Load-Balancing Clients-Server Connections During Membership Change

Alexander Shraer
Yahoo! Research
shralex@yahoo-inc.com

Benjamin Reed
Yahoo! Research
breed@yahoo-inc.com

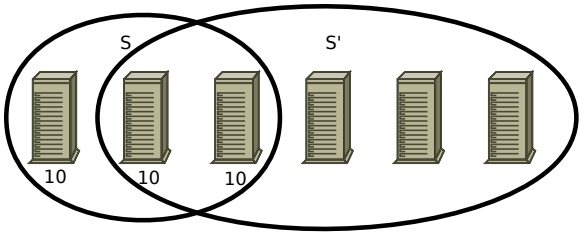

Figure 1. A perfectly balanced service (10 clients are connected to each of the servers) about to move to a new configuration S'.

Figure 1 shows a balanced service with configuration S that is about to move to S'. There are 30 clients in the system and each of the three servers in S serves 10 of the clients. When we change to S' we would like to make sure the new system is also load-balanced. In this example this means that each server should service 6 clients. We would also like to move as few clients as possible since session reestablishment puts load on both the clients and the servers and increases requests latency for client requests issued while the reestablishment is in process. A final goal is to accomplish the load balance using only logic at the clients so as not to burden the servers.

We are able to accomplish these three goals using case specific probabilities that follow from our first goal, that is, we assume that S is balanced and make sure that the expected number of clients connected to each server in S' is the same[1]. We denote by M the set of servers that are in both configurations, $S \cap S'$. Machines that are in the old configuration S but not in the new configuration we will label O, that is, $O = S \setminus M$. Machines that are in the new configuration S' but not in the old configuration are labeled N, that is, $N = S' \setminus M$. We have two cases to consider:

[1] due to lack of space we do not present the derivation of these policies here

Case 1: $|S| < |S'|$ Since the number of servers is increasing, load must move off from all servers. Of course the load migration is not uniform since some of the clients connected to M will be able to retain their connection, whereas all clients connected to O will have to migrate. Note, however, that these clients should only migrate to N as servers in M have too many clients to begin with.

Rule 1. *If $|S| < |S'|$ and a client is connected to M, then with probability $1 - |S|/|S'|$ the client disconnects from its server and then connects to a random server in N. That is, the choice among the servers in N is made uniformly at random.*

Rule 2. *If $|S| < |S'|$ and a client is connected to O, then the client moves to a random server in N.*

Case 2: $|S| \geq |S'|$ The key observation to make in this case is that, since the number of servers decreases or stays the same, the load on each server in S' will be greater or equal to the load on each server in $|S|$. Thus, a server in M will not need to decrease load. This gives our first rule:

Rule 3. *If $|S| \geq |S'|$ and a client is connected to a server in M, it should remain connected.*

There are no clients connected to N since they are new servers, but there are clients connected to O that must move to a server in S'. The clients that move should be biased to move to servers in N since they start with zero load.

Rule 4. *If $|S| \geq |S'|$ and a client is connected to a server in O, it moves to a random server in M with probability $\frac{|M|(|S|-|S'|)}{|S'||O|}$, otherwise it moves to a random server in N.*

By having each client independently apply these rules, the system maintains balanced load in a distributed fashion.

Evaluation We performed our evaluation on a cluster of 50 servers. Each server has one Xeon dual-core 2.13GHz processor, 4GB of RAM, gigabit ethernet, and two SATA hard drives. The servers run RHEL 5.3 using the ext3 file system. We use the 1.6 version of Sun's JVM.

In this section, we would like to evaluate our approach for load balancing clients as part of configuration changes. To this end, we experiment with a cluster of 9 servers and

1000 clients. Clients subscribe to configuration changes using the *config* command and update their list of servers when notified of a change. In order to avoid mass migration of clients at the same time, each client waits for a random period of time between 0 and 5sec. The graphs presented below include 4 reconfiguration events: (1) remove one random server; (2) remove two random servers; (3) remove one random server and add the three previously removed servers, and (4) add the server removed in step (3).

We evaluate load balancing by measuring the minimum and maximum number of clients connected to any of the servers and compare it to the average (number of clients divided by the number of servers currently in the cluster). When the client connections are balanced across the servers, the minimum and maximum are close to the average, i.e., there are no overloaded or under-utilized servers.

Baseline Our first base-line is the current implementation of load balancing in Zookeeper. The only measure of load is currently the number of clients connected to each server, and Zookeeper is trying to keep the number of connections the same for all servers. To this end, each client creates a random permutation of the list of servers and connects to the first server on its list. If that server fails, it moves on to the next server on the list and so on (in round robin). This approach works reasonably well when system membership is fixed, and can easily accommodate server removals. It does not, however, provide means for incorporating a new server added to the cluster. In order to account for additions in this scheme, we replace the client's list with a new list of servers. The client maintains its connection unless its current server is not in the new list. Figure 2 shows that load is balanced well as long as we perform removals (steps 1 and 2), however when servers are added in steps 3 and 4 the newly added servers are under-utilized. In the beginning of step 3 there are 6 servers in the system, thus approximately 166 clients are connected to every server. When we remove a server and add two new ones in step 3, the clients connected to the removed server migrate to a random server in the new configuration. Thus, every server out of the 8 servers in the new configuration gets 21 additional clients on expectation (the newly added servers will only have these clients, as no other clients disconnect from their servers). In step 4 we add back the last server, however no clients migrate to this server. Although all clients find out about the change and update their lists, no client disconnects from its server as it is still part of the system.

A possible way to mitigate the problem illustrated in Figure 2 is to disconnect all clients from their current servers and re-connect to randomly chosen servers in the new configuration. This, however, creates excessive migration and unnecessary loss of throughput. Ideally, we would like the number of migrating clients to be proportional to the change introduced by the reconfiguration. In the extreme case that all servers in the configuration are replaced, all clients should migrate to the new servers, however if only a single server is removed (or added), only clients that were (or should be) connected to that server should need to migrate.

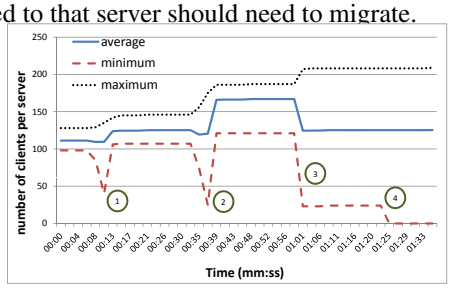

Figure 2. Baseline load balancing.

Consistent Hashing A natural way to achieve such limited migration, which we use as a second baseline, is to associate each client with a server using consistent hashing [1]. Client and server identifiers are randomly mapped to points in an m-bit space, which can be seen as circular (i.e., 0 follows $2^m - 1$). Each client is then associated with the server that immediately follows it in the circle. If a server is removed, only the clients that are associated with it will need to migrate by connecting to the next server on the circle. Similarly, if a new server is added a client migrates to it only if the new server was inserted between the client and the server to which it is currently connected. In order to improve load balancing, each server is sometimes hashed k times (usually k is chosen to be in the order of $log(N)$, where N is the number of servers). To evaluated the approach, we implemented it in Zookeeper. Figure 3 shows measurements for $k = 1$, $k = 5$ and $k = 20$. We used MD5 hashing to create random identifiers for clients and servers ($m = 128$). We can see that higher values of k achieve better load balancing. Note, however, that load-balancing in consistent hashing is uniform only with "high probability", which depends on N and k. In the case of Zookeeper, where 3-7 servers (N) are usually used, the values of N and k are not high enough to achieve reasonable load balancing.

Probabilistic Load Balancing Finally, Figure 4 shows measurements of load-balancing with the approach we have implemented in Zookeeper as outlined in Section **??**. Unlike consistent hashing, in this approach *every* client makes a probabilistic decision whether and where to migrate, such that the expected number of clients per server is the same for every server. As we can see from the figure the difference in number of clients between the server with the most clients and the least clients is very small. Using our simple case-based probabilistic load balancing we are able to achieve very close to optimal load-balance using logic entirely at the client.

References

[1] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. P. Abstract. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 1997.

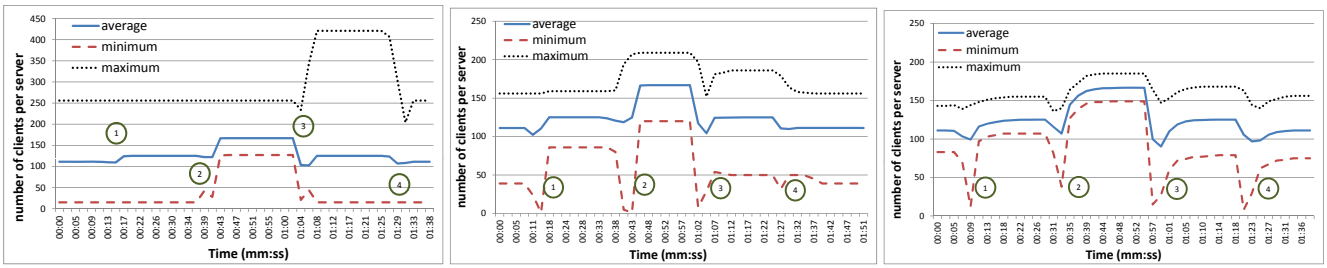

Figure 3. Load balancing using consistent hashing, with $k = 1$ (left), $k = 5$ (middle), and $k = 20$ (right).

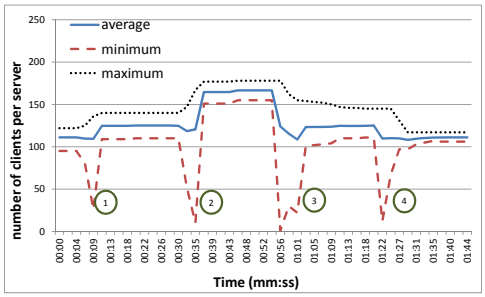

Figure 4. Load balancing using our method (Section **??**).