January 09, 2019

TECHNIQUES FOR EFFICIENT AND CONSISTENT HASHING WITH OPTIMAL FAIRNESS FOR FLOW DISTRIBUTION AND LOAD-BALANCING

Pierre Pfister

Mark Townsley

Yoann Desmouceaux

Aloys Augustin

TECHNIQUES FOR EFFICIENT AND CONSISTENT HASHING WITH OPTIMAL FAIRNESS FOR FLOW DISTRIBUTION AND LOAD-BALANCING

AUTHORS:
Pierre Pfister
Mark Townsley
Yoann Desmouceaux
Aloys Augustin

ABSTRACT

Techniques are described herein that improve software network load balancers by enforcing optimal fairness at the end of the computation of the consistent-hashing table. By employing the techniques described herein, upon reconfiguration of server backends, the number of impacted overhead flows will be very low, while still providing optimal fairness. Moreover, the described techniques are very fast to execute, and can run without a flow-table or a relatively large table. The described techniques may be a good candidate for hardware load-balancer and Equal Cost Multi-Path (ECMP) implementations.

DETAILED DESCRIPTION

ECMP is a widely deployed feature, but implementations can vary based on criteria used for flow-matching (e.g., destination address only, 5-tuple, etc.) and/or technique for path selection (e.g., hashing + modulo, consistent hashing, flow-table, etc.). Using a flow-table is typically costly in hardware, and can impact the TCAM space available to other features (e.g., the number of routes and/or firewall rules that can be used at a time). Stateless consistent hashing algorithms with good consistency are therefore key to ECMP performance.

One well known consistent hashing algorithm is a ring-based algorithm. In this algorithm, the flows and paths are placed on a 'ring', and each flow is sent on the closest path on the ring. Although this algorithm may provide optimal consistency, it does not distribute the flows equally between the different paths. This issue may be addressed by placing each path multiple times on the ring so as to decrease the variance (according to the central limit theorem), however, such a workaround comes at the cost of larger tables and/or more complex data-path algorithms.

Certain software network load balancers may address some of these problems by employing a consistent hashing algorithm for flow distribution (ECMP or load-balancing). In such arrangements, the algorithm presented herein provides optimal fairness with close-to-optimal consistency, while keeping the data structure and data-path small and simple.

For the purpose of the presentation of the algorithm introduced herein, it is assumed that **n_servers** servers (or paths) are available to serve a replicated application (or a replicated path), and a set of **n_buckets** virtual "buckets" is introduced, to which packets incoming at the load-balancer are hashed, using a conventional hash function on the 5-tuple of (or any other key extracted from) the packet. The goal of the algorithm introduced herein is then to assign one of the **n_servers** servers (or paths) to each of these buckets. This way, packets are assigned (via the hash function) a bucket, itself assigned (via the algorithm presented herein) a server (or path), finally constructing a mapping from packets to servers (or paths). To ease readability of the following paragraphs, the term "server" will be used indistinctly to refer to either servers or paths.

In order to assign a server to each bucket composing the hash space, each server is associated with a pseudo-random (based on server/path ID) permutation of the buckets. This can be seen as a "wish list" of buckets, organized from the highest preference to the lowest preference. Techniques described herein ensure optimal fairness, by assigning **floor(n_buckets/n_servers) + 1** buckets to **n_buckets % n_servers** servers and **floor(n_buckets/n_servers)** buckets to the other servers. During execution, the remaining number of servers that are allowed to have **floor(n_buckets/n_servers) + 1** buckets is therefore maintained (starting at **n_buckets % n_servers** and decrementing). This is achieved by sorting servers and organizing a certain number of successive "turns", at each of which one of the servers will try to pick its next choice in its associated buckets permutation. Once a bucket is picked by a server, it cannot be picked by any other server in any future turn – with the process looping back to the first server once the last server has been reached.

When a given turn is occurring, in order to assign a bucket to the server owning the current turn, that server only inspects a single element (the next bucket) in its own permutation. If the bucket (from the permutation) is available (i.e., the bucket has not been yet assigned to any other server), the bucket is assigned to the server running the current

turn. Otherwise, the server will not pick any bucket at this turn (i.e., it will have to wait for a full selection round before having a chance to pick a bucket again), and will end the turn by letting the next (or, in the case of the last server, the first) server proceed.

Servers are not allowed to pick further buckets once they reach **floor(n_buckets/n_servers)** buckets, or potentially **floor(n_buckets/n_servers) + 1** buckets if and only if the server is amongst the first **n_buckets % n_servers** servers to reach that bar. The reconfiguration process ends when all buckets are assigned to a server or path.

By employing the techniques described herein, consistency is preserved for as long as it doesn't impact fairness, and optimal fairness is ensured near the end of the execution. Techniques described herein also ensure good consistency by limiting the number of buckets that are needlessly moved from one server to another during reconfiguration.