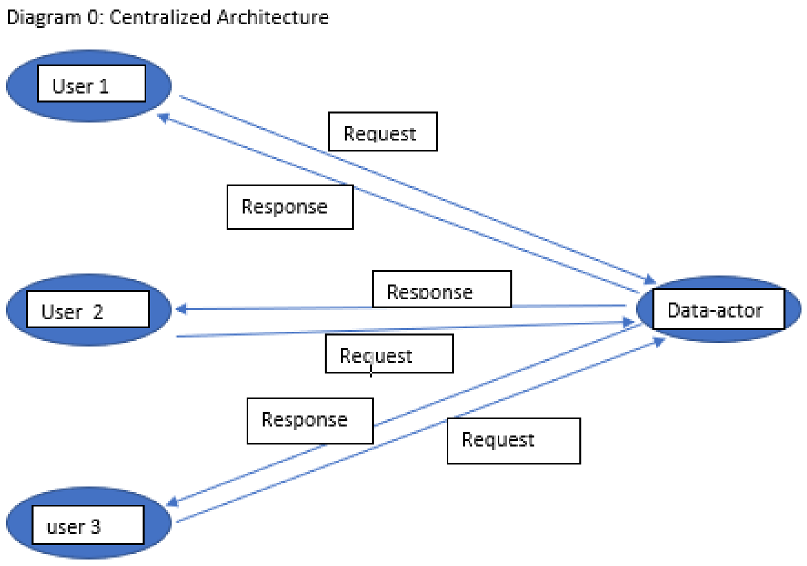
1 Introduction

In this project we set out to implement a distributed Erlang SnapChat-like application. We are provided with a centralized implementation which we then implement in parallel and benchmark as described in this report. In the centralized set up, all users communicate with the same server(data actor). In the distributed set up, the registration of users uses one central server. The registration server assigns each user its server which it communicates with from then on-wards. The application does various activities which include registration, posting stories, getting stories, fetching the user homepage, befriending other users, and registration. To benchmark the two setups with use metrics such as the number of users in the application, the number of friends per user, the number of stories each user posts, the expiration time of stories, and the number of homepage requests each user makes. We then determine the speedup achieved by decentralizing the application.

2 Implementation

a. Centralized architecture

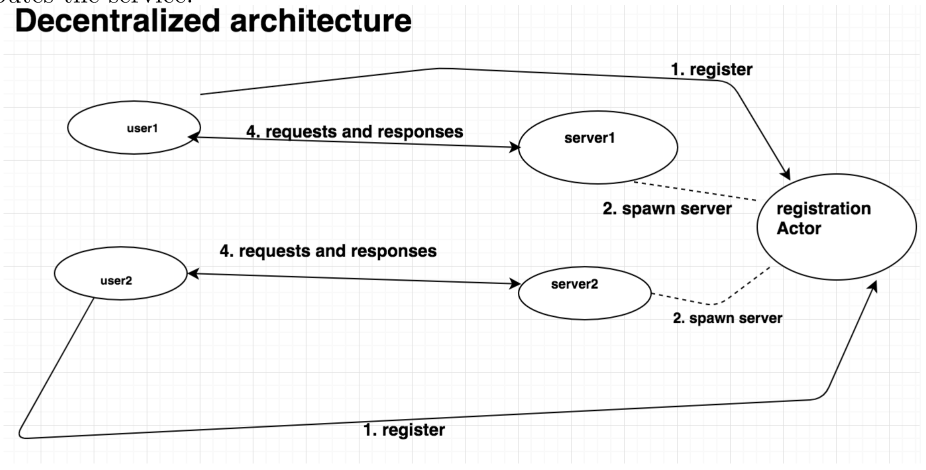
The first implementation of the application has one server waiting for user requests and responding to them as shown below.



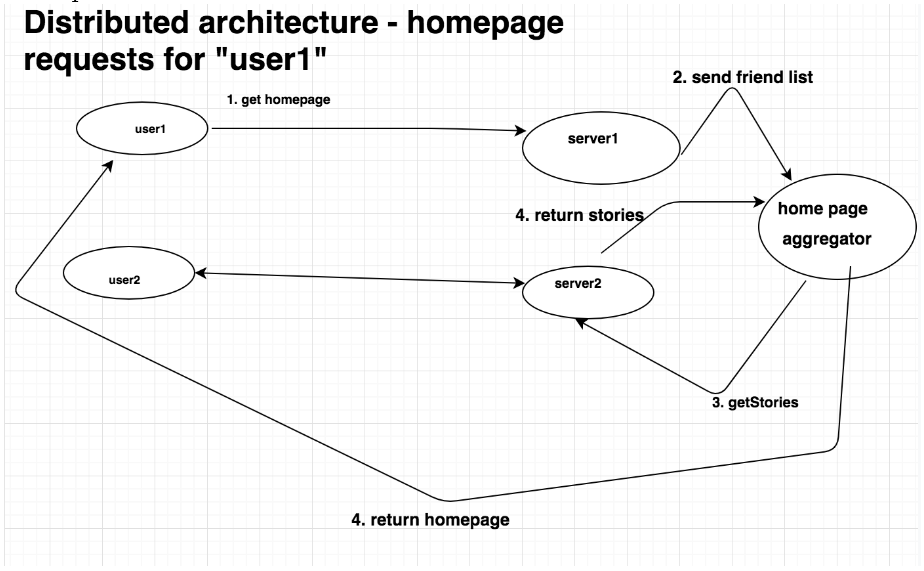
Requests include: registration,making a story, getting stories and getting homepage. The data_actor responds with corresponding messages to each of these requests. Registration returns a user identity, getting stories returns stories posted only by the user while homepage requests returns stories from the user's friends. Friends requests involve adding the befriended user to the set of the user friends and returning "ok".

b. Distributed architecture

This registration actor creates a new process(worker process) for each user on registration. Subsequent requests by this user go to to this process which distributes the service.



The get homepage uses a rather different approach. On getting a homepage request from its user, a server process spawns a new process to fetch all the requests of the users' friends. It aggregates them and sends them back directly to the requester.



2.1 c. Scalability

The centralized architecture uses a sequential approach to process user requests. All user requests go to one mailbox and they are executed one at a time. If the number of requests is increased, this implementation will have delayed responses as requests are handled one at a time. As such the central server is a bottleneck.

In the distributed architecture, we do away with the sequential processing and assign each user a server process to handle its requests. If the number of requests increases, this architecture responds by spawning more processes for the new requests. As a result, even under a high workload, the system is still responsive as these processes run in parallel.Registration of users goes through one server which can be a bottleneck.

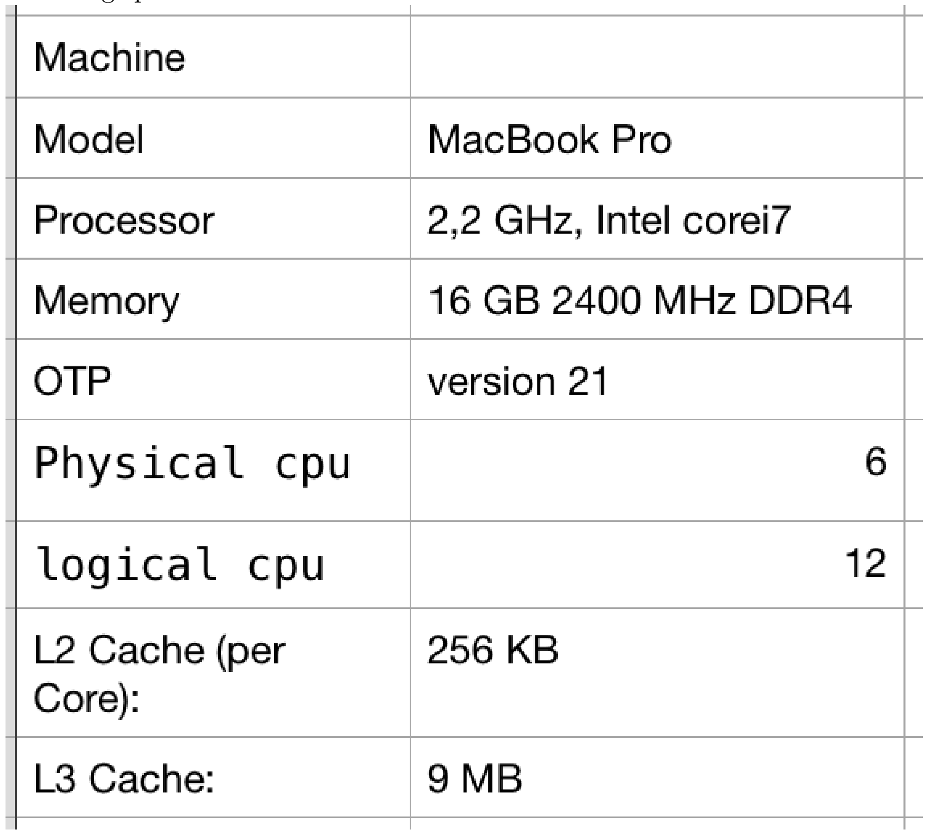
2.2 d. Consistency

The centralized architecture ensures consistency at the expense of scalability. It has one central consistent view of the data which is encapsulated in the data_actor. This data includes all users and their respective details(userId, stories, and sets of each user's friends). When a user makes a request to access the tweets, the user gets all the tweets made before the request. A user can access data available in the system in a timely fashion as the data_actor captures the entire state changes sequentially making all requests ordered in a happens before fashion.

The distributed architecture, on the other hand, stores each users data on a different process.However, homepage requests are sent to each friend and there is no inconsistency. We also store a bidirectional friend relationship which further makes the distributed implementation consistent. Relaxing these elements would make the data less consistent and the implementation faster than it already is.

3 Evaluation

As aforementioned, in the experiment we set up two versions of the SnapChat-like application, the centralized and the distributed architecture. We also use separate benchmarks for each set up. Both experiments run a platform with the following specifications.



Machine	
Model	MacBook Pro
Processor	2,2 GHz, Intel corei7
Memory	16 GB 2400 MHz DDR4
OTP	version 21
Physical cpu	6
logical cpu	12
L2 Cache (per Core):	256 KB
L3 Cache:	9 MB

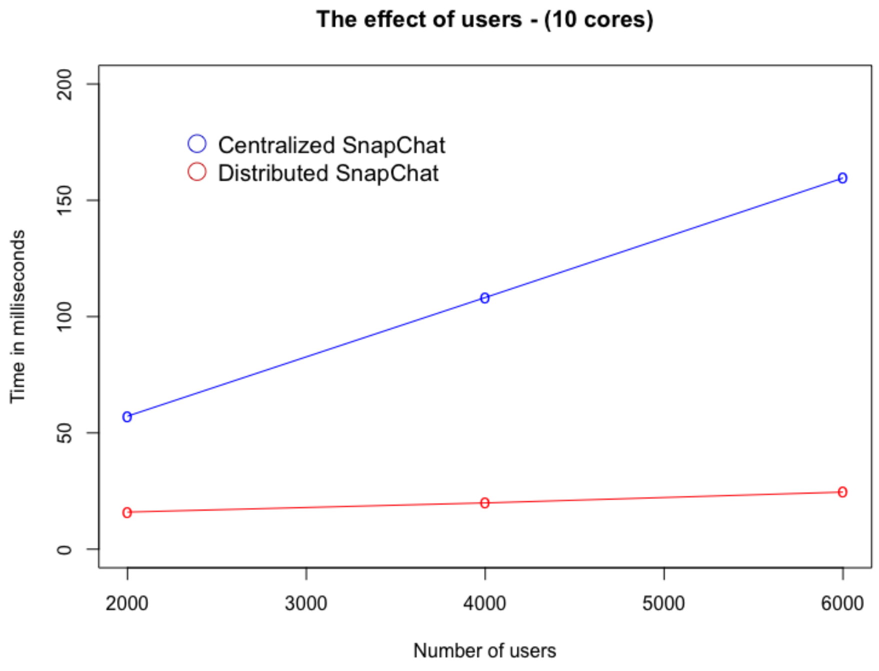
With these two application architectures running on the above platform, we give different types of requests. We vary the number of requests and/or parallelize or administer requests sequentially. We then compute line and bar plots of our observations accompanied by rigorous statistics to back up our findings.The aim of this experiment is to determine which configuration speeds up execution. As such we use the os:timestamp() and run the benchmark 33 times for each experimenterlangorg.

3.1 Benchmark setup

According to (Georges, Buytaert, & Eeckhout, 2007), 30 or more benchmark runs are sufficient to assume normal distribution while computing statistics. Additionally, it is advisable to do away with the first few benchmark runs as they are biased by the virtual machine initialization overhead georges2007statistically. We therefore take the last 30 of our 45 benchmark runs. We then report p-values and effect sizes as recommended by (Kalibera & Jones, 2013) who recommend this to ensure benchmark results are not misleading. We do our statistics in R-Studio. We use a t-test to see if the means of two data sets are significantly different. Note here that parallelizing requests means that for each request we spawn a process to send the request and wait for a response. Sequential requests are done from a single process.

3.1.1 The effect of users

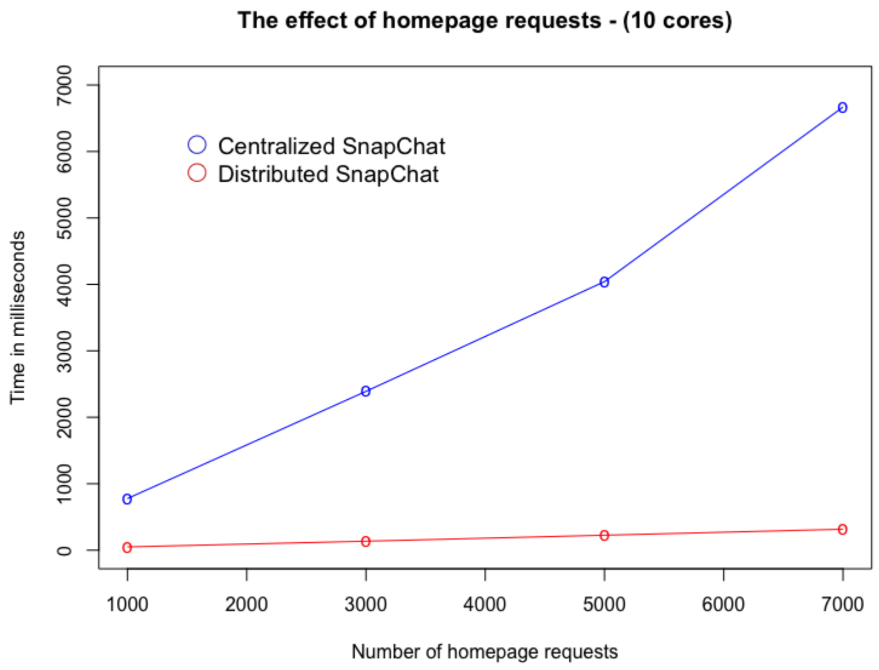
First, we looked at the effect of users on the overall performance of homepage requests. The benchmark was performed with 10 stories per user, 10 friends per user, and 500 homepage requests:



We observe that the distributed implementation scales gracefully with increasing users with an almost flat plot. The centralized implementation linearly gets slower with increasing load which is undesirable. We use normalization and geometric means comparison as recommended by (Fleming & Wallace, 1986). A t-test shows at a confidence level of 0.95 shows that the difference highly unlikely to be out of chance(we get a p-value of less ¡.001). This can be explained by the fact that the distributed architecture executes each homepage request in a different process unlike the centralized one that executes all requests in one process.

3.1.2 The effect of number of homepage requests

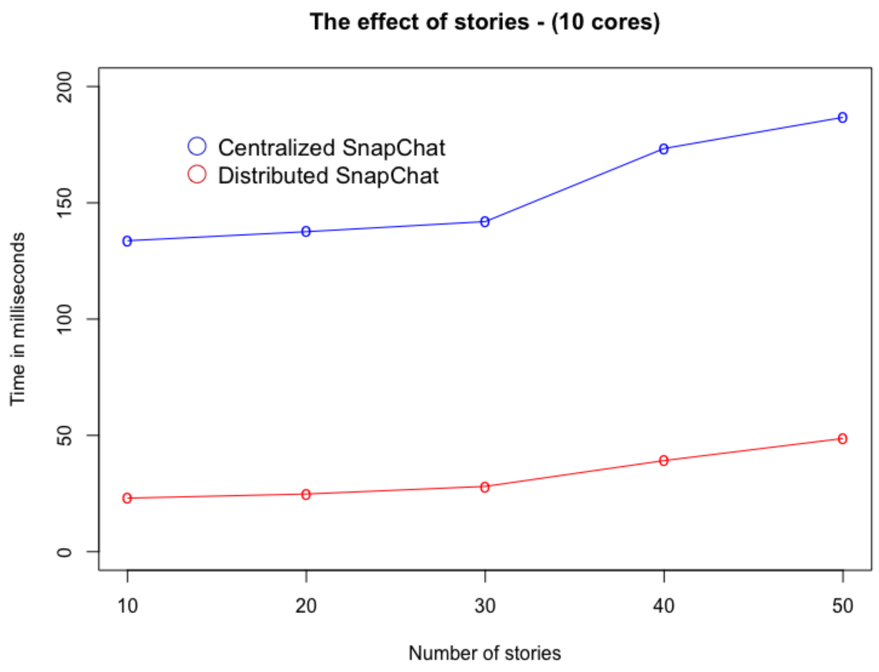
Secondly, we looked at the effect of the number of homepage requests on performance (homepage requests). The benchmark was performed with 10 stories per user, 10 friends per user, and 5000 users.



Again, we observe that homepage requests in the distributed architecture scale gracefully. The Plot shows the centralized implementation rise sharply after 5000 home page requests. This shows that our distributed implementation does exemplary well in distributing the homepage requests with the extra stories aggregation process as shown in the architectural description. The t-test also confirms that this difference in the two plots is not by chance.

3.1.3 The effect of stories

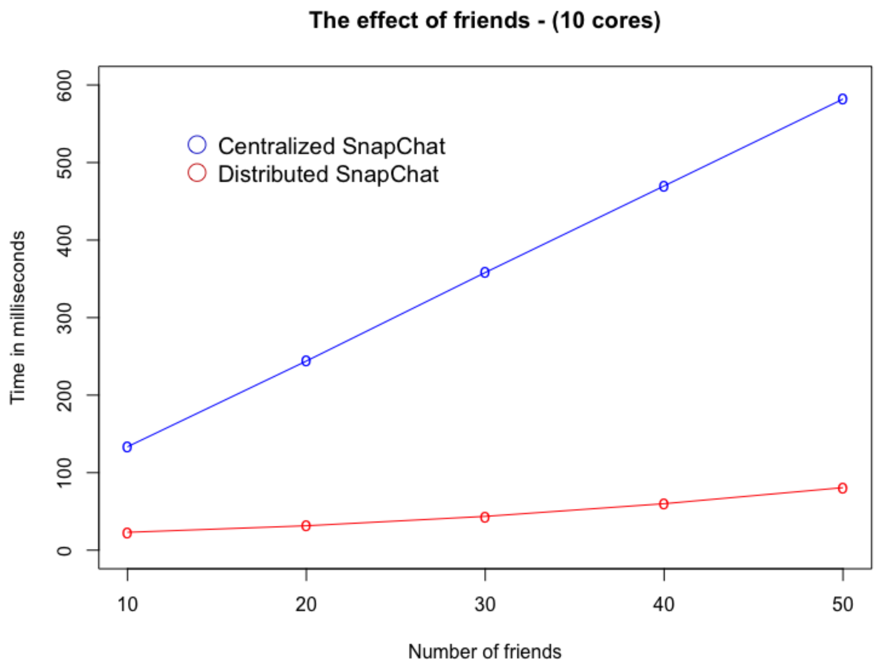
Thirdly, we looked at the effect of stories on the performance of the system (homepage requests). The benchmark was performed with 10 friends per user, and 500 homepage requests, and 5000 users.



We observe that as expected, homepage requests in the centralized architecture are much slower than in the distributed architecture. The centralized architecture does not scale as poorly as it did with the other metrics discussed above. The most prominent activity is compacting each users' stories which makes the execution more computation intensive as opposed to message intensive. Additionally, we note that after 30 stories, the distributed architecture also starts to respond with an almost linear increasing time. The slope has a low gradient which means the distributed server is not hard hit by increase in stories. The more the stories you have to fetch, the bigger the messages and so the bigger the penalty. A probable solution would be to fetch a maximum of 10 stories from each user. The difference between the two implementations is also not due to chance as confirmed by a t-test.

3.1.4 The effect of number of friends

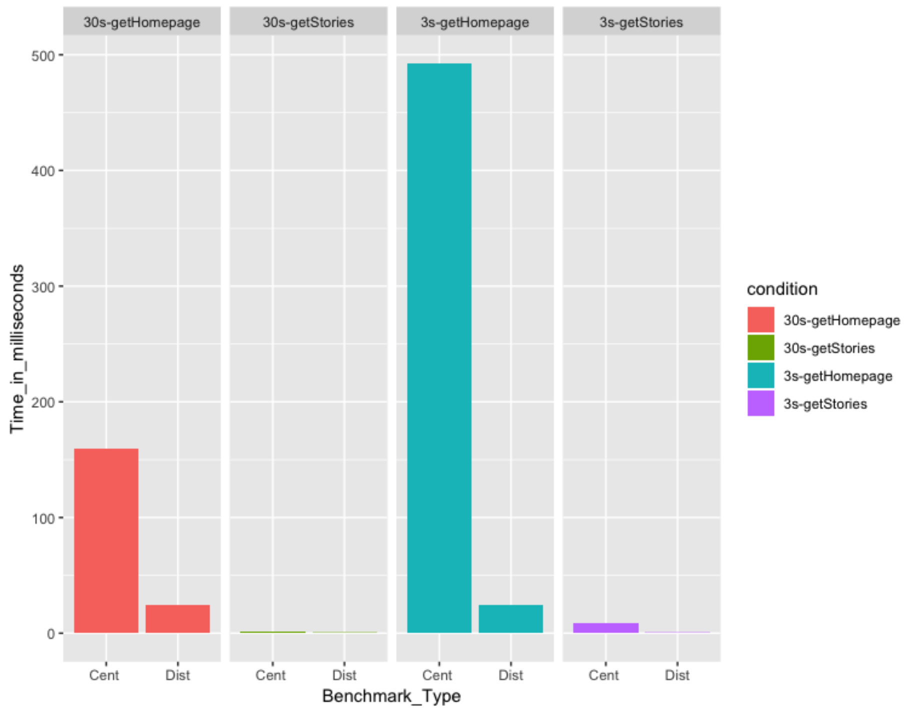
Fourthly, we looked at the effect of the number of friends on the homepage requests. The benchmark was performed with 10 stories per user, and 500 homepage requests, and 5000 users.



Friends have a similar effect on the decentralized architecture as the number of stories. This is explained by the fact that the more the stories the more the messages you fetch. However, the centralized implementation is hard hit by the number of friends as in this case sending out messages to get other users' (friends) stories becomes more prominent. The execution becomes more message intensive and because the centralized implementation is sequential, the mailbox becomes a bottleneck.

3.1.5 The effect of stories expiration duration

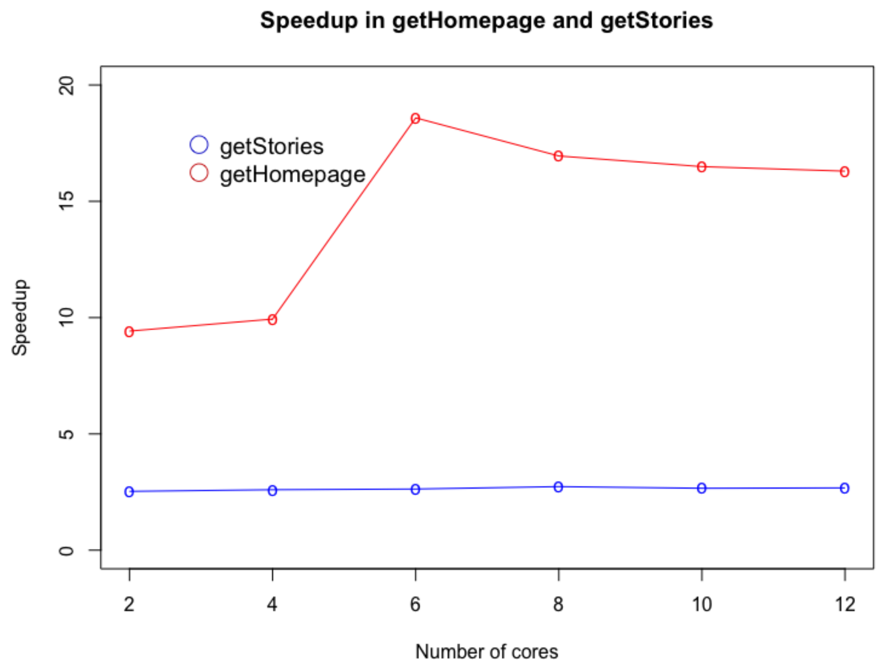
As a fifth benchmark, we looked at the effect of the stories expiration duration on performance. The set up had 6000 users, 10 stories per user, 10 friends per user and 500 homepage requests.



First, we observe that with lower stories expiration time, all requests are slower. Notably, the homepage request is worst hit. This is explained by the fact that low expiration duration flood the application with story destruct messages. Homepage requests are also message intensive and thus have to wait longer in the mailboxes. It is also observed that the centralized implementation is worse off than the distributed implementation as it has just a single mailbox where all messages queue.

3.1.6 Homepage request and stories request speedup

Finally, we investigated the speedup of our implementation. We investigated the number both homepage requests speed up and stories requests speed up.



We observe that the homepage speedup increases with the number of cores. On the contrary, getting stories does not speedup linearly. The homepage has a huge part that can be executed in parallel and thus the increased speed up with cores. The get stories has a constant part that can be executed in parallel and thus it does not respond linearly to increase in cores.

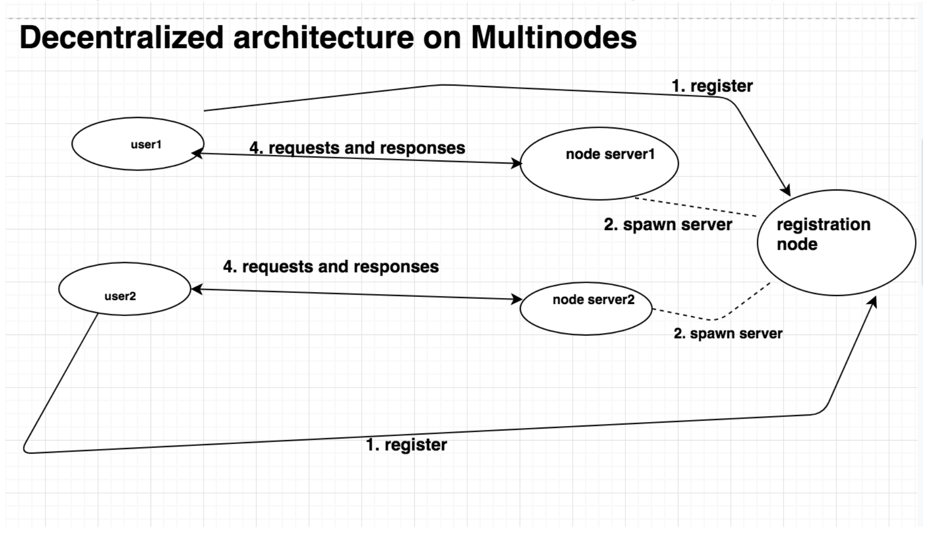
3.2 Throughput

For throughput we divide the total number of requests by their total execution time. We get that there are 60.16932 homepage requests per millisecond in the centralized setup. The distributed set up has a throughput of 419.7233 homepage requests per millisecond.

4 Insight questions

4.1 Question 1

To move from a multi-core to a multi-node environment, the architecture below is applicable. The registration node is responsible for spawning new server processes in other nodes. A bigger concern would be dealing with slower network connections and failures. This would require a more stringent fault tolerant implementation and less messages sent over the network to reduce network latency. A possible way around reducing network traffic is keeping information about users. Users might be in the same network and we will not want to send to messages to the same node. Collecting messages headed to the same node and sending them all at once will reduce network flooding and latency.



4.2 Question 2

In Erlang, spawning a process is a light process and as such it has been possible to spawn many processes at low costs to take advantage of extra cores on the machine. These processes run concurrently without high creation and synchronization costs making the distributed solution for this experiment scale gracefully with increased user requests. If this was otherwise, the spawning of processes would have added on to the execution overhead resulting into problems with scalability. That is the more processes we spawn, the slower the application becomes. In this case, it would be wise to limit the number of processes spawned so as to have useful work done in one processes as opposed to waste resources on spawning tasks.

References

Fleming, P. J., & Wallace, J. J. (1986). How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, *29*(3), 218–221.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, *42*(10), 57–76.

Kalibera, T., & Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Acm sigplan notices* (Vol. 48, pp. 63–74).