Proxy Servers using Bloom Filters

- Soumya Bharadwaj
- Bipul Bishal Singh

Introduction

Searching in Hash tables is fast and reliable, no doubt about it! However, in scenarios where there are millions or billions of records, the increasing size of the hash table can become a bottleneck for storage. Interestingly, if we compromise a bit on reliability, it leads us to an intriguing and powerful probabilistic data structure called Bloom Filters. In this project, we harness the power of Bloom Filters to implement a set of interacting proxy servers that can communicate with each other for a possible cache hit rather than directly calling the internet.

In essence, this project is an implementation of Summary Cache [1], where each proxy maintains a summary of the URLs of cached documents for every participating proxy. It checks these summaries for potential hits before sending any queries to the internet. Sharing caches among web proxies is an important technique to reduce web traffic and alleviate network bottlenecks, which is successfully achieved in this project.

The project is coded using Java and Spring framework, with network calls made using `` `*java.net*` `` library provided by Java out of the box. Proxy servers share their Bloom Filters in the network by multicasting the filters to all the subscribers, although this is purely a design choice; filters can also be broadcast over the network. To evaluate the performance of our Bloom Filter, we assess it against the theoretical probability of *false positives*.

Motivation

To start with, we were fascinated by the concept of Bloom filters and decided to gain hands-on experience in implementing them. Following our professor's recommendation, this project is inspired by the first example in the paper 'Sample network Application having distributed caching in the paper Network Applications using Bloom Filters: A Survey' [2].

Let us consider a scenario where this project could be particularly useful. Imagine a remote location connected to slow-bandwidth internet. To expedite network calls, web proxies are installed near densely populated regions. While these proxies effectively reduce calls to the internet, minimizing network overhead and providing a better user experience, the question arises: can we do even better? The answer is yes; improvement is possible if web proxies communicate with each other. For example, if one proxy server has a cache miss, it can ask another proxy server in the same region for the webpage if it's already cached. This further reduces network calls and enhances the user experience.

To implement the solution mentioned above, we could directly share the complete cache of one proxy server with others i.e. the Hash Table. However, as mentioned `Introduction1, using the Hash Table can sometimes have storage bottlenecks. Now this is a real-life example where using a Hash table for periodic sharing of URLs causes overhead on the network. Since every proxy server needs to share its cache information with every other proxy server, it requires $n(n-1)$ where n , is the number of servers in the network. Additionally, cache exchange happens frequently because we want the filters to be always updated. In this scenario sending the complete Hash Table, which might consist of millions of caches, from every proxy server in this scenario would be extremely costly for the network. For a solution, we introduce something called the *Summary cache* [1] that uses Bloom Filters: a fixed size bit-array. Let's understand Bloom Filters in depth.

Background Work: Bloom Filter

[2] A Bloom filter is a space-efficient probabilistic data structure conceived by Burton Howard Bloom in 1970, used to test whether an element is a set member. False positive matches are possible, but false positives are not – in other words, a query returns either "possibly in the set" or "definitely not in the set". Elements can be added to the set, but not removed (though this can be addressed with the counting bloom filter variant); the more items added, the larger the probability of false positives.

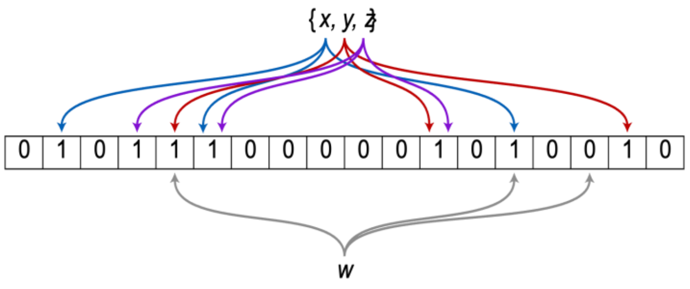


Figure 1: Bloom filter. (2023, December 5). In Wikipedia. https://en.wikipedia.org/wiki/Bloom_filter

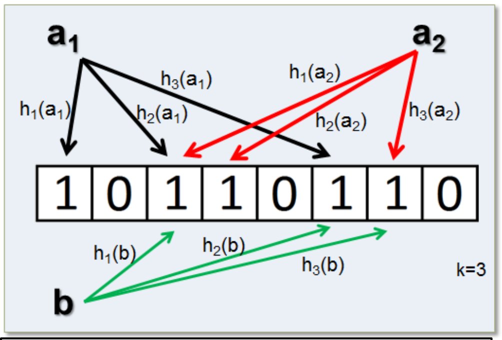


Figure 2: https://www.allthingsdistributed.com/2017/02/bloom-filters.html

A Bloom Filter is a bit-array, as depicted in the figures above. To add an element in the filter, that element passes through a certain number of distinct hash functions. Based on the resolved index from those hash functions, corresponding indexes in the bit-array (Bloom Filter) are set to 1. Similarly, to check if an element is present, that element undergoes the same hash functions, and the resolved indexes are checked in Bloom Filter to see if it contains 1 or 0. If all of them contain 1, then the element is possibly present; else if any of them contains 0, then definitely not present.

For example, in Figure 1: *x*, *y* and *z* elements are pas through 3 hash function, updating the bit-array (Bloom Filter). When we check if *w* is present or not, the third hash function points to an index marked 0. This means *w* is definitely not present in the pool of stored elements.

Conversely, in Figure 2: *a1* and *a2* are passed through three hash functions, updating the Bloom Filter. However, when we check if element *b* is present in the cache, all three hash functions are resolved to index which are marked 1 in the Bloom Filter. Thus, this is the case of *false positive,* where although an element is not present in the cache, still the indexes are resolved to 1.

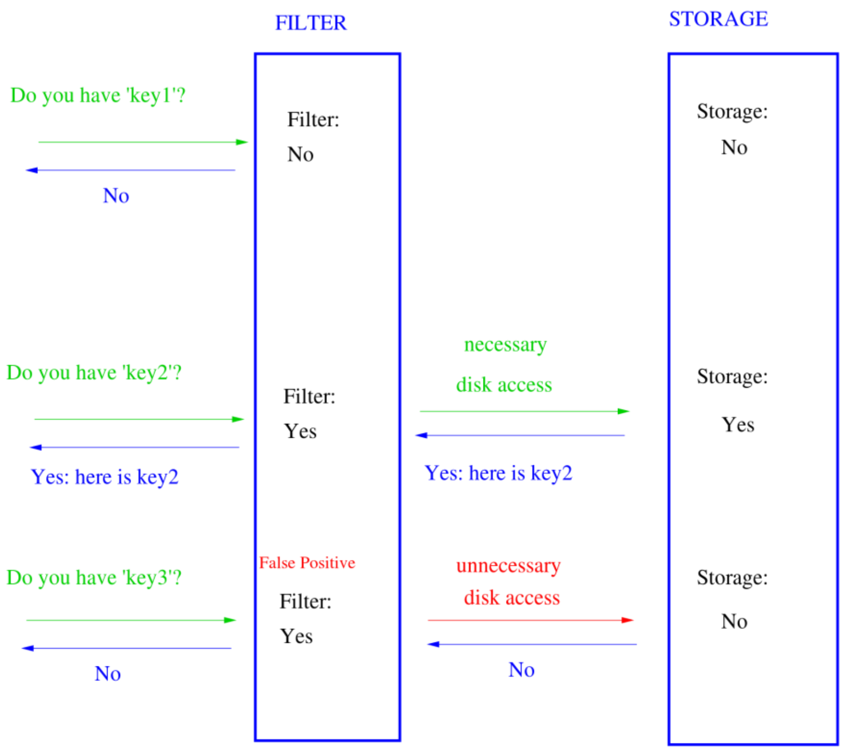
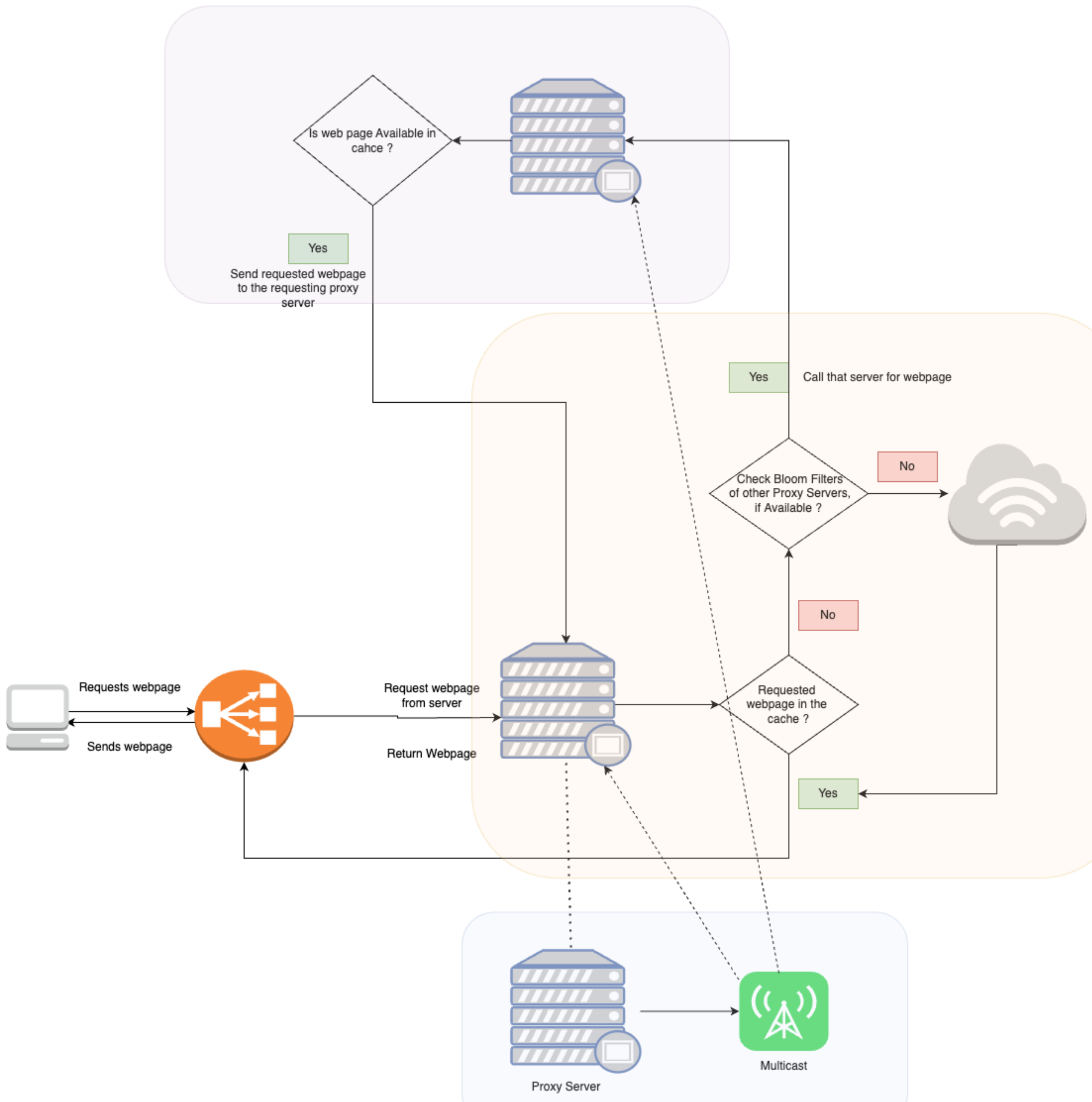


Figure: Showing Bloom filter used to speed up answers in a key-value storage system. Bloom filter. (2023, December 5). In Wikipedia. https://en.wikipedia.org/wiki/Bloom_filter

Technical Aspect

Architecture


Is web page Available in cahce ?
Yes
Send requested webpage to the requesting proxy server
Yes
Call that server for webpage
Check Bloom Filters of other Proxy Servers, if Available ?
No
No
Requested webpage in the cache ?
Yes
Requests webpage
Sends webpage
Request webpage from server
Return Webpage
Proxy Server
Multicast

Techniques:

1. A simple web client coded in Java using Maven requests the webpages.

2. Requests are distributed in round-robin fashion to Proxy Servers.
    2.1. Round-robin is implemented as a generic class instantiated using the Url of the next proxy servers.
    2.2. The RounRobin class has a method *next()*, which returns the URL of the next proxy server in sequence to be called.

3. The Proxy Server application is implemented in Java using Spring Boot, which can start multiple servers by configuring the multicast IP and port using the \`application.properties\` file.

4. If the requested webpage is available in cache of the proxy server, it directly returns webpage to the client.

5. Proxy Servers also maintain their individual bloom filters using *SHA-256* and the \`*BloomFilter*\` class. Each Proxy server multicasts its Bloom filter to other proxies on the prescribed multicast network.

6. If the webpage is not available in the cache of the requested Proxy Server, its existence is checked in the Bloom Filters of other proxy servers.
    6.1. If Bloom filter of some of the Proxy Server gives a possible positive result, then that Proxy Server is called for the webpage.

7. If the webpage exists in that Proxy Server, then webpage is sent to the requesting server and the first Proxy Server delivers the result to the client.

8. If the URL does not exist in the latter proxy server, then it fetches that from the internet, updates its cache and Bloom Filter.

9. The Bloom Filter is multicast every 30 seconds to keep the filters updated in other proxies; this is also configurable from the \`application.properties\`.

Github: *https://github.com/bsingh1597/bloom-filter-proxy.git*

Evaluation

We evaluated our project by comparing the theoretical probability of false positives of the bloom filter vs actual probability of false positives.

Dataset

We employed the dataset available on www.data.world, specifically at the following link: https://data.world/the-pudding/most-popular-links-shared-in-newsletters/workspace/file?filename=dump-2020-12-15.csv. This dataset includes links extracted from newsletters from June 18, 2020, to December 15, 2020, totaling about 150,000 records.

In our project, we extracted URLs that generated an HTTP response of 200 and utilized this subset.

Experimental setup

The theoretical probability of false positives in a Bloom filter can be calculated based on the size of the filter, the number of hash functions used, and the number of elements inserted into the filter. The formula for the false positive probability p i.s. derived from the assumptions of a perfect hash function and independence between hash functions.

For fixed filter size n , we calculated theoretical and actual probabilities for a different number of elements in the filter m .

We calculated the optimum number of hash functions for each combination of m and n using the below formula:

$$k = \frac{m}{n} \ln 2$$

Theoretical Probability of false positives [3]:

$$p = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

n : Number of elements in filter

m : Filter size

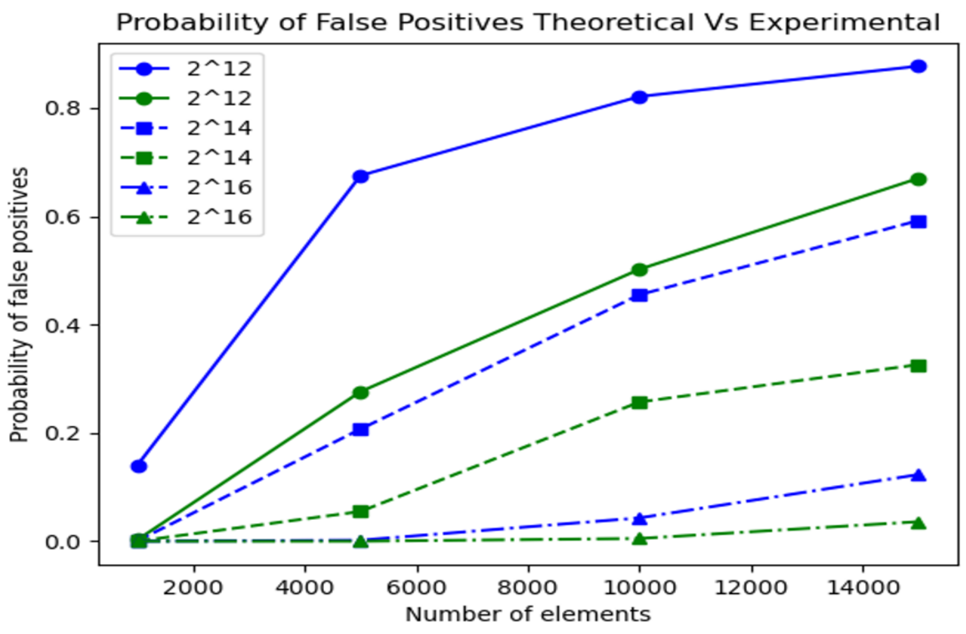
k : Number of hash functions

During the execution of the project, URLs are inserted into the Bloom filter, and set membership queries are performed. The observed or actual probability of false positives is determined by measuring the number of instances where the Bloom filter incorrectly indicates that an element is a member of the set when it is not.

Actual probability calculated as:

Average of *Number of false positive / Total number of requests made to proxy server*

Results



Blue Line indicates the Theoretical False +ve

Green Line indicates the Experimental False +ve

We can observe from the line chart that both probabilities exhibit similar trends. The variance between the theoretical and actual probabilities can be attributed to our approach to calculating hash functions. In this process, we opted for the ceiling value of the number of hash functions, for some of the scenarios we got the number of hash functions as 0.4 or 2.3 where we chose to take the ceiling value, thus contributing to the observed difference.

Although, the curve of both theoretical and experimental false positives is similar which provides positive insights to our implementation.

Conclusion

For the limited scope of this project, we did not include the security aspects of sharing bloom filters as there could be attacks by sharing malicious bloom filters in the pool. In conclusion, the proposed web client and proxy server system leverages caching, Bloom filters, and inter-proxy communication. The caching mechanism implemented by the proxy servers significantly reduces latency by storing previously requested URLs locally. This feature enables the servers to serve content directly to clients without making redundant calls to internet, resulting in a more responsive user experience. The incorporation of Bloom filters further enhances the system's efficiency by quickly determining the existence of a URL in the proxy server's cache. The inter-proxy communication through multicasting Bloom filters adds a layer of collaboration among proxy servers. This collaborative effort ensures that the knowledge of cached URLs is shared among servers, enhancing the overall cache hit rate and reducing redundant requests to the web.

References

1. Summary cache: a scalable wide-area Web cache sharing protocol
2. Network Applications of Bloom Filters: A Survey - Andrei Broder and Michael Mitzenmacher
3. Bloom filter. (2023, December 1). In *Wikipedia*. *https://en.wikipedia.org/wiki/Bloom_filter*