A Proxy-based Query Aggregation Method for Distributed Key-Value Stores

Daichi Kawanami*, Masanari Kamoshita†, Ryota Kawashima‡ and Hiroshi Matsuo§

Nagoya Institute of Technology, in Nagoya, Aichi, 466-8555, Japan
Email: *kawanami@matlab.nitech.ac.jp, †kamoshita@matlab.nitech.ac.jp, ‡kawa1983@ieee.org, §matsuo@nitech.ac.jp

***Abstract*—Distributed key-value stores (D-KVS) are critical backbone for SNS and cloud services. Some D-KVS are based on a ring architecture with multiple database nodes to handle large amount of data. Any of them can receive queries from clients, and the node forwards queries to an adequate node if necessary. Therefore, this architecture causes heavy overhead of packet processing for each node. Some D-KVS have adopted fast packet processing frameworks like DPDK, but this is not enough to handle huge amount of requests. We introduce a query aggregation method to D-KVS to reduce the network traffic. In our approach, client queries are aggregated into a few large-sized query packets by a centralized proxy. The proxy receives every query from the clients, and it routes aggregated queries to the destination nodes. The proxy is built on top of DPDK-based network stack and can deal with the growing of the clients by increasing the number of CPU cores for packet handling. We evaluated with the environment of three Cassandra nodes linked with 10 Gbps network. Our approach improved throughput by 19% compared with the non-proxy Cassandra.**

***Index Terms*—Distributed Key-Value Store, Aggregation, Query Routing, Proxy, DPDK, Many-core**

I. Introduction

Distributed key-value stores (D-KVS) are one of critical backbone for social networking services and cloud services [1]. D-KVS distributes the load among the database nodes to handle large amount of data. Some D-KVS, such as Apache Cassandra [2] and Amazon Dynamo [3], use a ring architecture that any node can be a coordinator for the clients, and the node forwards the query to the appropriate node getting or storing the requested data. In this architecture, the clients do not have to care about the location of the data, but the overhead of query forwarding increases.

Clients could directly send their queries to the appropriate nodes to eliminate the forwarding. However, additional information (e.g. the assignment of the keys) is required to find out the destination nodes, which can degrade entire throughput of D-KVS. KVS workloads can be large amount of small-sized key-value pairs [4] and their queries consist of a lot of small messages. Many existing studies [5] [6] [7] have shown that the network stack can be performance bottleneck for such workloads because costly processing like context switch and packet copy between the kernel and the user spaces is required for each packet.

Some D-KVS have adopted fast packet processing frameworks like DPDK [8] to mitigate the processing overhead. ScyllaDB [9] uses a fast user-space network stack based on Seastar [10] and DPDK, and its throughput is 10 times higher than that of Cassandra [11]. However, the existing study [12] has reported that using a fast packet I/O is not enough to achieve the wire-rate of the high bandwidth networks (over 10 Gbps). Therefore, further improvement of packet processing efficiency is required.

Packet aggregation is a well-known technique to increase the network performance by reducing the number of packets. NetAgg [13] decreases the traffic amount of MapReduce and distributed search engine, and improved throughputs of Hadoop and Solr by up to 5.2 times and 9.3 times respectively. PA-Flow [14] achieved 1.7 times higher throughput than that of a DPDK-based system by aggregating consecutive outgoing packets. These studies showed that packet aggregation is a key to efficiently handle huge amount of small-size requests, and we believe that this approach is effective for D-KVS involving lots of small key-value queries.

In this paper, we propose a proxy-based query aggregation method. The proxy is introduced between the clients and the database nodes. Clients communicate with the proxy and the client queries are aggregated into a few large-sized query packets by the proxy. The proxy parses the query and calculates the destination because the same destination queries should be aggregated to decrease the forwarding. The proxy is built on top of DPDK-based network stack and many-core CPU. Using DPDK mitigates the overhead of the query forwarding, and the proxy can deal with the growing of the clients by increasing the number of CPU cores for packet handling.

Even though the proxy incurs inevitable forwarding, query aggregation brings far more performance merit by the reduction of the total number of packets. In addition to improving of performance with proxy, this centralized architecture enables exclusion control without inter-node communication. We focus on the effect of a proxy-based query aggregation in this paper.

The remainder of the paper is organized as follows. We explain a basic architecture of D-KVS and its problem in Section II. In Section III, we present the related work. In Section IV, we explain the proxy-based query aggregation method. The implementation of our proposal is described in Section V. In Section VI, we evaluate the throughput of our proposed architecture with the proxy. Finally, we present conclusions and future work in Section VII.

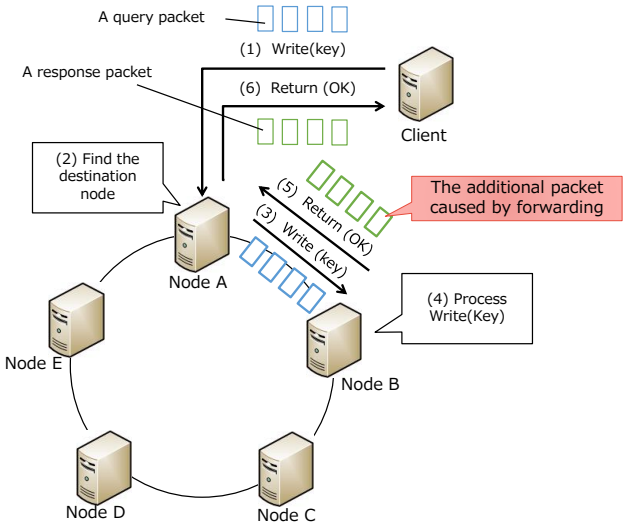

Fig. 1. A ring architecture

II. The Architecture of D-KVS

D-KVS can be categorized by how to distribute data to each node. GFS [15], a storage infrastructure of Google Bigtable [16], uses a master-slave architecture. In this architecture, the master node is used to place the data and monitors the data node. Clients ask the master node which data node has the requested data. This architecture has a crucial performance concern such that the requests concentrate on the master node. Amazon Dynamo [3] and Apache Cassandra use a ring architecture to avoid the bottleneck. In this architecture, all nodes can act as both a master node and a data node. Therefore, the aforementioned performance bottleneck can be avoided. We describe the processing flow of the ring architecture and its problem in the following subsections.

A. Ring Architecture

In the ring architecture, all nodes can receive query requests from any client because all of them can be a coordinator forwarding the queries to the destination nodes. Distributing query requests can average the load of query processing among the nodes. In this architecture, clients need not care about the location of data.

B. The Performance Degradation by Query Forwarding

Figure 1 shows the processing flow of the query request. The destination node is determined by an algorithm (e.g. consistent hashing). The number of query forwarding between the database nodes increases as the ring size grows. If clients directly access the nodes storing data, only two packet transfers are required. Otherwise, the additional communications of the query and result are required between the coordinator node and the data node. Therefore, the forwarding doubles the communication overhead.

III. Related Work

A. Consistent Hashing

Many of D-KVS, such as Amazon Dynamo, Cassandra, memcached [17] and Redis [18], use consistent hashing [19] to distribute data. Amazon Dynamo and Cassandra compute the destination of the query on the server. All of their nodes accept the query, and forward the query to the destination node. Memcached computes the destination node on the client. This method does not need forwarding among the cluster, but additional communications are required to inform the client of the server configuration. Redis can compute the destination both on the client and on the server. In our method, the proxy computes the destination and forwards the queries.

B. Faster Packet Processing

ScyllaDB [9] is a Cassandra-compatible D-KVS and internally uses a fast user-space network stack based on Seastar [10] and DPDK. Evaluations using YCSB [20] benchmark showed that ScyllaDB achieved 10 times higher throughput than that of original Cassandra implementation. In addition, memcached with Seastar has achieved 3.2 times higher throughput than the original memcached using the network stack in the linux kernel [21]. However, existing study [12] has reported that using a fast packet I/O is not enough to achieve the wire-rate of the network with large amount of short packets. Therefore, further improvement of packet processing efficiency is required.

C. Aggregation Method

Luo Mai et al. proposed NetAgg [13] to decrease the traffic amount by pre-aggregating data. Data center applications, such as batch processing or search engine, use a partition/aggregation pattern. In this pattern, tasks are first partitioned across servers that locally process data, and then those partial results are aggregated. In this aggregation step, large amount of traffic gathers to the edge servers and degrades the throughput. NetAgg consists of Agg boxes, middlebox-like designed servers, and connects them to each network switch. Agg boxes perform the aggregation tasks instead of the edge servers and gradually aggregate the data through the network path. NetAgg improved throughput by 9.3 times with Apache Solr query and 5.2 times with Hadoop job. This method can be applied only to the applications using the partition/aggregation pattern. Therefore, this method cannot be applied to D-KVS.

PA-Flow [14] is a packet aggregation method used on Network Functions Virtualization (NFV) environment. The packets produced by end servers get centered on upstream network function, which degrade throughput. Their method runs at the virtual network I/O layer like DPDK and aggregates the same destined packets to decrease the number of packets. Their method made efficient upstream network and improved throughput by 1.7 times. Their method showed that combining DPDK and aggregation method enables to improve throughput.

Our method introduces the PA-Flow-based packet aggregation to D-KVS. The proxy aggregates the same destined queries into single packet and forwards the packet.

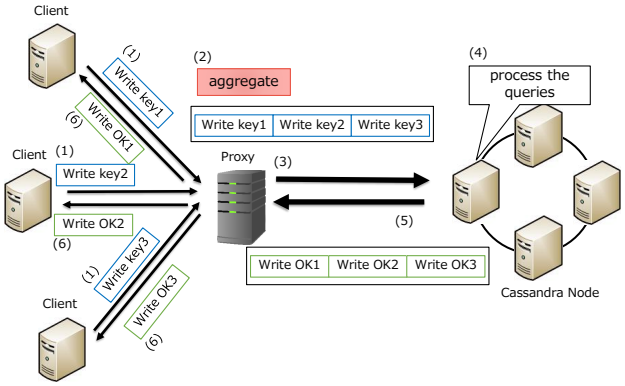

Fig. 2. System architecture

IV. Proposed Work

In this section, we describe the system architecture of our approach implemented to Cassandra.

A. Purpose

The purpose of our proposed method is to improve throughput of D-KVS by aggregating the queries in the high-performance proxy. The centralized architecture is achieved by improving throughput.

B. System Architecture

Figure 2 shows the system architecture with proxy. Three clients communicate with four Cassandra nodes via our proxy. The proxy opens a connection to each Cassandra node on startup.

1) Clients send the queries to the proxy.
2) The proxy finds out the destination nodes and aggregates the queries.
3) The proxy forwards the aggregated query packets to the destination nodes.
4) The destination nodes extract the embedded queries and process each of them.
5) The destination nodes aggregate the results and respond the aggregated result to the proxy.
6) The proxy extracts the results and sends each result to the clients.

C. Components of Our D-KVS Architecture

1) Client: The clients only communicate with our proxy. The clients do not need the state of the cluster because the proxy routes the query. No change is required for the clients.

2) Cassandra node: Cassandra nodes receive the aggregated query and extract embedded queries. The node processes the queries sequentially and aggregate the result of the queries in the same sequence. The node responds the result of the aggregated query to the proxy. We add the extraction of queries and the aggregation of queries to Cassandra.

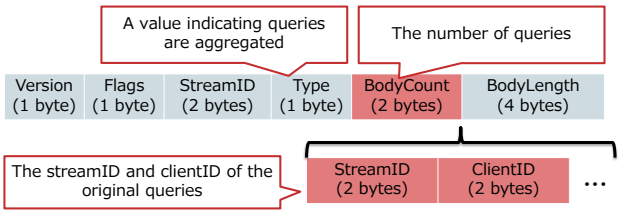

Fig. 3. Modified Cassandra protocol

3) Proxy: The proxy receives the queries from all clients and aggregate the queries. The proxy aggregates queries which have the same destination and directly sends to the destination to eliminate the forwarding procedure. The queries arriving in specified period are aggregated to limit the latency.

The proxy should use multiple CPU cores to separate its tasks to multiple CPU-core-dominated threads, such as Rx threads and Tx threads of clients, aggregation threads and disaggregation threads.

D. Cassandra Protocol

Cassandra protocol is an application layer protocol used by clients and Cassandra nodes. This protocol is used for the requests of queries and the responses of results.

The query request and its response must have the same stream ID. In our method, this header is used in communication between the client and the proxy not to change the client. Cassandra nodes should distinguish whether the received packet contains multiple queries or not. In addition, the proxy must distinguish whose results are aggregated. Therefore, we extend the Cassandra protocol.

Figure 3 shows our protocol header. Our determined custom value (0x64) is specified in 'type' field to indicate that the queries are aggregated. We add three fields ('BodyCount, 'StreamID' and 'ClientID'). The BodyCount field is the number of embedded queries used by Cassandra nodes. The StreamID and ClientID fields are added for each query to remember the requesting clients and the original StreamID respectively. The ClientID is assigned by the proxy. This header is used in communication between the proxy and Cassandra nodes.

V. Implementation

A. Cassandra Node

We describe the details of modifications to Cassandra for our method. Figure 5 shows the processing added to Cassandra. Cassandra nodes disaggregate the query using the BodyCount field and execute queries. Cassandra nodes aggregate the results of the queries and send to the proxy. We explains the modified classes in the following.

1) Frame class: Frame class processes byte sequence of the protocol header. The original Cassandra has fixed size of header. In our method, we use variable length of header in the case of aggregated query. If the type field indicates that the queries are aggregated, Frame class handles the BodyCount,

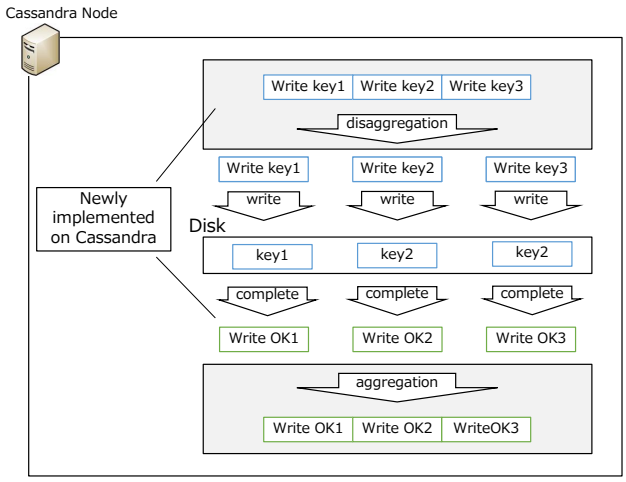

Fig. 5. Processing of Cassandra node

StreamID and ClientID field. The StreamID and ClientID fields are used with the response.

2) Message class: In the Message class, MultiMap is used storing the pair of StreamID and ClientID. The body of queries are parsed with the number of embedded queries.

B. Proxy

The internal implementation of the proxy is shown in Fig. 4. DPDK and DPDK-ANS [22], a library of TCP/IP protocol stack are used for fast packet processing. We implemented the following logic in C++.

1) Startup: The proxy opens the connections for each Cassandra node and gets ring information. Aggregation thread and disaggregation thread are created in the proxy for each node.

2) Receiving query from client: The proxy creates two threads (Rx thread and Tx thread) for each client. When the Rx thread receives the query, the Rx thread parses the query and hashes the key and gets the destination. Rx thread enqueues the query to the destination's Request Queue with atomic operation (Blue line in Fig.4). Request Queue manages the structure containing clientID, streamID and query. Aggregation thread waits for the specified time and dequeues the queries with atomic operation (Green line in Fig. 4). Finally, aggregation thread aggregates the queries based on the protocol described above and sends to the destination node.

3) Sending result to client: On receiving the result from Cassandra node, disaggregation thread disaggregates the results and enqueue to client's Response Queue. Tx thread dequeues the result and sends to client.

VI. Evaluation

In this section, we performed the following evaluation to confirm the effectiveness of query aggregation and routing with our proxy. As a client, we made a program asynchronously sending queries. YCSB is implemented to send the next query after receiving the result of the sent query.

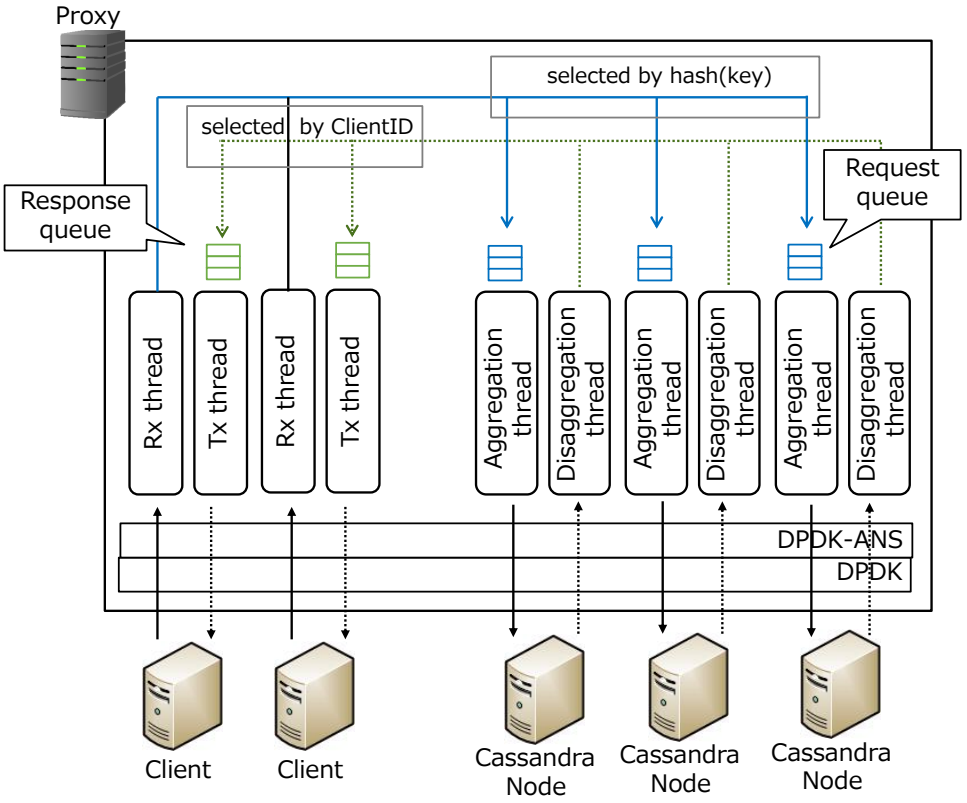

Fig. 4. The internal implementation of the proxy

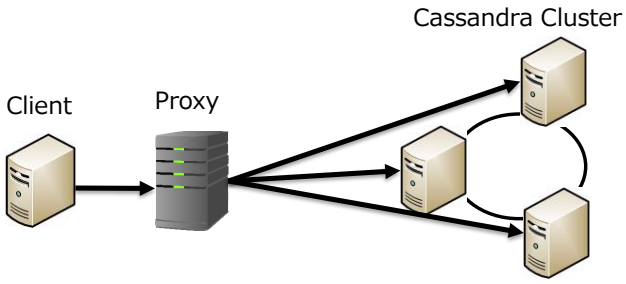

Fig. 6. Evaluation environment: one client and three Cassandra nodes

TABLE I
SPECIFICATION OF HARDWARE OF PROXY

OS	Ubuntu 16.04
CPU	Intel Core i9-7900K / 3.3 GHz (10 cores)
Memory	64 GB
Storage	1 TB SATA HDD
Network	Intel X540-T2 (10 GbE, dual port)

TABLE II
SPECIFICATION OF HARDWARE OF CLIENT AND CASSANDRA NODE

OS	Ubuntu 16.04
CPU	Intel Core i5-4460 / 3.2 GHz (4 cores)
Memory	16 GB
Storage	120 GB SATA SSD
Network	Intel X540-T2 (10 GbE, dual port)
Cassandra	version 2.2.2

To simplify the implementation of proxy, we can assign only one client for one physical machine. In that case, we cannot aggregate more queries than the number of client. Therefore, we made a program that asynchronously sends a query using Cassandra Driver and measured the throughput at the client. We also investigated the distribution of the number of aggregated queries.

A. Evaluation Environment

The evaluation environment is as shown in Figure 6. We assigned different network segments to clients and Cassandra cluster. The proxy uses a dual port Ethernet, one port belongs to the network of the client and another belongs to the network of Cassandra cluster. The performance of the computer used in the experiment is shown in Table I and II. As a client, we use a program that sends queries asynchronously using Java Driver for Apache Cassandra [23]. In the workload, we executed 100,000 INSERT operations with 20 bytes of key-value items. For comparison, we used two types of conventional client with Cassandra Driver. One is a TokenAware client which calculates the destination and sends the node directly. The TokenAware client does not cause the forwarding. Another is a RoundRobin client which sends the query to one of the three nodes in turn. The RoundRobin client can cause the forwarding.

B. Evaluation Results

1) Throughput evaluation: Figure 7 shows the throughput of the conventional method and the proposed method. The vertical axis shows the throughput. The green bars show the conventional methods TokenAware and RoundRobin. The orange bars show the proposed method with proxy. We specified the waiting time from 1 μs and 30 μs. When the waiting time is set to 20 μs, our method achieved 19% higher throughput than RoundRobin method. From the result, it is considered that query aggregation method improves throughput.

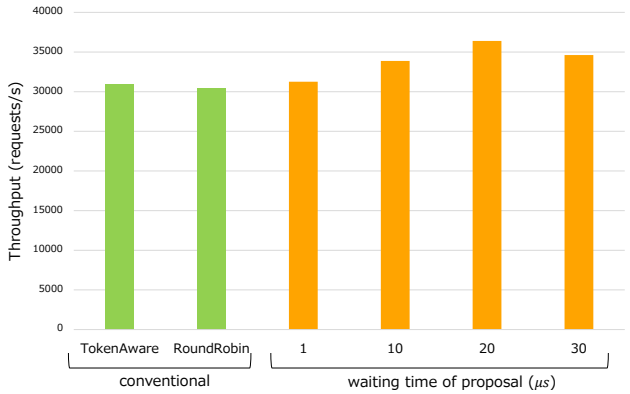

Fig. 7. Throughput of conventional and proposal with some of waiting time

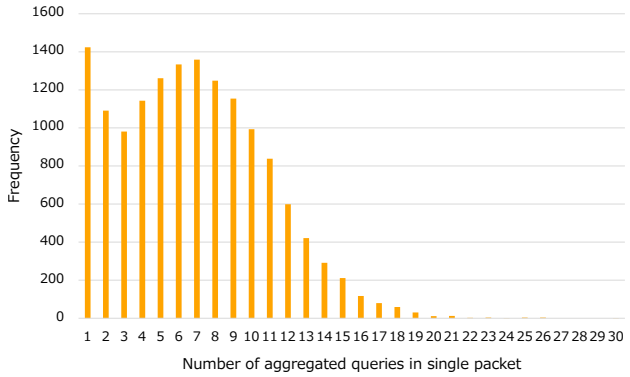

Fig. 8. The distribution of the number of aggregated queries with 10 μs waiting time

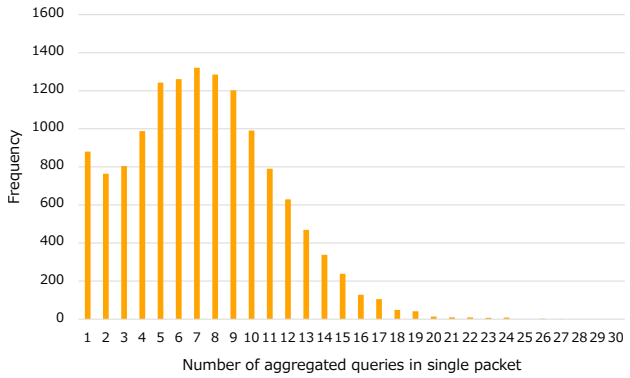

Fig. 9. The distribution of the number of aggregated queries with 20 μs waiting time

2) Evaluating the number of aggregated queries: Figure 8 and 9 show the distribution of the number of aggregated queries in a proxy with a wait time of 10 μs and 20 μs re-

spectively. The horizontal axis shows the aggregation number. The vertical axis shows the number of aggregated queries. When we prolonged the waiting time from 10 μs to 20 μs, more queries were aggregated at a time. The increasing of aggregated queries decreases the processing of network, which improved throughput.

VII. Conclusion and Future Work

In this paper, we proposed the proxy-based query aggregation method for distributed key-value stores. The proxy-based query aggregation results in reduction of traffic amount and improvement of D-KVS performance. Our evaluation result showed that the query aggregation method achieved 19% higher throughput than that of the original non-proxy Cassandra.

As future work, we extend our centralized architecture to achieve a lock mechanism. A strong consistency is required among database nodes to implement the lock mechanism on D-KVS. Ensuring this consistency incurs a communication overhead. Implementing a lock mechanism on the proxy enables mutual exclusion and transaction mechanisms without the communication overhead.

Acknowledgments

This work was supported in part by MEXT KAKENHI Grant Number 18K11324.

References

[1] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: current state and future opportunities," *Proc. the 14th International Conference on Extending Database Technology*, 2011, pp. 530–533.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *Proc. of the ACM SIGOPS Operating Systems Review*, vol. 41, 2007, pp. 205–220.

[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, 2013, pp. 385–398.

[5] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," *Proc. the USENIX conference on Operating Systems Design and Implementation*, vol. 12, 2012, pp. 135–148.

[6] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation*, vol. 14, 2014, pp. 489–502.

[7] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2014.

[8] "DPDK (Data Plane Development Kit)," [Mar. 20, 2018]. [Online]. Available: https://dpdk.org

[9] Scylla Team. (2015) ScyllaDB. [Mar. 20, 2018]. [Online]. Available: http://http://www.scylladb.com/

[10] Cloudius Systems, "Seastar," 2014, [Mar. 20, 2018]. [Online]. Available: http://www.seastar-project.org/

[11] YCSB Cluster Benchmark - Scylla vs Apache Cassandra. [Mar. 29, 2018]. [Online]. Available: https://www.scylladb.com/product/benchmarks/ycsb-cluster-benchmark/

[12] R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "Evaluation of Forwarding Efficiency in NFV-Nodes Toward Predictable Service Chain Performance," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 920–933, 2017.

[13] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "NetAgg: Using middleboxes for application-specific on-path aggregation in data centres," *Proc. 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 249–262.

[14] Y. Taguchi, R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "PA-Flow: Gradual Packet Aggregation at Virtual Network I/O for Efficient Service Chaining," *Proc. 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, China, Dec 2017, pp. 335–340.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *Proc. the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, 2003, pp. 29–43.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[17] Dormando. memcached - a distributed memory object caching system. [Mar. 20, 2018]. [Online]. Available: https://memcached.org/

[18] RedisLabs. Redis. [Mar. 20, 2018]. [Online]. Available: https://redis.io/

[19] 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," *Proc. Twenty-ninth Annual ACM Symposium on Theory of Computing*, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[20] A. S. Brian F. Cooper, R. R. Erwin Tam, and R. Sears, "Benchmarking cloud serving systems with ycsb," 2010, http://github.com/brianfrankcooper/YCSB.

[21] "Seastar memcached," [Mar. 20, 2018]. [Online]. Available: http://www.seastar-project.org/memcached/

[22] ANS team, "ANS(Accelerated Network Stack) on DPDK, DPDK native TCP/IP stack," 2017, [Mar. 29, 2018]. [Online]. Available: https://github.com/ansyun/dpdk-ans

[23] Datastax, "GitHub - datastax/java-driver: DataStax Java Driver for Apache Cassandra," mar. 30, 2018. [Online]. Available: https://github.com/datastax/java-driver