може бути масштабована та використана як основа для більш складних інтелектуальних платформ у сфері домашньої безпеки та моніторингу.

Список використаних джерел

1. Smart Home: Architecture, Technologies and Systems / M. Li та ін. Procedia Computer Science. 2018. Т. 131. С. 393–400. URL: https://doi.org/10.1016/j.procs.2018.04.219.
2. Lee K.-M., Teng W.-G., Hou T.-W. Point-n-Press: An Intelligent Universal Remote Control System for Home Appliances. IEEE Transactions on Automation Science and Engineering. 2016. Vol. 13, no. 3. P. 1308–1317. URL: https://doi.org/10.1109/tase.2016.2539381.
3. Puthiyidam J. J. IoT Smart Home: Protocols and Architectures. International Journal for Research in Applied Science and Engineering Technology. 2017. Vol. V, no. XI. P. 2031–2038. URL: https://doi.org/10.22214/ijraset.2017.11293.

DOI 10.70286/ISU-03.12.2025.004

A METHOD FOR CONSTRUCTING A DISTRIBUTED BLOOM FILTER WITH CONSISTENT HASHING

Zhovtaniuk Maksym
Master student
Oleksii Molchanov
Ph.D.
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"

Modern information systems that operate with large amounts of data often encounter the issue of quickly checking the presence of an element in a dataset. Common approaches, such as the use of hash tables, work well in small systems but lose their properties when dealing with large amounts of data and the need for high throughput. Modern technology companies like Google, Amazon often need to check whether a user is already registered in their system or not. This operation has to be as fast as possible to maintain a good user experience. Other issue is that such a presence check has to be performed on the whole set of data, which is very time-costly [1]. That's why in practice, such systems use a data structure called a Bloom Filter [2], which acts like an index. A Bloom Filter is a probabilistic data structure that can precisely identify that an element is not present or is possibly present in a given data set. To create a Bloom Filter user has to have an approximate amount of elements and a false positive rate, which could be tolerated. In the classical approach, once a data structure is created, its size cannot be dynamically changed. A Bloom filter is represented as a bit array, to insert an element, K amount of hash functions have to be applied to it to identify positions which have to be set in the bit array. To check the

presence, the same set of hash functions is used, and if all of the bits are set, the element is probably in a set. Probability is achieved through collisions of bit positions for different elements (figure 1).

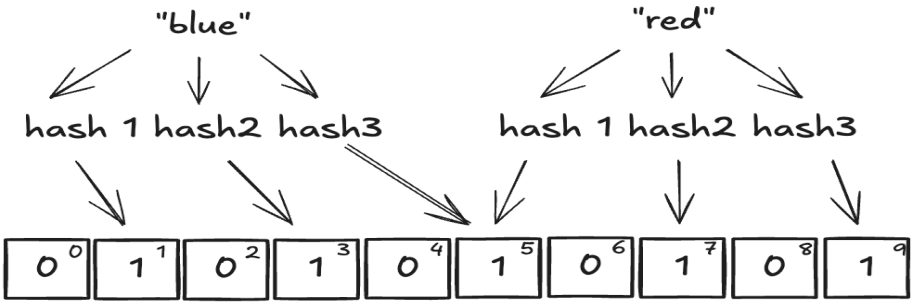

Figure 1. A schema of a classical Bloom Filter

Such an approach is more efficient in terms of RAM usage and requires less computational power. This is suitable for cases when a user can tolerate some degree of false positive results.

Even though Bloom Filter solves most cases, in cases of very large datasets (i.e., millions and billions of elements), the system becomes dependent on vertical scaling, requiring a large amount of RAM on a single physical machine. Although such systems are easier to maintain, they are more likely to completely shut down in case of errors on a single node.

In this study, a method for the construction of a scalable distributed bloom filter system is presented. The system is represented as a set of nodes, each containing some part of a single logical Bloom Filter. To distribute keys between nodes, a consistent hashing [3] method is used, where nodes are placed on a ring, and each segment of the ring is transferred to a particular node. This mechanism allows to dynamic change in the number of nodes without full system rebalancing (figure 2).

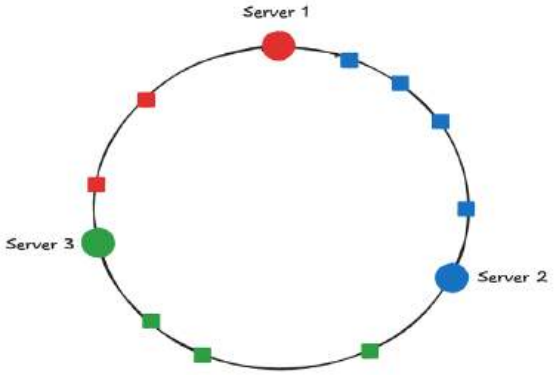

Figure 2. A schema of a distributed bloom filter

To direct an element to a node, a hash function has to be applied to an element, which returns a position on ring, clockwise next node on a ring must contain a given element. To distribute elements evenly, it is crucial to choose a proper hash function. Another requirement is that hash computation has to be as fast as possible. That's why MurmurhHash3 [4] was chosen for the method.

To maintain a correct system state, a coordination service called Zookeeper is implemented. Its main responsibilities are:

- Read input data and prepare local bloom filters on each node using protocols like HTTP/GRPC.
- Perform health checks of each node.
- Dynamically change the number of nodes when needed.
- Monitor nodes' metrics like: CPU utilization, RAM usage, network load etc.

For communication with the system, external applications have to use the SDK that performs the following functions:

- Communication with the Zookeeper service to monitor an actual system state.
- Receiving an element, apply the hash function.
- Identifying the node that contains the given element.
- User replication in case the primary node is unavailable.

To maintain the system's fault tolerance, each node can have a replication node, which copies the state of the primary node through the network, i.e., sends a request to copy the local Bloom Filter. The replication node is registered in both the Zookeeper service and SDK. When the primary node is down, the user has the ability to switch to a replica and doesn't lose the system's properties in case of physical outages (figure 3).

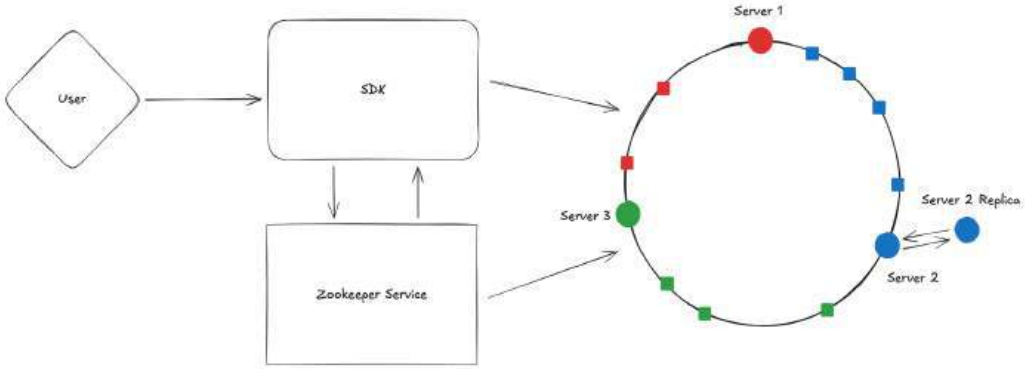

Figure 3. A scalable distributed Bloom Filter system design

The most important aspect of any system is to have high throughput and low latency for high load cases. For the system that implements the previously described schema, a load test was performed to identify potential bottlenecks and analyze the system's performance.

One of the most important metrics in highload systems are request rate per second and request latency. Under the conditions of 3-5k requests per second, result of the tests show that request latency is under 5ms (figure 4).

Figure 4. Dashboards. Request Rate per second and request latency

Other important technical metrics are CPU utilization and RAM usage. Under the same conditions, it was shown that there were no significant load on CPU and memory usage was under 45% (figure 5).

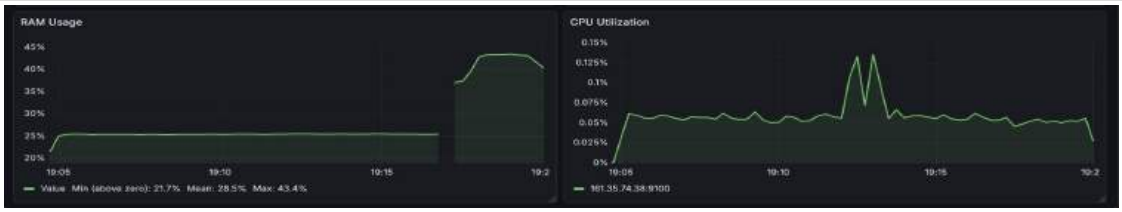

Figure 5. RAM usage and CPU Utilization

In conclusion, implemented system works as expected, all functions of classical Bloom Filter were preserved. Experimental tests showed that the system has high performance under heavy load.

References

1. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 205–218. https://doi.org/10.1145/1365815.1365816
2. Bloom, B. H. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM. vol. 13, no 7. pp. 422–426. https://doi.org/10.1145/362686.362692
3. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D. (1997). 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. ACM Press New York, NY, USA. pp. 654–663. https://dl.acm.org/doi/10.1145/258533.258660
4. Appleby, A. (2016). SMHasher. Github. https://github.com/aappleby/smhasher

ІНТЕЛЕКТУАЛЬНА МОДЕЛЬ ВИЯВЛЕННЯ КІБЕРІНЦИДЕНТІВ У SIEM-СИСТЕМІ ЗА ДОПОМОГОЮ ТЕХНОЛОГІЙ НЕЧІТКОЇ ЛОГІКИ

Анна Булковська
здобувач вищої освіти магістерського рівня
https://orcid.org/0009-0005-4322-3495
Військовий інститут телекомунікації та
інформатизації імені Героїв Крут

У сучасних умовах стрімкого розвитку цифрових технологій та зростання рівня кіберзагроз, що активно використовуються у військових, політичних та інформаційно-психологічних операціях, особливої актуальності набуває проблема своєчасного виявлення кіберінцидентів. Системи класу SIEM (Security Information and Event Management), які виконують функції централізованого збору, кореляції, нормалізації та аналізу подій безпеки, залишаються базовим