2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS) | 979-8-3503-6387-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACSOS61780.2024.00031

Load Balancing and Generalized Split State Reconciliation in Event Driven Systems

Ratnadeep Bhattacharya
Department of Computer Science
George Washington University
ratnadeepb@gwu.edu

Timothy Wood
Department of Computer Science
George Washington University
timwood@gwu.edu

***Abstract*—Event driven applications are often built with message queuing systems that provide no temporal upper bound on message delivery. However, many modern event driven applications, like a system inferring traffic conditions and generating recommendations to road users based on sensor data, are latency sensitive. Traditional message queuing systems use static load assignment algorithms that guarantee event ordering while mostly ignoring a temporal upper bound on message delivery. Another class of message queuing systems use stateless operators which deliver messages (events) quickly but pass the burden of stream state management to user applications. Synchronous communication patterns, on the other hand, provide an upper bound for message delivery while ensuring message ordering but unnecessarily bind limited resources reducing efficiency.**

In this paper we explore load balancing choices in asynchronous systems and their impact on queuing delay. We then propose a load balancing framework, SMALOPS, for event driven applications with dynamically changing load and quick message delivery requirements. Our experiments confirm that with smarter load balancing, the 99%ile response times for events can be improved by as much as 73%, compared to traditional message queuing systems. SMALOPS introduces the following:

- **A load balancing algorithm that can significantly reduce queuing delay in message delivery systems.**
- **Mechanisms enabling consumers to recover stream state when either the message delivery system does not support stateful operators or the state has been split by moving streams between operators.**

***Index Terms*—stream load balancing, asynchronous, flow ordering, split state reconciliation, stream processing**

I. Introduction

Message queues are a popular tool for building event driven services. Traditional asynchronous services have a fire and forget model where messages tend to trigger a change of state of some downstream application(s). Usually, queuing delay is ignored since typical asynchronous applications have a high tolerance for state change latencies. For example, when operating one's bank account, it is often acceptable for transactions to remain in "transit" for days without updating the state of the account. In other cases, the state change latency requirements are relaxed as such a state might be hidden from users, such as in log processing applications. As far as message delivery times are concerned, message queuing systems make no guarantees.

Some message queuing systems provide strong message ordering guarantees leading such systems to disregard skewed load build-up. Further, most message queuing systems treat load balancing and stateful stream operations as orthogonal design choices. Systems like Apache Kafka assign load statically for the lifetime of the streams. While this prevents against split stream state, such a system is unable to handle significantly skewed input at a massive scale [1], [2]. Other message queuing systems like NSQ [3] offer a better scaling profile by foregoing stateful stream processing completely.

There is an emerging class of applications that use message queuing systems to integrate physical systems to data intensive applications [1]. These applications operate on streams of data bounded by the notion of "freshness" imbibed in the messages that constitute the streams. Informally, we define this "freshness" to be a temporal bound beyond which the information carried by the message looses its relevance. Mostly, such freshness of messages can typically be guaranteed by splitting the streams and balancing them across multiple message queues, where the queues are stateless operators. If an ordering of the messages are required then it is up to the consumers of the messages to rebuild that ordering. Generally speaking, if the messages originate from a multitude of sources and are split across a variety of processors, then finding a consistent, strict order is a very hard challenge. On the other hand, building data intensive cyber-physical systems using synchronous patterns, where the source of the data waits for a response to the request before sending the next request, is wasteful of the limited computing resources available in physical systems.

In this work, we present SMALOPS, which makes two primary contributions:

- A load balancing algorithm, focusing on the heavier flows (in terms of messages per unit time) thus reducing state tracking overhead, that is able to respond to dynamic workload changes.
- Such an algorithm affects a split in the streams that it operates on, thus we also discuss a mechanism to rebuild message ordering to support stateful stream processing.

SMALOPS optimizes for queuing delay while being cognizant of the desirability of stream state and fully aware of possible migrations of local application state stored in the downstream services. Finally, SMALOPS is also biased towards keeping larger streams pinned and migrates them only

979-8-3503-6387-6/24/$31.00 ©2024 IEEE
DOI 10.1109/ACSOS61780.2024.00031

when absolutely required. As per our knowledge, SMALOPS is the first framework of its kind.

We implemented SMALOPS as a Go based framework on top of Apache Kafka v3.3.1 and deployed it in a Kubernetes cluster. Our evaluations show that SMALOPS can improve the 99th%ile queuing latency for streams by as much as 73%. Though SMALOPS is built on top of Apache Kafka, it is not dependent on any Kafka specific features. We demonstrate the principles behind SMALOPS in a manner completely agnostic of the underlying system.

II. System Design

We refer the frequency with which messages occur in a stream as its "size". Based on this definition, we classify streams as "heavy" when their size is above some predefined threshold. The size of a message broker queue ("partitions" borrowing notation from Kafka), is defined as the sum of the sizes of the heavy streams assigned to that partition. We refer to the key associated with a "heavy" (or large) stream as a "hot key". We use the lossy counting algorithm [4] to detect these heavy streams. The primary idea behind SMALOPS is to detect and manage the load from the heavy streams. The initial assignment of heavy streams to a broker instance are done by using the power-of-two random choices algorithm [5]. The reassignment problem can generally be posed as a linear integer programming problem and is known to be NP-complete [6], [7]. In this paper we present a heuristic to use "best-fit" to solve this problem. Heavier keys, with relatively smaller size, have more infrequent messages and as such can tolerate larger relative delays. Thus we biased the heuristics to migrate these streams first.

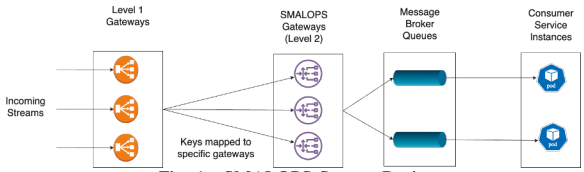


Fig. 1. SMALOPS System Design

The overall system design of SMALOPS is shown in Fig. 1. SMALOPS is a gateway service that maps incoming keys into Kafka partitions. We assume, without loss of generality, that multiple instances of the same consumer service are listening to these messages. We introduce another layer of gateway service in front of the SMALOPS gateways, "Level 1 Gateways", to map specific set of keys to specific SMALOPS gateways. This additional gateway layer is mainly introduced to avoid distributed algorithms to detect heavy flows across multiple gateway nodes, [8], [9]. Level 1 gateways can use load balancing algorithms described in [10], [11] or [12], which we leave for future research. In this paper we focus on reducing queueing delays in the message broker queues.

SMALOPS balances the need to optimize end-to-end latency against minimizing the need to rebuild state in streams. Given a set of partitions, $P = \{p_1, p_2, ..., p_m\}$ and a set of keys K, let R_{p_i} be the sum of the sizes of the heavy keys that have been assigned to the partition p_i, the size of the partition. Then the aim of SMALOPS is to achieve:

$$R_{p_i} \approx R_{p_j} \forall i, j \in \{1, ..., m\} \wedge i \neq j \tag{1}$$

We further define an error function for balancing the partitions,

$$E_{LB} = \sum_j |R_{p_j} - R_{avg}| \tag{2}$$

where E_{LB} is the load balancing error, R_{avg} is the average load across all partitions and R_{p_j} is the rate of incoming messages due to hot keys into the partition p_j. Simply speaking, we want assign equal load due to heavy streams to each partition.

Further we divide the partition in two sets - the "underloaded" and the "overloaded" set based on whether a partition's size less or more than R_{avg}. We then look at each heavy stream in each partition of the overloaded set and attempt to migrate the stream to a partition in the underloaded set. SMALOPS aims to bring both partitions nearer to R_{avg} eventually optimizing Equation 2. This is a greedy algorithm that attempts to migrate, one by one, each of the heavy streams in the overloaded partitions to an underloaded partition. We define the difference in size between two partitions as "load distance", refer to Fig. 2. If the size of the selected stream is R_m and the size of a potential target partition is $R_{p_{target}}$, then this algorithm will select a partition that minimizes the following expression:

$$|(R_m + R_{p_{target}}) - R_{avg}| \tag{3}$$

If for every possible target partition, the result of migrating the stream is an increase in the load distance between the source and target partitions, then the algorithm will not migrate that stream.

However, migrating a stream from one partition to another destroys the ordering of messages within that stream. To resolve this issue, we define subdivisions within a stream called message sets. Each stream is now subdivided into sets which have a unique ID within the stream that increments every time the stream is reassigned, Fig. 3. This information is encoded within the stream metadata, Fig 4. Every time a stream is migrated, a new message set is started. Thus the problem of reordering the messages reduces to a problem of reordering the message sets. Towards that goal, the message

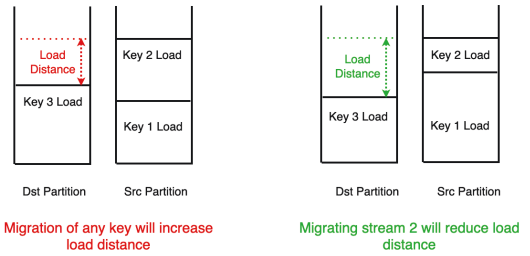


Fig. 2. Migration Feasibility

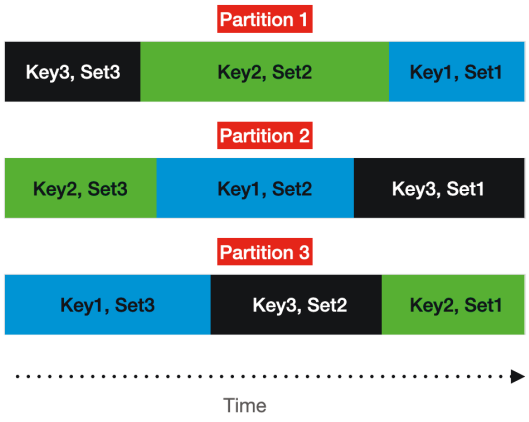


Fig. 3. Message Sets

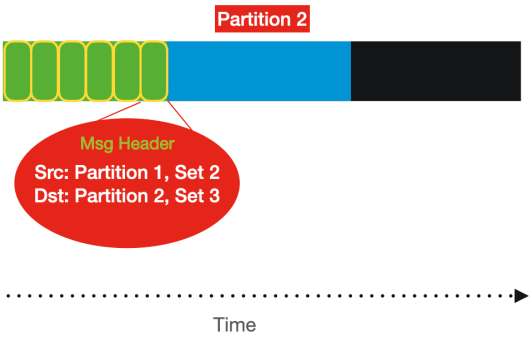


Fig. 4. Message Set Header

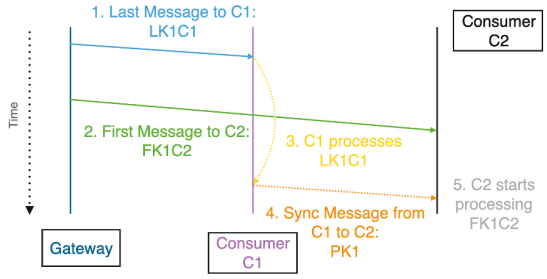


Fig. 5. Consumer side protocol

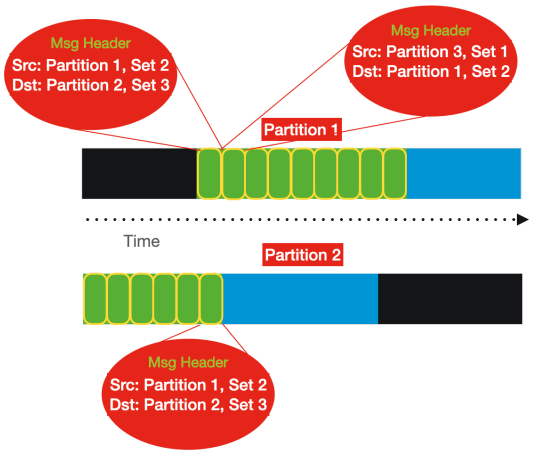


Fig. 6. Message Set Migration

set header includes a unique identifier, within the stream, with a total ordering.

Rebuilding the state of the stream requires some effort on the consumer side since the migrated stream is now sending messages to a different consumer instance. When a stream S is migrated from consumer instance C1 to consumer instance C2, the message set headers will contain the following information:

- The message set header on the last message of stream S sent to consumer application C1 will inform C1 that stream S is being migrated to C2.
- The message set header on the first message of stream S sent to consumer application instance C2 will inform C2 that the stream S is being migrated from C1.

When C2 receives the messages of S, it queues them internally without processing them. In the meantime, C1 finished processing the messages of S. It then sends a message to C2 informing that it has finished processing all messages of S that was sent to it. C1 then starts processing messages belonging to S. Fig. 5 shows the timeline of the gateway and the two consumers, C1 and C2, involved in the migration.

SMALOPS needs to maintain some state for the "heavy" streams. Specifically, SMALOPS needs to track the partition each migrated stream is mapped to and it needs to track the "size" associated with each hot key and partition. In order to reduce the amount of state that needs to be maintained, SMALOPS only tracks keys that have currently been detected as hot. Any key that was previously detected as hot but is not associated with a heavy flow anymore is dropped. In a long running experiment where keys sending 100+ messages in any counting window were defined as "hot" and $|K|$ was defined to be 28M (million), we found that our custom power-law, open-loop generator actually used 3.2M keys while sending a total of 28M messages. Out of these 3.2M keys, 14K (thousand) were detected as being "hot" over the course of the experiment. These 14K keys accounted for 20M+ of the 28M total messages sent by the generator. The SMALOPS gateway tracked no more than 1-1.5K keys in any of their counting windows. Thus we can reasonably expect SMALOPS to balance a significant portion, $\frac{\sum_{j=1}^{H} R_j T}{m c_p} \approx \frac{3}{4}$, of the load on the messaging system while only maintaining a very small state, tracking only $\sim 0.05\%$ of the number of keys seen. Here H is the number of hot keys in the gateway, m is the number of partitions, R_j is the size of each of the hot keys, T is the threshold above which a partition is considered heavy and c_p is the average load generated by the non-hot keys.

SMALOPS also considers the fact that consumers may maintain local application state, "appstate", associated with each stream. Migrating a stream to a different partition prob-

ably migrates it to a different consumer as well, potentially triggering an appstate migration. In such cases, SMALOPS needs to be aware of the state migration costs. Let us consider an example. Let us assume that processing time for a message from any stream is t_{msg}. Let K_{p_1} be the set of all keys whose messages are sent to partition p_1 and thus to the consumer C_1. Also, let R_{k_n} be the incoming rate of a stream, identified by key $k_n \in K_{p_1}$. Then the processing and queueing time for a message belonging to that stream can be approximated by:

$$t_{msg}(1 + (1 - \frac{R_{k_n}}{R_{p_1}}))$$

where the last term is the amount of time a message belonging to the stream identified by the key k_i has to wait in a message queue before being read.

Now let us consider if the stream identified by key K_1 is migrated to another partition p_2 and thus consumer C_2. Let K_{p_2} be the existing set of hot keys on partition p_2 and R_{K_j} be the corresponding rates of incoming messages. Thus the processing and queueing time for a message belonging to the stream identified by key K_1 on partition p_2 can be approximated by:

$$t_{msg}(1 + (1 - \frac{I}{R_{p_2}}))$$

Here I denotes a (unit) rate of incoming messages, $\frac{1}{\text{unit time}}$, since we are only interested in how long the first message has to wait in the message broker.

Finally, let the cost of migrating the appstate corresponding to key K_1 be t_{SM}. Then migrating the stream makes sense only if,

$$t_{msg}(1 + (1 - \frac{R_{k_n}}{R_{p_1}})) > t_{SM} + t_{msg}(1 + (1 - \frac{I}{R_{p_2}})) \quad (4)$$

In order to handle local consumer state, the gateways need to track **rate** of incoming messages and the **cost of state migration**. This tracking would significantly bloat the state stored in the gateways. This **state bloat** can be completely avoided if the stream migrations are made at discrete time intervals greater than some threshold that ensures that all state migrations have completed before new migrations are started. In our experiments, we choose this decision time interval to be 10 seconds which is an order of magnitude higher than the processing time of the messages.

III. Evaluation

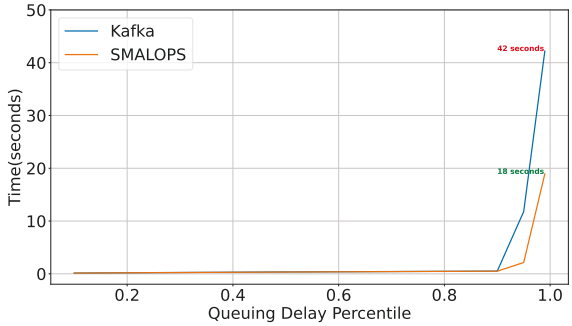
A. Experimental Setup

We ran our experiments on cloudlab [13] servers. A Kubernetes cluster was created with four Intel Xeon servers, each with 20 cores and 196GB of memory. We then deployed our control plane that ran a pod on each of the servers. These pods form the service that is queried to get information about the backends of the gateway service running in the cluster.

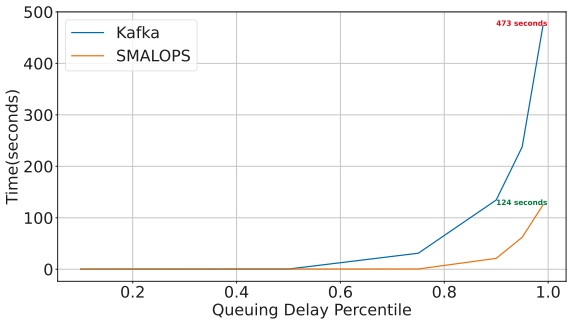
We use an in-house power-law based open loop generator. We use the tool to send requests to the gateway for a fixed amount of time (5 minutes) where every request is part of a flow, identified by a key selected using the Zipf distribution. We also experimented with messages without any key metadata that could be routed on a per message basis to get a measure of what an optimal performance on a single gateway might look like.

We used the Zipf distribution to generate load due to its prevalance in different workload like Twitter hashtags [14] and web caches [15]. We define the Zipf distribution as per $k \in [0, num_{keys}] : P(k) \propto (v+k)^{-s}$, where $s > 1$ and $v \geq 1$. Our generator uses $s = 1.1$ and $v = 2.72$ [16].

The core motivation for designing the experimental setup was to track the queuing delay of individual events in Apache Kafka given a workload that follows a power law distribution.



(a) Queuing Delay Distribution Improvement at 20 μs



(b) Queuing Delay Distribution Improvement at 50 μs

Fig. 7. Performance improvement with SMALOPS is realised earlier

B. SMALOPS Overall Performance

In our first experiment, we set a processing time of 50 microseconds for each message on the consumers and compare the queuing delay distribution achieved through Kafka and SMALOPS respectively. Here we see that SMALOPS improves overall native Kafka performance, with the 99%ile queuing delay improving by 73%, Fig. 7(b).

We repeated this experiment varying consumer processing times between 10 and 50 microseconds. The comparison between the 99%ile queuing delay can be seen in Fig. 8. We note that as the system starts getting overloaded, the 99%ile latency of Kafka keeps increasing while the improvements achieved by

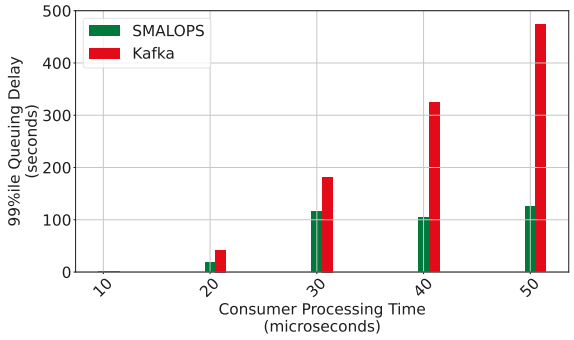


Fig. 8. 99%ile Queuing Delay Comparison

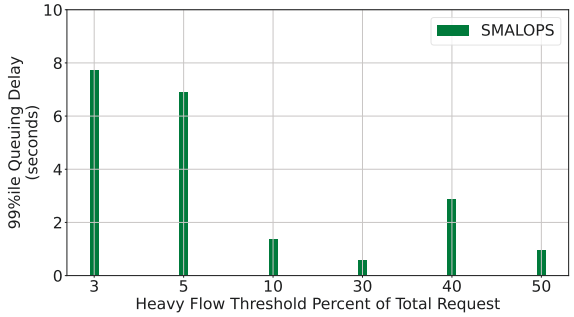


Fig. 9. Impact of Threshold Definition on Latency

SMALOPS remain stable. The 99%ile improvement at 50 microseconds consumer processing time is 73% while the 99%ile queuing delay improvement at 20 microseconds consumer processing time is 57%. In our experiment, 50ms processing time/message was just before the system is overloaded (tail latencies become really long).

Furthermore, we see that as the system starts getting overloaded, Kafka's performance start degrading earlier. For example, for a 20 microseconds consumer processing time, the performance of Kafka and SMALOPS are indistinguishable till 90%ile latency. However, for a 50 microseconds consumer processing time, Kafka starts degrading around the 50%ile mark while SMALOPS's performance remains nearly identical. This comparison between the performances of SMALOPS and Kafka can be seen in Fig. 7.

In Fig. 9 we experiment with the threshold, as a percentage of total number of messages, to be used to classify a flow as "heavy". We see that when we classify flows that account for 10% or more of the messages received as "heavy", SMALOPS' tail latency drops sharply. We also see that further increasing the threshold has no significant impact on our results. Rather keeping this threshold as low as possible allows us to keep the number of keys being tracked lower.

Finally, we ran SMALOPS against "dynamic load", where the load is generated by shifting the keys through our zipf based generator every minute. This ensures that:

- A few new streams are introduced to the system regularly.

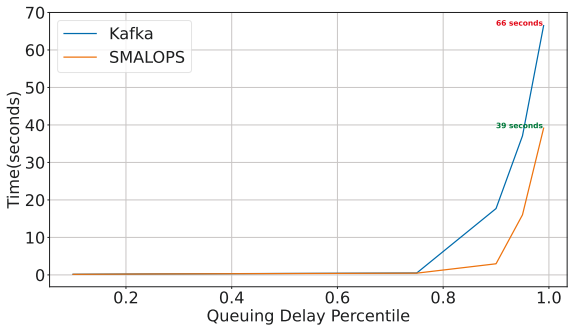


Fig. 10. Kafka vs SMALOPS under dynamic load conditions

- A few streams stop altogether.
- Load on remaining streams change.

Fig. 10 shows that SMALOPS can significantly improve streaming system performance even in the face of these new challenges. Each message in this experiment takes 20ms to be processed.

IV. Related Work

Load balancing is a well known problem that has been extensively studied for a long time. Advent of modern distributed systems has renewed interest in the area. Load balancing in the topic based pub-sub systems like Apache Kafka have generally been in two forms:

- Balance the partitions themselves within the brokers [17].
- Assign topics to partitions as the topics enter the system [18], [19].
- Build overlay networks [20], [21] to connect relevant nodes directly.

Gedik et al. [22], have used the lossy algorithm to track heavier flows and map those to partition explicitly. Other flows are mapped using the consistent hash function. This work is probably the most similar to ours. They use three lossy counters over tumbling windows to emulate a sliding window whereas we only use a single lossy counter over strictly demarcated window.

Nasir et al. [23], has proposed **PKG** that uses power of two random choices to map each key to the least loaded partition selected by two different hash functions. This results in every flow, heavy and otherwise, being in a split state that requires reconciliation.

Finally, Rivetti et al. [24], proposed **DKG** that learns the distribution of the keys before using a global mapping function to achieve near-optimal load assignment. They map non-heavy keys to "buckets", where the number of buckets is user-defined and larger than the number of partitions, using a random hash function from two separate hash function families. Their solution maps the heavier flows, identified by the space saving algorithm, explicitly to specific partitions.

SMALOPS extends existing research in fundamental ways:

- SMALOPS tracks and load balances only the heavy hitters since with the zipfian distribution the majority of

the system load come from a small number of heavier flows.

- SMALOPS focuses on dynamic workload distributions in two important ways:
  - SMALOPS accounts for the fact that different keys produce heavy streams at different times.
  - SMALOPS realizes that not every heavy hitter is active at the same time. This allows SMALOPS to significantly reduce the number of flows it is tracking.
- SMALOPS balances the gains from migrating streams against the cost of rebuilding stream state or migrating applications state.

V. Conclusions and Future Work

To our knowledge, SMALOPS is the first system that considers dynamic load while proposing a generalized split state reconciliation. Moreover, SMALOPS achieves stable performance even under higher loads while dealing with dynamic changes in existing streams. SMALOPS can reduce 99%ile latency in message queuing systems to almost a constant factor. Migrating stateful streams disrupts their state and SMALOPS provides a mechanism by which consumers can rebuild the state. Our future research would investigate SMALOPS as a comprehensive message broker that would provide both the load balancing and the required state reconciliation within the system without any participation from consumers.

Acknowledgements: This work was supported in part by NSF Grant CNS-1837382.

References

[1] T. P. Raptis and A. Passarella, "A Survey on Networked Data Streaming With Apache Kafka," *IEEE Access*, vol. 11, pp. 85 333–85 350. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10213406

[2] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8509390

[3] NSQ Docs 1.3.0 - A realtime distributed messaging platform. [Online]. Available: https://nsq.io/

[4] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1699, aug 2012. [Online]. Available: https://doi.org/10.14778/2367502.2367508

[5] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, p. 1094–1104, oct 2001. [Online]. Available: https://doi.org/10.1109/71.963420

[6] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems theoretical and practical results," *J. ACM*, vol. 34, no. 1, p. 144–162, jan 1987. [Online]. Available: https://doi.org/10.1145/7531.7535

[7] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM J. Comput.*, vol. 7, no. 1, p. 1–17, feb 1978. [Online]. Available: https://doi.org/10.1137/0207001

[8] H. Dai, M. Li, A. X. Liu, J. Zheng, and G. Chen, "Finding persistent items in distributed datasets," *IEEE/ACM Transactions on Networking*, vol. 28, no. 1, pp. 1–14, 2020.

[9] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *21st International Conference on Data Engineering (ICDE'05)*, pp. 767–778. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1410191

[10] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. Association for Computing Machinery, pp. 168–181. [Online]. Available: https://doi.org/10.1145/3472883.3487014

[11] R. Bhattacharya, Y. Gao, and T. Wood, "Dynamically balancing load with overload control for microservices," *ACM Trans. Auton. Adapt. Syst.*, jul 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3676167

[12] R. Lösch, J. Schmidt, and N. G. Felde, "Weighted Load Balancing in Distributed Hash Tables," in *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*, ser. iiWAS2019. Association for Computing Machinery, pp. 473–482. [Online]. Available: https://doi.org/10.1145/3366030.3366069

[13] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of cloudlab," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 1–14.

[14] J. A. Pérez Melián, J. A. Conejero, and C. Ferri Ramírez, "Zipf's and Benford's laws in Twitter hashtags," in *Proceedings of the Student Research Workshop at the 15th Conference of the European Chapter of the Association for Computational Linguistics*, F. Kunneman, U. Iñurrieta, J. J. Camilleri, and M. C. Ardanuy, Eds. Association for Computational Linguistics, pp. 84–93. [Online]. Available: https://aclanthology.org/E17-4009

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future Is Now (Cat. No.99CH36320)*, vol. 1, pp. 126–134 vol.1. [Online]. Available: https://ieeexplore.ieee.org/document/749260

[16] S. T. Piantadosi, "Zipf's word frequency law in natural language: A critical review and future directions," *Psychonomic Bulletin & Review*, vol. 21, no. 5, pp. 1112–1130, Oct 2014. [Online]. Available: https://doi.org/10.3758/s13423-014-0585-6

[17] D. . Burato, "Load balancing and fault early detection for Apache Kafka clusters," *Load balancing and early fault detection for Apache Kafka clusters*. [Online]. Available: http://dspace.unive.it/handle/10579/15159

[18] T. P. Raptis and A. Passarella, "On efficiently partitioning a topic in apache kafka," in *2022 International Conference on Computer, Information and Telecommunication Systems (CITS)*, 2022, pp. 1–8.

[19] D. Dedousis, N. Zacheilas, and V. Kalogeraki, "On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 76–86.

[20] C. Chen, H.-A. Jacobsen, and R. Vitenberg, "Algorithms Based on Divide and Conquer for Topic-Based Publish/Subscribe Overlay Design," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 422–436. [Online]. Available: https://ieeexplore.ieee.org/document/6971250

[21] V. Turau and G. Siegemund, "Scalable Routing for Topic-Based Publish/Subscribe Systems Under Fluctuations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1608–1617. [Online]. Available: https://ieeexplore.ieee.org/document/7980098

[22] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517–539. [Online]. Available: https://dl.acm.org/doi/10.1007/s00778-013-0335-9

[23] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *2015 IEEE 31st International Conference on Data Engineering*, pp. 137–148.

[24] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, "Efficient key grouping for near-optimal load balancing in stream processing systems," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. Association for Computing Machinery, pp. 80–91. [Online]. Available: https://dl.acm.org/doi/10.1145/2675743.2771827