CS552: Computer Networks
Project Report

Guido Tagliavini Ponce

June 9, 2018

Abstract

We consider the paper "Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services" [3]. In that paper it is shown, theoretically and empirically, that for a distributed key-value store randomly partitioned over n back-end nodes, a front-end cache with $O(n \log n)$ items guarantees that no node will ever be overloaded. We take the most significant experiments and build a simulator to mimic them. Our results match the paper's to a great extent.

Contents

1 Introduction **1**

2 Load balancing by means of caching **2**

3 The $O(n \log n)$ bound **3**
3.1 Model and assumptions 3
3.2 Optimal adversary 4
3.3 Key partitioning as a balls-into-bins game 5

4 The simulation **6**
4.1 System specification 7
4.2 Simulation design 7
4.3 Calibration 8

5 Results and discussion **11**
5.1 Further experimentation 12

6 Appendix: about the software **14**

1 Introduction

In the paper "Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services" [3] (from now on, *the paper*), Fan et al. prove that given a key-value store (KVS) randomly partitioned across n back-end servers, an $O(n \log n)$-size front-end cache is enough to guarantee that no adversary can overload a server. In other words, caching the $O(n \log n)$ most popular items is enough to guarantee load balance. This bound implies that only a small cache is needed, regardless of the number of items stored in the KVS or the access pattern.

Web services usually rely on front-end in-memory caches to reduce back-end servers' workload and hopefully balance the aggregate load [4]. Service designers can leverage the provable effectiveness of a

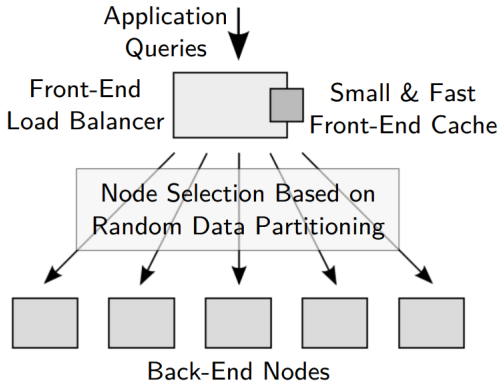


Figure 1: A distributed KVS with a front-end cache.

small cache to work out service architectures that will meet, with high confidence, latency service-level agreements (SLAs). Moreover, it allows distributed KVS designers to bound the memory usage for caching purposes when resources are limited, for example when the cache is located in a network switch [6][5].

Besides proving the theoretical bound, the authors validated their results by setting up a distributed KVS over $n = 85$ back-end nodes with a front-end cache stored on SRAM, and exposing the system to an adversarial workload. This type of workloads depend on the number of different keys that the adversary queries. We call this number the *adversarial range*. The larger the adversarial range, the more spreaded the load. The authors perform several experiments, of which we are interested in three of them. They are presented in the following figures from the paper:

- Figure 9, that shows the system-wide throughput as a function of the adversarial range, when there is no cache. The goal is to show that the system's performance with no cache can be terrible.
- Figure 11, that shows the system-wide throughput as a function of the cache size. For each cache size, several adversarial ranges are tested and the worst system performance is plotted. The goal is to find out what is the minimum cache size we need to guarantee full aggregate system throughput.
- Figure 10, that shows the system-wide throughput and back-end throughput as a function of the adversarial range, when there is an $\Theta(n \log n)$ cache. The goal is to show that an $\Theta(n \log n)$ cache effectively balances the load.

The goal of this project is to reproduce these measurements. The rest of this report is organized as follows. Is Section 2 we explain why caching is a good load balancing technique. In Section 3 we derive the $O(n \log n)$ bound proof. We describe the abstract model the authors use, which is useful to precisely understand what assumptions are made, and we discuss how these assumptions may limit the applicability of the bound. In Section 4 we describe the simulator we built to reproduce the experiments, and the challenges we faced during its development. Finally, in Section 5 we present our results and compare them against the paper's.

2 Load balancing by means of caching

Load balancing is an essential component of reliable web services. Query load is naturally imbalanced, because some of the items are more popular than others, and thus requested more frequently. It's

also dynamic, since hot items change from time to time. A simple distributed KVS that uses a static assignment of items to machines, and does no effort to balance the load, will utterly fail to provide a good service because as soon as an item gets hot the machine that owns that item will get overloaded.

There are several load balancing techniques. Many of them are based on moving data around or replicating partitions. For example, Dean and Barroso [2] describe a set of techniques called *micro-partitioning* and *selective replication*, which consist of splitting the universe of key-values into small partitions, such that there are many more partitions than machines. Then, as load imbalance shows up, small hot partitions are moved from one machine to another to rebalance workload among all machines. On the downside, these kind of techniques are hard to engineer, and they incur on some network and space overhead.

Front-end caching is, in contrast, a much simpler load balancing technique. The basic idea is that items cached in a front-end node are retrieved much faster than queries that need to go to the back-end, thus alleviating back-end load. When an item is brought from the back-end, it is cached so that subsequent queries for that item are served instantly by the front-end. A typical web service can store several TB of data over the back-end nodes, so only a small fraction of the whole store can be cached in the front-end node.

Why is caching effective at load balancing? The key idea is that if there are only a few hot items, then the cache will answer all those requests, and as queries get spreaded over many items the load on individual back-end nodes gets diluted, so every server can keep up with the demand. Thus, the cache absorbs most of the work in the unbalanced scenario that hurts the back-end.

How big does the cache needs to be? We can assume the maximum query rate is bounded by the aggregate throughput of all back-end nodes, since beyond that threshold some node will necessarily get overloaded independently of how well the load is balanced. Since an adversary can put only a bounded load on the system, then for a sufficiently big adversarial range, the query rate per server will be small. The cache needs to be big enough so that if the adversary is able to bypass the cache, then by the time queries hit the back-end the maximum load on a single server is not too big.

3 The $O(n \log n)$ bound

3.1 Model and assumptions

We start defining some notation:

$n = \#$ of back-end nodes

$m = \#$ of different keys stored

$c = \#$ of items the cache can store

$q =$ client query rate

$t =$ maximum throughput of any back-end node

Query rate and throughput are measured in QPS. The authors assume a system with the following properties:

1. **Randomized key partitioning.** Each key is assigned to a random server, and this partitioning is unknown to clients.

2. **Front-end is arbitrarily fast.** The front-end node is fast enough to handle an arbitrary number of queries per unit time, be it retrieving a cached item or forwarding a query to the back-end. Thus, the front-end is never a bottleneck.

3. **Network is arbitrarily fast.** Network bandwidth and delay are never a bottleneck.

4. **Perfect caching.** The cache knows what the c most queried items will be, even before the sequence of requests starts.

5. **Uniform cost.** The cost of processing a query is independent of the key and the back-end node serving the query.

The authors do not explicitly state the fast network assumption, but they mention it when they describe the experimental setting.

It's natural to ask is if these assumptions are reasonable. Randomized partitioning is realistic, since spreading the items uniformly is the most simple way to avoid imbalance, and it's what most distributed stores do. The front-end being arbitrarily fast is also reasonable, since a fast memory cache is able to perform hundreds of millions of lookups per second, enough to keep up with reasonably large demand rates. If the front-end turned into a bottleneck, we could add front-end nodes, and replicate the whole cache. This is feasible since, as we already advertised, only a small cache is necessary. The same reasoning holds for the arbitrarily fast network assumption. The uniform cost assumption is reasonable as well, because KVSs typically have similar access times for most of the keys.

In contrast, the perfect caching assumption is obviously unrealistic, since there is no such thing as an oracle that can tell us in advance which will the future requests be. The goal of this assumption is to simplify things enough to allow a theoretical analysis. However, the small cache argument extends, at least from a practical point of view, to real-life caches. To support this claim, we perform some experiments with an LRU cache.

We can outline the proof as follows:

1. Bound the maximum query rate per key.

2. Bound the expected number of maximum uncached queried keys on any server.

3. The product of these two values yields a bound on the expected maximum load per machine. Since this bound is a function of c, we can choose c such that the bound is below the capacity t of every machine.

3.2 Optimal adversary

The first step towards the bound is to devise an adversary that maximizes the chances of overloading some server. Let $q_{\max}$ be the maximum query rate going to some server. Our goal is to characterize an adversary that maximizes $E[q_{\max}]$. The bigger the $E[q_{\max}]$, the higher are the chances of overloading a server. In particular, if $E[q_{\max}] > t$, the adversary is expected to overload some server.

We assume that the adversary knows:

- Which keys are stored in the system.

- The number of back-end servers n.

- The cache size c.

However, we assume it doesn't know how the keys are hashed into the servers, and that it cannot learn this partitioning (i.e. it's not adaptive).

Number the keys from 1 to m, and let p_i be the probability that the adversary queries key i. That is, the adversary queries key i, p_i of the time. Note that $p_1 + \cdots + p_m = 1$. Assume, w.l.o.g., that $p_1 \geq p_2 \geq \cdots \geq p_m$ (otherwise, reassign key numbers). Since the cache is perfect, it will contain keys $1, \ldots, c$, at all times. Thus, for the adversary to maximize the expected $q_{\max}$, it should always prefer to query keys $c+1, \ldots, m$, and within this subset it should try to query as few keys as possible, as so to consolidate most of the load in as few servers as possible.

The key observation is that for any given key $1 \leq i < c$, the adversary doesn't benefit from querying i more than $i+1$. If so, we could subtract that difference from p_i (the ranking $p_1 \geq \cdots \geq p_m$ of keys by

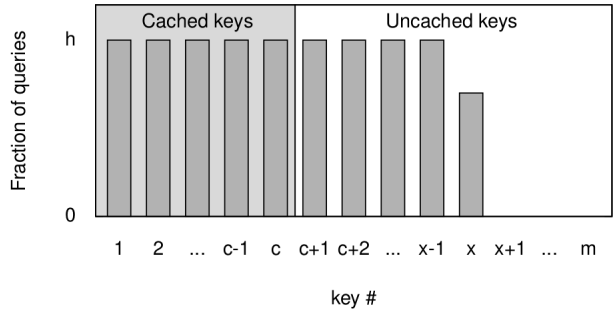


Figure 2: Optimal adversarial strategy.

query frequency remains unchanged), and use that extra time to query uncached keys, increasing $q_{\max}$. This implies that $p_1 = \cdots = p_c = h$ for some $h \in (0, 1)$. Thus, $p_i \leq h$ for every $i > c$. Since the optimal strategy is to put all the load in as few keys as possible, the adversary assigns probability $p_i = h$ for $i \geq c + 1$, while possible. Specifically, the adversary assigns probabilities $p_{c+1} = \cdots = p_{x-1} = h$ and $p_x = 1-(x-1)h$, for some $x > c$. The value x is the number of different keys queried by the adversary. The number x is what we called the adversarial range. Figure 2 shows a picture of the strategy, which we took from the paper.

The important thing about this strategy is that it distributes the query frequency almost evenly over the adversarial range. More precisely, $h < 1/(x-1)$. We conclude that the maximum query rate per key is $hq < q/(x-1)$.

3.3 Key partitioning as a balls-into-bins game

We can think the process of hashing keys into servers as randomly placing balls into bins. The maximum number of keys assigned to the same machine equals to the maximum number of balls in the same bin.

Balls-into-bins games are a well studied topic in probability theory, and it has many applications in computer science. In 1998, Raab and Steger [7] proved the following theorem, which provides an estimation on the maximum number of balls in a single bin.

Theorem 1 (Restated Theorem 1 from [7])**.** *Let B be the random variable that counts the maximum number of balls in any bin, if we throw M balls independently and uniformly at random into N bins, with $N \log N \ll M \leq N$ polylog$N$. Then,*

$$\Pr\left[B < \frac{M}{N} + \alpha\sqrt{2\frac{M}{N}\log N}\right] = 1 - o(1)$$

for any $\alpha > 1$.

The hypothesis $N \log N \ll M \leq N$ polylog$N$ means that there should be a log factor more balls than bins, but at most a polylog factor. In the small cache paper, the authors use this bound in a context where the number of balls is certainly not bounded by N polylog$N$, because there are many more keys (usually millions) than a polylog factor of servers (usually tens or hundreds). The original theorem in [7] also states an estimation for the case $M \gg N \log^3 N$, which would be more adequate for our use case, but the authors didn't use it (presumably because it's more complex and thus harder to manipulate). However, they present evidence showing that the former estimation is still useful for this case (see Figure 4 in the paper).

In our case, we have $N = n$ bins, and $M = x - c$ balls, because we are only interested in counting uncached keys. Thus, by the previous theorem, the maximum number of queried keys in any server is bounded by

$$\frac{x-c}{n} + \alpha\sqrt{2\frac{x-c}{n}\log n}$$

for any $\alpha > 1$, w.h.p. Multiplying by the maximum query rate per key, we get

$$\begin{aligned} E[q_{\max}] &\leq \left(\frac{x-c}{n} + \alpha\sqrt{2\frac{x-c}{n}\log n}\right)\frac{q}{x-1} \\ &= \left(\frac{x-c}{x-1} + \alpha\sqrt{2\frac{x-c}{(x-1)^2}n\log n}\right)\frac{q}{n} \end{aligned}$$

If we maximize this expression as a function of x, we get

$$E[q_{\max}] \leq \frac{1}{2}\left(1 + \sqrt{1 + 2\alpha^2\frac{n\log n}{c-1}}\right)\frac{q}{n} \tag{1}$$

Thus, if we take $c = kn\log n + 1$, for some positive constant k, we get

$$E[q_{\max}] \leq \left(\frac{1}{2} + \alpha\frac{\sqrt{2}}{2}\right)\frac{q}{n}$$

This means that the expected *maximum* load per machine is at most a constant factor of the *average* load per machine, implying that the overall load q will be spreaded evenly among all n servers. Since the aggregate back-end throughput must be q or more, each machine should guarantee throughput $t \geq q/n$. Thus no machine will get more (than a small constant factor of the maximum) load than it can handle, in expectation. This concludes the proof of the bound.

Suppose $E[q_{\max}] = t$, which means that some machine is at maximum utilization but the system is not overloaded. Then, from Equation 1 we can derive the following lower bound on the maximum query rate tolerable by the system,

$$q \geq \frac{2nt}{\left(1 + \sqrt{1 + 2\alpha^2\frac{n\log n}{c-1}}\right)} \tag{2}$$

4 The simulation

To reproduce the chosen set of experiments, we had to decide between simulating or emulating the system. Initially, we considered running an emulation of the distributed KVS system, using some network emulator software like Mininet, so as to replicate the system behavior as precisely as possible. It was soon clear that this was infeasible if we planned to run several back-end nodes, because they would all intensively contend with each other for the same machine resources. Too much contention could influence the measurements making them less reliable.

On the other hand, a simulation requires substantially less amount of resources at the cost of abstracting some features. Specifically, a simulation wouldn't execute real lookups on a store, and wouldn't send or receive queries through the network. Fortunately, these are not essential features, since cache effectiveness depends on the number of back-end nodes and how many QPS they can answer, but it doesn't matter what the lookup process is or what the answer is. Also, network communication between the client and the front-end, and between the front-end the servers, can be safely abstracted in since the distributed KVS built by the authors the line card bandwidth is high enough in every node so that network cost is a negligible part of the query latency. All in all, we decided that a simulation could accurately mimic the KVS.

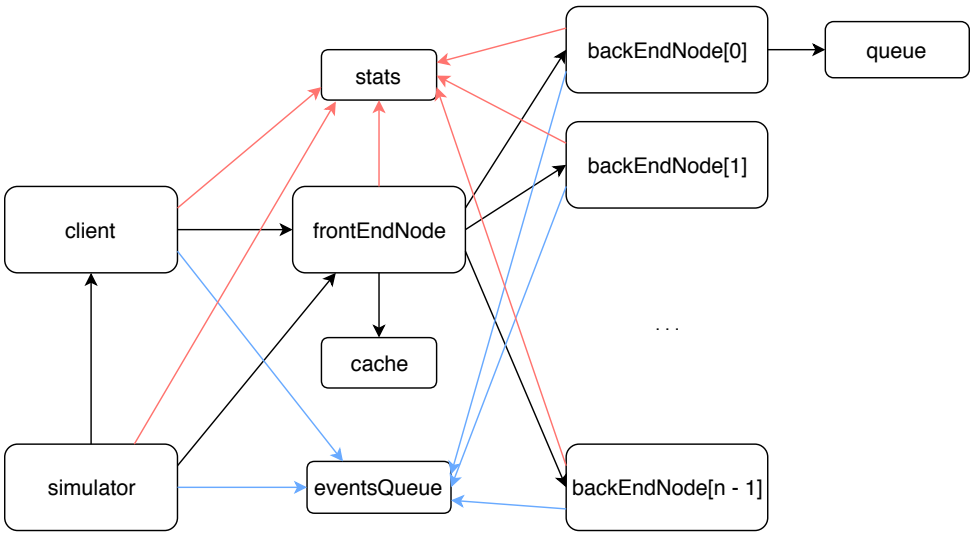


Figure 3: Simplified object diagram of the simulator. Colors are just for clarity. Despite a single `queue` object is shown, every back-end node has one.

4.1 System specification

The authors deployed the distributed store FAWN-KV [1] with $n = 85$ back-end nodes. The front-end node uses a 10GbE link and the back-end nodes use a 1GbE link. The cluster stores $m = 8.5$M key-value pairs, each node responsible for 100k of them. Back-end nodes serve queries at a rate of $t = 10000$QPS.

The client pipelines queries (i.e., sends one query after the other, without waiting for the answer) to hide network latency, but in order to keep the per-query latency under control it limits the maximum outstanding queries per back-end node to 1000. When this value is reached, the client waits for the node's queue to empty, and then resends the query. The paper does not indicate the rate q at which the client issues queries, but we can infer that for some experiments they used $q = nt$ (the aggregate throughput of all nodes), and for some others $q = 2nt$ (to stress the cache).

4.2 Simulation design

Overview The basic structure of our simulation is shown in Figure 3. The main entities in the system are present as concrete objects of the simulation, and they interact in the expected way. These are the `client`, the `frontEndNode`, the `cache`, the `backEndNodes` and the `queues`. The `queue` attached to each `backEndNode` contains all the pending queries for that node. When it reaches 1000 queries, it rejects new requests. The rest of the objects are:

- `simulator`: controls the simulation flow.
- `eventsQueue`: contains the events that will be executed next.
- `stats`: maintains statistics (like the number of queries issued so far) and how much time we have simulated.

The basic idea is that events (e.g., the client issues a query) are triggered by the `simulator` the moment they are taken out from the `eventsQueue`. The events will be popped out in the order they should occur. To materialize this time ordering, we discretize the continuous into slots, each corresponding to some fixed amount of time, and each event is scheduled to happen at a specific time slot. There are three types of events:

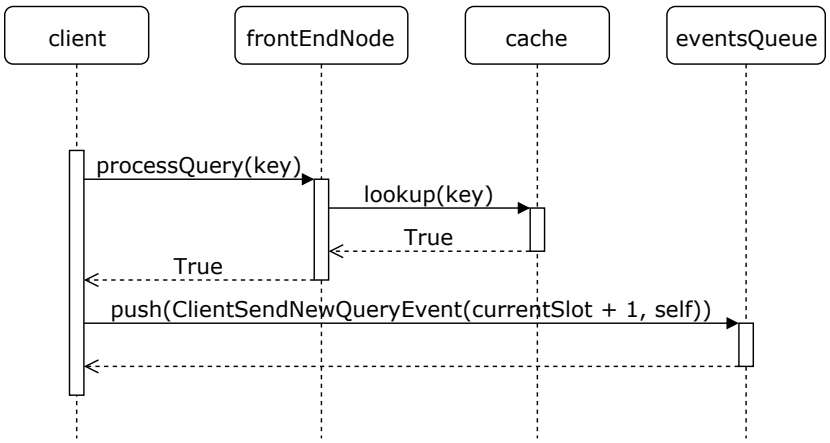


Figure 4: Client sends a query and the item is cached.

- `ClientSendNewQueryEvent`: when the client sends a new query.
- `ClientResendQueryEvent`: when a query that was rejected because a of a full queue, is now retried.
- `BackEndAnswerQueryEvent`: when a back-end server completes a key lookup.

Time discretization Let Δ be the amount of time that spans a single time slot. Since the client issues q QPS and any back-end node answers t QPS, we want $1/q$ and $1/t$ to be an integral multiple of Δ. Since the adversarial query rate q is usually set to a multiple of the server's throughput (e.g. $q = nt$ or $q = 2nt$), we can assume $q = st$ for some positive integer s. If we divide each second into q slots, then $\Delta = 1/q$, so a single slot is the time it takes to issue a query. Moreover, s slots represent $s\Delta = s/q = 1/t$ seconds, and thus s slots are the time that takes a server to answer a query. Thus, this simple discretization works. In what follows, we will write `slotsPerLookup` $:= s$.

Event execution The simulation can be described as a series of two-steps rounds. On each round, the `simulator` fetches the next event from the `eventsQueue`, and then executes it. This loop repeats until we have simulated as much time as desired.

When a `ClientSendNewQueryEvent` or `ClientResendQueryEvent` is executed, the `client` is called to send a query for a given key, and there are essentially three outcomes. In the event of a cache hit (Figure 4), the control goes back immediatly to the client, which schedules a `ClientSendNewQueryEvent` for the next time slot. If the item is not cached, there are two possibilities. If the target server's queue is not full (Figure 5), the query is enqueued and the control goes back to the client, which schedules the next `ClientSendNewQueryEvent`. Otherwise, if the queue is full (Figure 6), the client is informed the query has failed, and it schedules a `ClientResendQueryEvent` far enough in the future (after 1000 back-end lookups) to retry when the queue is empty.

When a `BackEndAnswerQueryEvent` is executed (Figure 7), the corresponding `backEndNode` informs the `frontEndNode`, which in turn notices the cache. Upon learning about the new resolved query, and depending on the type of cache, the cache can decide to write-back the new item. In particular, perfect caches do not write-back, because they already have the hottest items, but our simplified implementation of an LRU cache does so.

4.3 Calibration

Since our goal is reproducing the paper's experiments, we had to make sure our simulation setting is representative of the real-life system the authors built. In particular, our simulator could be abstracting

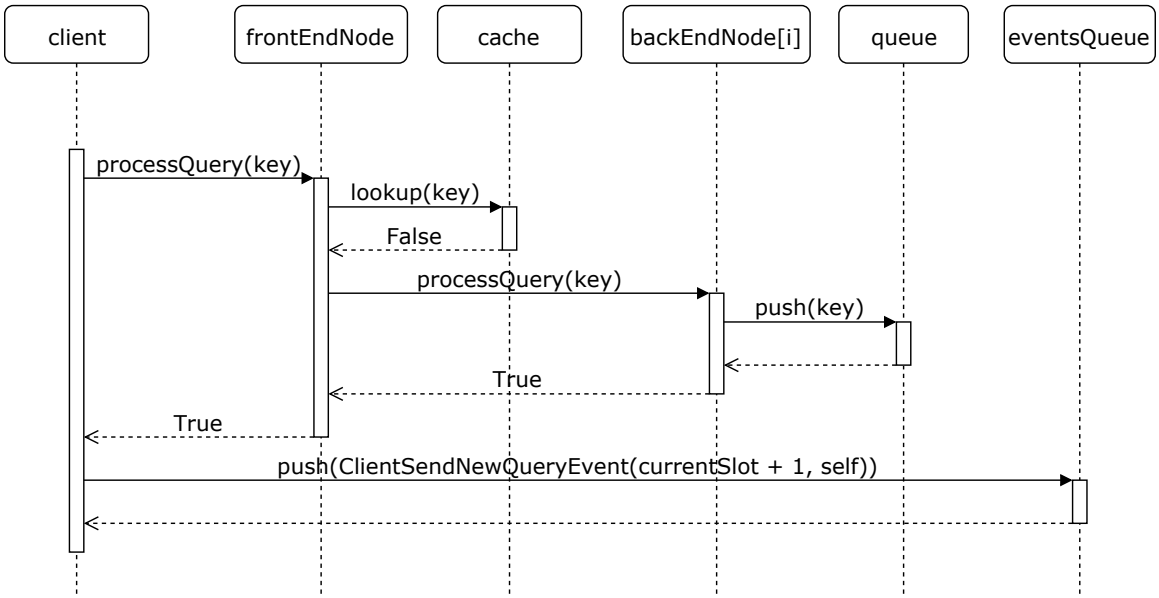


Figure 5: Client sends a query and the item is not cached, so it's enqueued in the back-end.

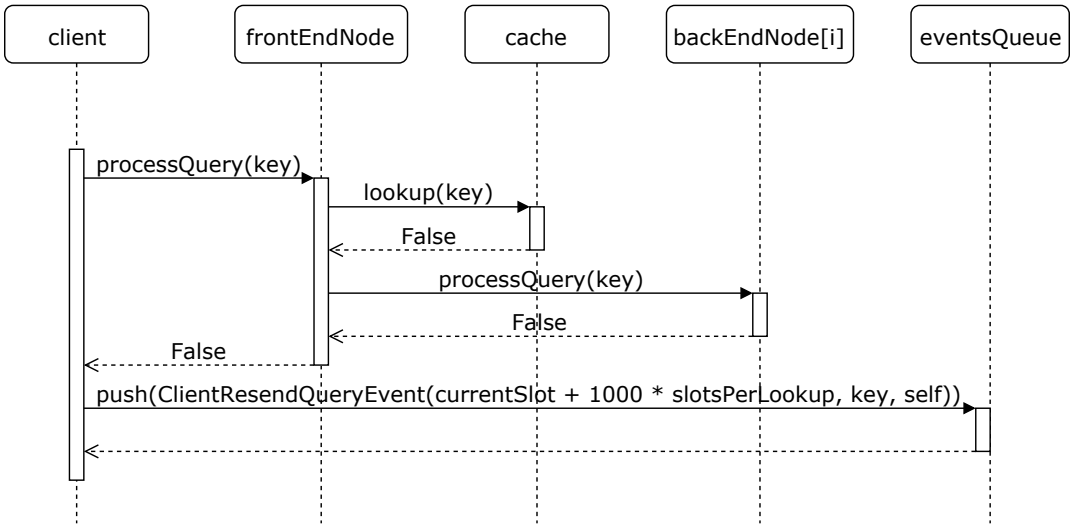


Figure 6: Client sends a query and the item is not cached, but the back-end queue is full.

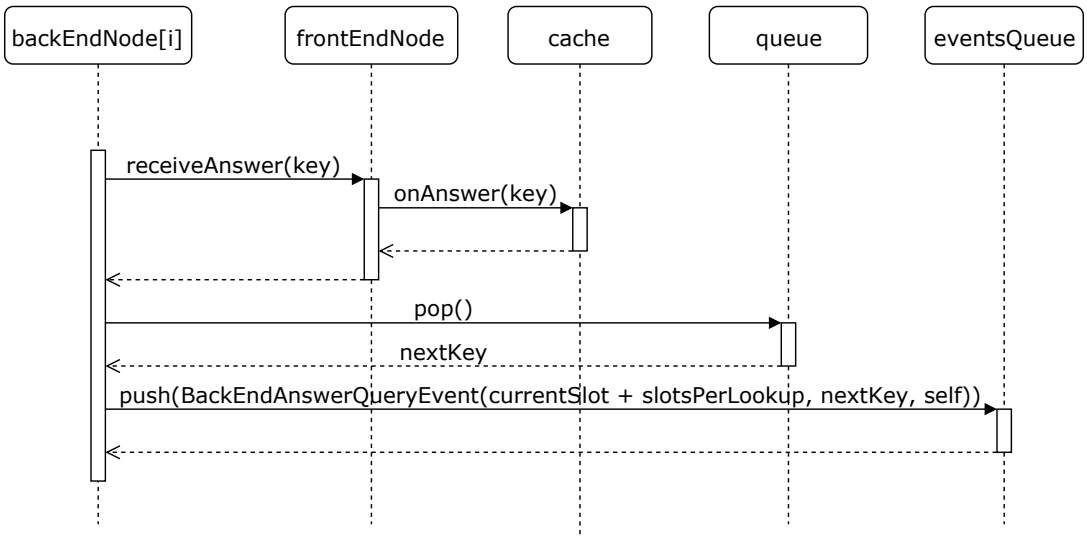


Figure 7: Back-end node answers a query.

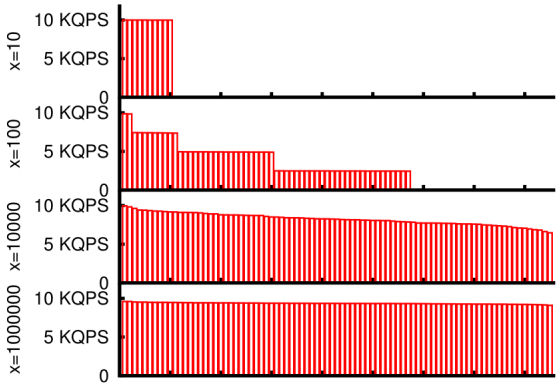


Figure 8: Throughput of each back-end node for different values of x (the adversarial range) when $c = 0$. Nodes are sorted according to their throughput.

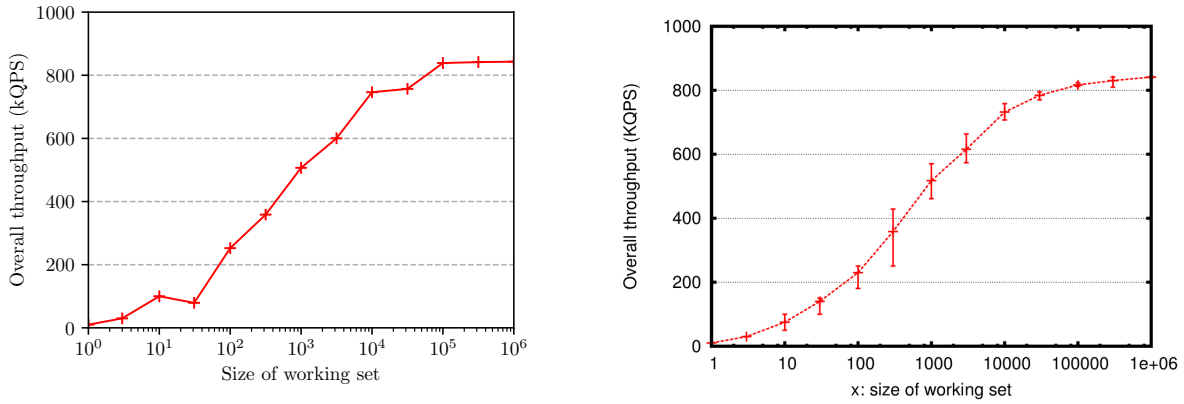
some part of the real system which plays a non-negligible role.

We started by taking the measurements of Figure 8 in the paper (Figure 8 in our report), which shows the individual throughput of each back-end node. One would expect that given the uniformity of the adversarial strategy, queries get evenly distributed on the back-end and so should the per-machine throughput. However, since the client temporarily stops emitting requests as soon as a back-end queue gets full, a skew is introduced. When a queue gets full, the client waits until the queue gets empty to send the next query to that machine. Utilization of the congested server will be 100%, because it is processing queries at all times. In contrast, the remaining machines, with underfull queues, will be sitting idle waiting for the client to resume, and hence their throughput will drop.

Initially, our measurements didn't match with the paper's, because our first version didn't implement any kind of queuing or query rate throttling (we incorrectly underestimated this aspect). The client would continously send queries, regardless of how full a server was, which is unrealistic since servers do not have arbitrarily large buffers. After refining our simulator, we finally matched the numbers.

5 Results and discussion

We attempted to replicate three experiments and obtained good results. The first experiment consists of measuring the system's throughput as a function of the adversarial range (or *working set*, as called by the authors), when there is no cache. Without a cache, the worst-case scenario is an adversary that queries only one key, putting all the load on a single machine. As the adversarial range grows, the load gets distributed across all servers, so throughput increases. Our results and the paper's are shown in Figure 9. As we can see, despite our curve is not as smooth as the paper's, the growth is similar. Discrepancies with our results are within the error ranges measured by the authors (vertical bars on each marked point), and can be attributed to features abstracted by our model. In both plots, the maximum throughput is achieved only for adversarial ranges greater than 100000 keys, thus showing that without a cache an adversary could severely hurt performance.



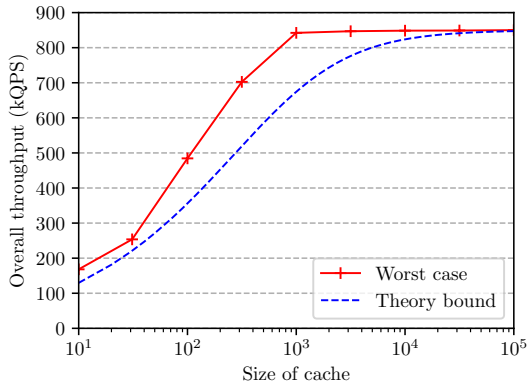
(a) Our results

(b) Paper's results

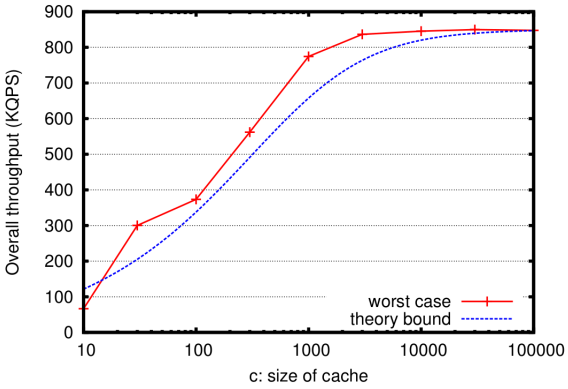
Figure 9: Throughput versus adversarial range, without cache (Figure 9 in the paper).

The second experiment aims to learn what is the minimum cache size we need to attain maximum throughput. Specifically, we want to plot the throughput versus the cache size. The sweet spot should be about $c = kn \log n$, for a reasonable constant k. Figure 10 shows the results. The red curve indicates the system's throughput, whereas the blue curve is the theoretical lower bound on the system-wide achievable throughput, given by Equation 2. Since the blue curve is always below the red one, the plot corroborates Equation 2. At first sight, we can conclude that the simulation's results roughly match the paper's. A more careful examination shows that our simulation achieves maximum throughput for a cache size 1000, whereas their measurements indicate 3000. In any case, the order of magnitude is similar, and validates the theoretical bound. Note, however, that since the authors found the best choice to be 3000 items, this is the cache size they used for subsequent experiments. Thus, we used $c = 3000$ as well.

Indeed, the third and final experiment tests the system's performance for $c = 3000$, versus several adversarial ranges. The comparison is presented in Figure 11. Again, we get similar results. The red curve shows the total throughput, which is the sum of cache and back-end throughputs. The blue line is the back-end's throughput. Since the total throughput is always greater than or close to the aggregate throughput nt = 850kQPS, we can conclude that such a cache allows the system to operate close to maximum utilization even under adversarial workloads. Note that at times the total throughput is over 850kQPS. The reason is that in this experiment the query rate was set at $2nt$ (twice the back-end's capacity) to put a stressing load on the cache. Cache utilization starts at a peak (the difference between the red and blue curves is maximum), and gradually drops until both curves converge at 800kQPS (the reason for being slightly below the aggregate throughput is, we think, related to the queuing overhead aggravated by doubling the query rate). As the adversarial



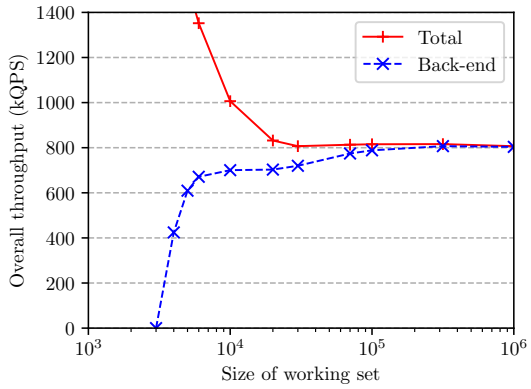
(a) Our results



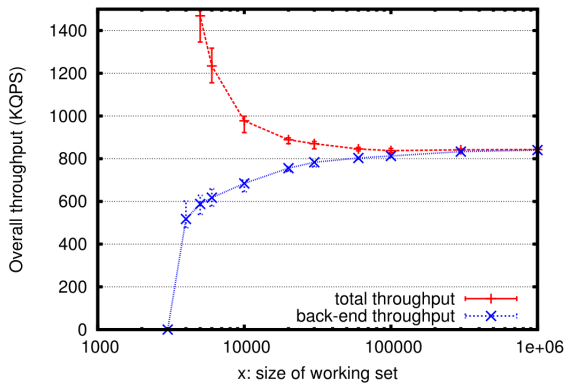
(b) Paper's results

Figure 10: Throughput versus cache size (Figure 11 in the paper).

range grows, cache utilization gets smaller, and the back-end starts taking care of most of the queries. Eventually, when both curves converge, the back-end is fully responsible for all the load.



(a) Our results



(b) Paper's results

Figure 11: Throughput versus adversarial range, with cache size 3000 (Figure 10 in the paper).

5.1 Further experimentation

Given our relative success in replicating the authors' system and experiments, a logical follow-up is attempting to use the simulator to run further experiments and extend the paper's conclusions. This is particularly attractive, given that a simulator allows us to test arbitrarily large systems, but without the real costs of setting them up.

In particular, we wanted to test the system under a LRU caching policy, instead of a perfect cache. In this case, for a cache size c, an adversary will cyclically query keys $1, 2, \ldots, c, c+1$, forcing a miss for every lookup. Note that in our distributed setting cache misses do not instantly trigger an eviction followed by an insertion, because items are written after they are retrieved by the back-end. In the meantime, the client keeps issuing new queries. Thus, there might be some cache hits due to delayed evictions. However, such a cyclic pattern eventually causes cached items to be evicted at every time slot, contributing to the adversary's goal.

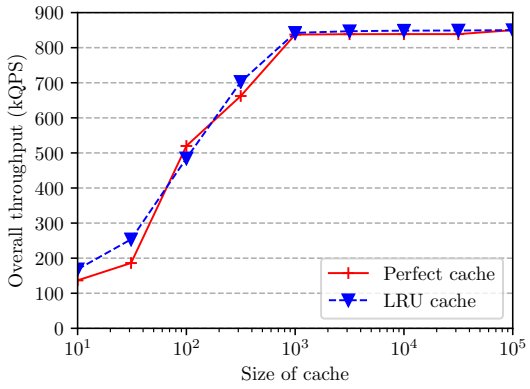


Figure 12: Throughput versus cache size, for two types of caches.

We repeated the experiment of Figure 10, to see what is the minimum cache size required to achieve maximum throughput under adversarial loads, compared with the perfect cache case. The results are shown in Figure 12. The throughput in both cases is almost identical for every cache size, supporting the idea that the arguments hold for other types of caches as well.

References

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[2] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[3] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[4] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[5] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[6] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 31–44, Berkeley, CA, USA, 2016. USENIX Association.

[7] Martin Raab and Angelika Steger. "balls into bins" - a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London, UK, UK, 1998. Springer-Verlag.

6 Appendix: about the software

To run the simulation, simply execute the `run.sh` script. This will run all experiments discussed, writing results on standard output and additionally creating plots which are stored in the (automatically created) `output` folder. To change the desired set of experiments to perform, go to the main method in `simulator.py`. Each experiment has an associated flag variable that can be turned on or off.

The simulator was developed in Python, and is composed of the following files:

- `cache.py`: implementations of a perfect cache, an LRU cache and adversaries.
- `client.py`: the client that sends queries with a particular adversarial pattern.
- `events.py`: the events queue and the different types of events that can happen.
- `nodes.py`: front-end and back-end nodes.
- `plot.py`: a helper class to plot experiment results.
- `simulator.py`: the simulator and several hardcoded experiments.
- `stats.py`: simulation metrics and statistics.