Plastic Hashing for Even, Stable, Fast Load Balancing

Anand Natrajan[†]
Germantown MD, USA
anand@anandnatrajan.com

ABSTRACT

Distributed systems adapt to changing load conditions by adding or removing servers that process requests. In such systems, it is often efficient to persist connections between specific clients and servers, while concurrently balancing the load between servers. Our algorithm, called plastic hashing, achieves high connection stability with an even load distribution. The algorithm is fast and straightforward to implement and lends itself to distributed decision-making.

[†]The author conducted this work independent of any affiliation to any organisation.

https://doi.org/ ID: CACM-20-12-3853

CCS CONCEPTS

• Computer systems organization~Architectures~Distributed architectures~Client-server architectures

KEYWORDS

plastic hashing, load balancing

ACM Reference format:

Anand Natrajan. 2020. Plastic Hashing for Even, Stable, Fast Load Balancing. Submitted to *Communications of ACM*.

1 Introduction

Architects of distributed systems often try to balance the load generated by a large number of clients communicating with a fleet of servers. The mechanisms they use include a hashing algorithm that uses a request identifier to identify the server that will process the request. Once the server is identified, they may prefer to persist the client-server connection in order to make ongoing communications between the two more efficient. The client-server connection may be stateful, with each installment in the ongoing communication using information from prior installments. Typical examples of explicit and implicit information are sessions, connection streams and server-side caches.

A typical load balancing approach applies a hashing algorithm *h(•)* to a request to identify the server that will process the request. This hashing algorithm often uses a *modulo* function to generate a hashed request identifier, or *hashed request*. The resulting remainder can be used to select the server directly by identifying the ordinal number of a server, or indirectly by identifying the ordinal number of a virtual server, which subsequently identifies the actual server. This approach is *fast*, using the modulo function just once in one invocation of the hashing algorithm. It is *even* because uniformly-distributed hashed requests will be distributed uniformly over the fleet of servers. It is *stable* because requests that result in the same hashed request will always be sent to the same server. Speed, evenness and stability are crucial to load balancing algorithms or technologies because they enable the design of robust and efficient systems.

Changes to the operating characteristics of the system, specifically, a change to the volume of inbound requests, may change the number of servers in the fleet. If the server count grows, the new servers should service the excess volume. However, the process of re-hashing all inbound requests may cause some ongoing requests to be handed off to a different server, which may necessitate an expensive set of operations to re-establish the state between the client and the new server. Likewise, if the server count reduces, indubitably some requests will have to move from defunct to surviving servers. The goals of evenness, stability and speed can sometimes conflict with each other in the face of changing server counts.

We propose a novel hashing algorithm, called *plastic hashing* that achieves a desirable balance between evenness, stability and speed. In the next section, we will examine popular alternative approaches so as to highlight the difficulties involved in achieving even, stable and fast load balancing. Subsequently, we will present the plastic hashing algorithm. Next, we will present comparisons of plastic hashing with the alternatives for simulated workloads. These workloads are crafted to subject our simulated fleet of servers to shocks small and large in either direction. Our results show how all of the algorithms adapt to these shocks, in terms of evenness and stability. We will then discuss results and policies briefly before concluding.

2 Related Work

Distributed systems often adapt to changes in the request load by changing the number of servers. Obviously, increasing the number of servers when load increases improves throughput and latency. Decreasing the number of servers when load decreases results in lower costs by increasing utilisation. The naïve approach to balancing changing load simply uses the new count of servers in the modulo function within *h(•)*. Doing so is fast and even, but not stable. The change in the divisor causes a large proportion of requests to move from one server to another.

Consistent hashing is an alternative approach that achieves more stability, but at the price of evenness. Here, the load balancing system maintains a list of slots, arranged as a ring [1]. Each server is hashed using *h(•)*, the same algorithm as the requests, but the enclosed modulo function uses the number of

slots, not the number of servers. A server that hashes to a specific slot is said to occupy that slot. Inbound requests are hashed and subjected to the same modulo function. If the resultant slot for a request coincides with the slot occupied by a server, that request is processed by that server. If the slot is unoccupied, the algorithm finds the nearest occupied slot, usually by walking along the ring clockwise. The first occupied slot identifies the server used to process the request. When a new server is added, it occupies a new slot in the ring. The only requests that move are the ones that hash in between this new server and the server occupying a previous slot. Likewise, removing a server affects only the requests incident on this server which have to move further along the ring to the next occupied slot. Finding the next server in the ring can be slow if the ring is sparse, i.e., the number of slots far exceeds the number of servers. However, speed can be improved by constructing more sophisticated support data structures and algorithms to search the ring. The most serious criticism of consistent hashing is that it can lead to unbalanced loads, i.e., violate evenness greatly. This criticism can be rebuffed somewhat by adjusting the modulo function, the number of slots and the algorithm to make every server occupy multiple slots on the ring.

Rendezvous hashing is another approach that achieves stability by hashing each request with every server in the fleet and picking a winning hash [2]. In other words, for every request, the algorithm invokes *h(•)* for every server. From the resulting set of hashes, the algorithm picks a predictable winner, usually the maximum. The server contributing to that hash is chosen as the one to process the request. Choosing several algorithm parameters wisely is critical. Specifically, adding and removing servers must preserve the roughly uniform probability of any server being selected. With a good choice of hashing algorithm, rendezvous hashing can achieve a high degree of evenness and stability, but at the expense of speed. For large fleets of servers, computing the per-server hash for every request can become expensive.

3 Algorithm Description

Our new approach, called plastic hashing, is an alternative to previous approaches. Plastic hashing uses a simple but novel algorithm to assign a server for each request. The algorithm achieves evenness comparable to the naïve approach, stability comparable to consistent and rendezvous hashing, and speed comparable to naïve and consistent hashing. The algorithm relies on a "configuration history", which is simply a list of the counts of servers in each epoch. For example, if in a particular system, the initial count of servers is 5, then grows to 7, then shrinks to 4, the configuration history is (5, 7, 4) at the end of those three epochs. This list can grow unbounded, although in a later section we will show how to reduce its size. The algorithm does not specify what an epoch is, nor how long. Practically, we expect an epoch to be any length of time during which the server counts do not change. Presently, we will introduce the concept of a "quiet" epoch during which we can perform housekeeping.

We present a pseudocode version of the algorithm below, and work through it with some examples in the table following. The table shows some sample requests with request identifiers in the first row. Successive rows show initial, growing and shrinking epochs respectively. The numbers in each cell show the server number allocated by plastic hashing (servers are numbered from 0 onwards). The subscripted numbers show what naïve hashing would have done, as comparison.

```
1  def h(•) as algorithm to get server for one request
2     let id be the identifier for the request
3     let N: N0, N1, ... Nk be the configuration history
4     let Nold ← N0
5     let Sold ← id modulo Nold be the chosen server
6     for each Nnew ← N1 ... Nk
7        let Snew ← id modulo Nnew
8        if ( Nnew > Nold and Snew ≥ Nold ) or
9           ( Nnew < Nold and Sold ≥ Nnew )
10             let Nold ← Nnew
11             let Sold ← Snew
12    return Sold as the chosen server
```

The algorithm cycles through the configuration history, from oldest to newest. Consecutive epochs grow ($N_{new} > N_{old}$) or shrink ($N_{new} < N_{old}$) server counts. The algorithm begins by computing the request modulo the server count of the first epoch in the configuration history. Tentatively, the algorithm selects the resulting server as the chosen server. The algorithm then goes to the next epoch in the configuration history and computes the request modulo the server count in the new epoch.

Epoch / **id**	**280**	**78**	**111**	**354**	**417**	**361**
N = (5)	$_0$0	$_3$3	$_1$1	$_4$4	$_2$2	$_1$1
N = (5, 7)	$_0$0	$_1$3	$_6$6	$_4$4	$_4$2	$_4$1
N = (5, 7, 4)	$_0$0	$_2$3	$_3$3	$_2$2	$_1$2	$_1$1

Consider the request with id = 78. In the first epoch, both plastic hashing and naïve hashing place the request on server 3 (78 modulo 5). In the next epoch, the server count grows to 7, and N = (5, 7). The naïve algorithm would move the request to server 1 (78 modulo 7). In contrast, the plastic algorithm computes S_{old} = 3, N_{old} = 5, N_{new} = 7 and S_{new} = 1. As a result, the condition on line 8 evaluates to false, leaving S_{old} unchanged as the server choice. In the next epoch, the server count shrinks to 4, and N = (5, 7, 4). The naïve algorithm would again move the request to server 2 (78 modulo 4). But the plastic algorithm computes S_{old} = 3, N_{old} = 5, and on the second iteration, N_{new} = 4 and S_{new} = 2. This time, the condition on line 9 evaluates to false, again leaving S_{old} untouched. This conservative behaviour of the plastic algorithm contributes to its stability.

Of course, the algorithm also must try to even the load. Consider the request with id = 111. In the first epoch, it is placed on server 1. In the next epoch, S_{old} = 1, N_{old} = 5, N_{new} = 7 and S_{new} = 6. Now, the condition on line 8 evaluates to true, funnelling off this request to the newly-added server 6. The next epoch shrinks the fleet, so by the second loop iteration, S_{old} = 6, N_{old} = 7, N_{new} = 4 and S_{new} = 3. The condition on line 9 evaluates to true, representing a forced move from defunct to surviving servers. In this case, the algorithm sacrificed stability either to even out the load or out of compulsion.

This small sample of requests clearly shows plastic hashing attempting to preserve the server allocation for requests across epochs. The algorithm is conservative about changing the server to which a request is allocated, and does so only when forced to in a shrinking phase, or in a proportional manner in a growing phase.

A challenge with plastic hashing is that the time required to identify a server grows as the configuration history grows. For each request, the number of modulo operations required increases as the configuration history accretes epochs. We introduce a housekeeping operation, called "snap", to speed up plastic hashing. The snap operation, typically performed during some quiet epoch, discards the entire configuration history except for the last entry, i.e., the current server configuration. Since the entire configuration history vanishes, the algorithm incurs an epoch's worth of moves. However, after those moves, plastic hash functions as efficiently as the naïve algorithm.

Virtual hosts can be used in conjunction with consistent hashing and rendezvous hashing as well as plastic hashing for an additional layer of load balancing or for redundancy. Here, the algorithms are deployed to select a virtual server, as opposed to an actual server. The virtual server in turn points to one or more actual servers that field the request. Virtual hosts permit adding and removing servers in arbitrary order, not just last-in-first-out.

Plastic hashing is well-suited for making distributed decisions. Some systems eliminate the load balancer itself within the architecture, relying instead on the clients to balance the load. This approach not only alleviates the risk of a single point of failure, but can also reduce the number of network hops a request must traverse before landing on a server. Such client-side load balancing requires all clients to share not just the hashing algorithm *h(•)*, but also the state variables that factor into *h(•)*. The state variable that must be shared in plastic hashing is the configuration history. Specifically, the *change* in the configuration history is the *only* state variable that must be coordinated among clients. The hashing algorithm, the maintenance of a sequence of counts, and the triggers for an automatic snap can all be decided before the system becomes operational. In contrast, the shared-state requirements of consistent hashing and rendezvous hashing are significantly larger.

4 Methodology and Results

In order to view and compare the various algorithms in action, we simulated a synthetic load and ran it for several epochs against each algorithm. In our simulation, we ran 100,000 requests in each epoch. In a real-life workload, a request would likely be identified by some transaction ID. For our purposes, a simple counter from 0 through 99,999 sufficed as the request identifier because it is uniformly distributed. A request with an identifier *R* can be considered the next installment of a request with the same identifier *R* from a previous epoch. Controlling the request identifiers enabled us to control all of the variables that might affect which server processes which request.

The requests themselves did nothing and incurred no processing time. Given our desire to merely compare algorithms, our simulation abstracted away all of the complexity of distributed systems, such as server failures, network connection speeds, faulty responses, etc. While those considerations are important in the design of distributed systems, none of them affects the operation of any of the load balancing algorithms. These algorithms can be imagined as running entirely within a load balancer, and simply computing a server number for each request.

We crafted two workloads, labelled "adjusting" and "chaotic", each with 10 epochs. In each workload, we changed server counts in the first 7 epochs, reserving the last 3 epochs for quiet periods that did not change server counts. In the adjusting workload, the server counts changed by small amounts in the range 45-55 at random. In the chaotic workload, the server counts changed to any number in the range 1-99. For consistent hashing, we chose 1024 slots with 8-way replication for each server.

For each request in each epoch, we noted if the request ended up on the same server as it did on the immediately prior epoch. If it did not, we incremented a counter to count such moves. We computed the percentage of requests that moved, to lend perspective to the magnitude of changes. The lower the percentage, the greater the stability. At the end of each epoch, we tallied up the number of requests that were processed on each server in the fleet. We then measured the coefficient of variance for the distribution of requests (i.e., the ratio of the standard deviation to the mean). The lower the coefficient of variance, the more even the load distribution. Finally, we measured the time each algorithm took to complete each epoch.

In Figure 1, we show the number of moves incurred by each algorithm as a percentage of the total number of requests, for the adjusting workload. The X-axis shows that the server counts change by relatively small numbers as the epochs progress from left to right. In the first epoch, every algorithm assigns requests to servers for the first time, so we do not count them as moves. For every subsequent epoch, we count how many times a request moved to a server different from the server it was on in the previous epoch. The large bars for the naïve algorithm are expected. Every alternative tries to reduce the number of moves. Plastic hashing compares favourably against all of the alternatives except for the solitary spike in the third-to-last epoch. This spike is because we initiate housekeeping in a quiet epoch, the benefits of which can be seen shortly.

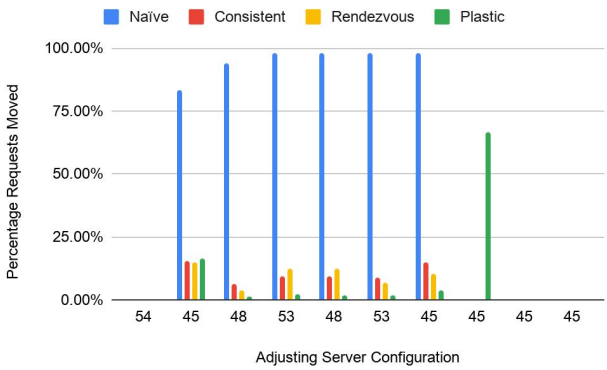

Figure 1: **Number of moves as a percentage of total requests for each epoch in an adjusting workload. Lower percentages indicate desirably greater stability.**

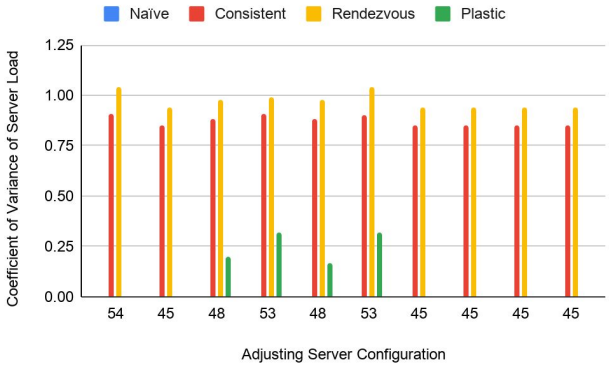

Figure 2: **Coefficient of variance of server load at the end of each epoch in an adjusting workload. Lower coefficients of variance indicate desirably more even load distribution.**

In Figure 2, we show the coefficient of variance of the load distribution across the fleet of servers for the same adjusting workload. The naïve algorithm always has a perfect distribution of load across servers, and therefore shows zero coefficient of variance. Every other alternative encounters some non-zero coefficient of variance. Of these, plastic hashing shows the lowest coefficient of variance, i.e., the most even load distribution.

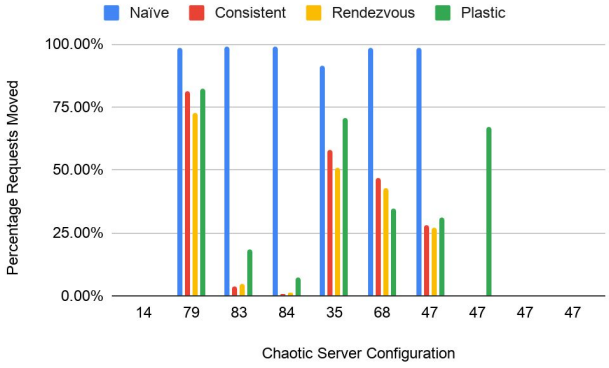

Figure 3: **Number of moves as a percentage of total requests for each epoch in a chaotic workload. Lower percentages indicate desirably greater stability.**

For the chaotic workload, the X-axis in Figure 3 shows that the server counts change by relatively large numbers for consecutive epochs. Every algorithm incurs a greater number of moves to adapt to these changes, although the naïve algorithm continues to be the least stable. Plastic hashing is within reach of all of the alternatives. Once again, plastic hashing incurs a spike in moves because of the housekeeping in a quiet epoch.

The naïve algorithm continues to have a perfect load distribution for the chaotic workload as well, as seen in Figure 4. Plastic hashing shows the next lowest coefficient of variance, i.e., the next most even load distribution. Notice how the coefficient of variance for plastic drops even further, to a perfect zero during the quiet periods. The benefit of the snap is that it compresses the configuration history to a *single* entry, i.e., the current server count. Doing so results in an even load automatically.

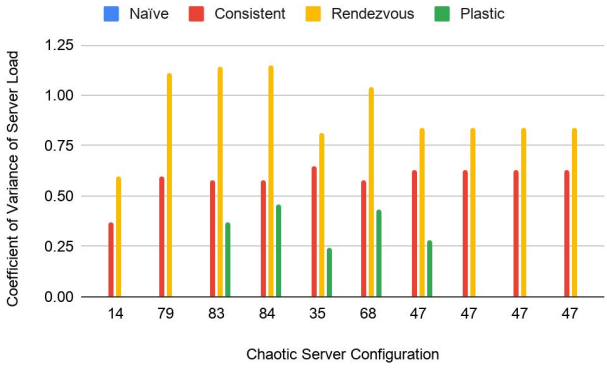

Figure 4: **Coefficient of variance of server load at the end of each epoch in a chaotic workload. Lower coefficients of variance indicate desirably more even load distribution.**

We present empirical observations about the performance of the algorithms rather than numerical comparisons. Clearly, load balancers must select a server for a request rapidly, so as to not add to the request processing latency. The modulo operation inherent to all hashing algorithms typically takes a few nanoseconds on the modern processors within most load balancers. However, repeated invocations of the modulo function per request can add up to a significant latency penalty. In deference to work done by others, we did not attempt to fine-tune our implementations of each algorithm to improve wall-clock execution time. In our empirical observations, the naïve algorithm is expectedly the fastest. Even our simple-minded implementation of the ring traversal for consistent hashing is fast, typically taking only twice as long as naïve. Plastic hashing is next fastest, taking anywhere from twice to eight times as long as naïve. Rendezvous hashing is the slowest, with speeds two orders of magnitude slower than naïve. These observations comport with a straightforward tallying up of the number of modulo operations each algorithm requires per request.

5 Discussion of Results

Expectedly, the naïve algorithm has the worst stability, i.e., greatest number of moves. Changing the number of servers, whether more or fewer, by a little or a lot, causes virtually all requests to move around. This well-known behaviour of the naïve algorithm is the reason alternatives were proposed in the first place. The naïve algorithm does have the property of perfect evenness. The number of requests on each server is identical to the number on any other server (ignoring "off by one" cases when the number of requests is not an integral multiple of the count of servers). Intuitively, the naïve algorithm is fast because the only work involved is computing one modulo function. Our empirical observations corroborate that intuition.

Consistent hashing and rendezvous hashing have the best stability (least number of moves). However, stability comes at the price of uneven load distribution (large coefficient of variance). It

is somewhat difficult to adjudge which of these is better than which other because the results depend a lot on various configuration factors. Consistent hashing relies on a wise choice of the number of slots and number of replicas for a server in the ring of slots. The slot hashing can result in an uneven placement of servers in the slot ring, which makes server load uneven. Adding and removing servers necessitates removing all replicas from the slot ring, which can perturb the load distribution even more. Consistent hashing is a pretty quick algorithm. Although our simple-minded implementation performed a sequential search around the slot ring, it was still quite speedy. Faster implementations using binary search can speed up the algorithm even more, and are likely necessary if the number of slots is large and the slot ring is sparse.

Rendezvous hashing has stability and evenness comparable to consistent hashing. The algorithm relies on a wise choice of hashing function to produce a joint hash of every request with every server. The winning hash for every request (usually the maximum), must itself be evenly distributed across the fleet of servers. Although several excellent hashing functions exist and can be employed, the rendezvous algorithm does run the hazard of encountering server configurations that favour some servers more than others, resulting in an uneven load distribution. The choice of the hash function has to be wise in yet another regard; it must be fast. Because rendezvous hashing requires each request to be hashed with every server in the fleet, if the number of servers becomes large, the algorithm slows down. Some alternative approaches have been proposed in the literature to speed up this process. Our simple-minded implementation performed a brute force calculation of all hashes, which penalised the wall-clock performance of rendezvous hashing.

Plastic hashing exhibits a coefficient of variance significantly better than the alternatives, approaching naïve hashing. It does so with a stability that is almost as good as consistent hashing and rendezvous hashing. Already fast, plastic hashing speeds up further every time the snap operation is performed.

Snap Judgement. The snap clearly adds to the number of moves incurred by plastic hashing. Without a snap, plastic hashing would still have stability that rivals consistent and rendezvous hashing, with superior evenness. However, the performance of the algorithm would steadily degrade as the configuration history grew. The snap offers an opportunity to wipe the slate clean, albeit with some transient pain. Notice that the coefficient of variance in the quiet periods after the snap is zero, unlike the alternative algorithms, which retain their final distribution, however uneven.

The snap operation can be triggered at any convenient time. In our simulations we triggered it at the first epoch where the server configuration did not change. Upon reflection, the snap operation really devolves into two separate and orthogonal decisions: *when* to invoke it, and *what* to do when invoked.

As to *when*, it is always possible to invoke the judgement of a human operator. However, we can strive to encode and automate the decisions such a human would make. Extreme choices for that decision are "never" and "before every epoch". More nuanced choices, such as "before every *k*th epoch", "when server counts are below some *n*", "when request volumes are below some *r*" and "when server counts have not changed in *k* consecutive epochs" are worth considering as well.

As to *what*, the gentlest change to the configuration history is to unify adjacent configurations that have identical server counts. The snap operation we discussed earlier is drastic, decapitating all of the history except for the last entry. Cleaving the configuration history into two equal or unequal sections is another option, although it is unclear which section to retain; the latter section contains the current server configuration, but the former section builds up the flavour of the conservative allocation within plastic hashing. Yet another option is to spring back to an earlier configuration history. For example, if the configuration history is $N = (53, 47, 51, 59, 61)$, and the next epoch is about to add 47 to the list, we can spring the history back to $N = (53, 47)$. We could also anneal the history, gradually changing previous configurations by small amounts so that the final history resembles the current server configuration alone.

Most of the *when*/*what* combinations possible from the choices above outperform alternative hashing algorithms. Evidently, some of the possible combinations are pathological, therefore undesirable. For example, snapping before every epoch causes plastic hashing to degenerate to naïve hashing. Any combination involving annealing causes too many moves over time. Our preliminary studies indicate that the best combinations are those that are sensitive to the current state of the system. In particular, our recommended snap at periods of stasis, when server counts are not changing, achieves a balance between number of moves, load distribution and speed. Likewise, springing back before every epoch to a prior history if possible, also achieves a similar balance. By design, our simulations do not permit modelling periods of low request volumes, but intuitively, a policy built using low request volume as a trigger is likely to work well.

6 Conclusions

We have presented a novel algorithm, called plastic hashing, that can be used for selecting which servers process which requests. The algorithm results in an even, stable and fast distribution of requests to servers. Addition and subtraction of any number of servers causes a small number of requests to move between servers, but the percentage of requests that must move is comparable to the best alternatives. Where plastic hashing excels is in the even distribution of requests, a characteristic improved upon even further with an infrequent snap operation that restores the variance to the ideal value of zero. The snap also improves runtime performance of the algorithm.

ACKNOWLEDGEMENTS

We thank several colleagues who read earlier versions of this paper, and provided valuable feedback.

REFERENCES

[1] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, 654-663, 1997.

[2] D. Thaler and C. Ravishankar. A Name-Based Mapping Scheme for Rendezvous, *University of Michigan Technical Report*, CSE-TR-316-96, 1996.