Structured Allocation-Based Consistent Hashing With Improved Balancing for Cloud Infrastructure

Yuichi Nakatani

Abstract—Consistent hashing has played an indispensable role in cloud infrastructure, although its load balancing performance is not necessarily perfect. Consistent hashing has long remained the most widely used method despite many methods being proposed to improve load balancing because these methods trade off load balancing against consistency, memory usage, lookup performance, and/or fault-tolerance. This article presents Structured Allocation-based Consistent Hashing (SACH), a cloud-optimized consistent hashing algorithm that overcomes the trade-offs by taking advantage of the characteristics of cloud environments: scaling management and auto-healing. Since scaling can be distinguished from failures, SACH applies two different algorithms to update hashing functions: a fast-update algorithm for unmanaged backend failures to satisfy fault-tolerance with quick response and a slow-update algorithm for managed scaling. Hashing functions are initialized or slow-updated considering the characteristics of the fast-update algorithm to satisfy load balancing and the other properties as far as the number of failed backends is kept small by auto-healing. The experimental results show that SACH outperforms existing algorithms in each aspect. SACH will improve the load balancing of cloud infrastructure components, where the trade-offs have prevented the renewal of hashing functions.

Index Terms—Consistent hashing, load balancing, network load balancer, distributed key-value store, cloud infrastructure

1 Introduction

Consistent hashing (CH) [11], [23] has become an essential building block in many cloud infrastructure components such as network load balancers and distributed databases [6], [7]. CH evenly maps the keys of incoming packets or requests with *consistency* into backends, which compose distributed systems. Even if the set of backends changes, CH avoids remapping of keys unnecessarily; a key mapped once to a backend is consistently assigned to the same backend in most cases. This consistency is a crucial property that enables network load balancers to maintain TCP connections and that prevents distributed database systems from disruption by huge data-replacement.

As CH has been commonly used, the demand for more even load balancing has been increasing to suppress capital expenditure due to resource overprovisioning. CH leaves room for improvement of its load balancing, though it robustly manages a certain level of load balancing even in P2P-like severe environments. In P2P-like environments, the joining and leaving of backends are entirely uncontrollable. Therefore, it is valuable to reconsider cloud-optimized hashing with consistency, which achieves more even load balancing at the expense of some of the robustness of CH.

• *The author is with the NTT Network Service Systems Laboratories, Tokyo 108-8019, Japan. E-mail: yuichi.nakatani.rd@hco.ntt.co.jp.*

Manuscript received 14 May 2020; revised 5 Feb. 2021; accepted 8 Feb. 2021. Date of publication 12 Feb. 2021; date of current version 25 Mar. 2021.
Recommended for acceptance by A. Sussman.
Digital Object Identifier no. 10.1109/TPDS.2021.3058963

Many methods have been proposed to improve load balancing of hashing with consistency [3], [7], [15], [16], [17], [20], [22], [24]. However, they have all struggled to overcome a trade-off among the following five properties, so most cloud infrastructure components still use original CH.

- *Consistency*: when the set of backends changes, the key mapping does not exchange among the backends that have not been added or removed.
- *Uniform load balancing*: load distribution among backends is as uniform as possible.
- *Fast lookup*: time to map a given key is as short as possible.
- *Low memory usage*: memory usage is less than the amount of memory in a general server.
- *Fault-tolerance*: time to update hashing functions in response to backend failures is as short as possible.

CH achieves fast lookup and fault-tolerance but has a trade-off between load balancing and memory usage. Memory usage may be impractical if uniform load balancing is strongly required as in cloud infrastructure components [26]. Highest random weight hashing (HRW) [24] achieves equal load balancing with little memory usage. However, it may not be appropriate for applications that need a high lookup rate, because its lookup time increases as backends are inserted. Maglev hashing [7] achieves almost even load balancing and a high lookup rate. However, it takes so long to update a lookup table that it might be difficult to use in the distributed systems where backends can often fail.

This paper proposes Structured Allocation-based Consistent Hashing (SACH), which simultaneously satisfies all the properties in cloud infrastructure-like environments, where backend provisioning is under control and the percentage of simultaneously failed backends is small. In current cloud infrastructure environments, backends (machines) are virtualized and are controlled by so-called orchestrators [1], [2]. As the orchestrators comprehensively manage resource provisioning and automatically resume failed backends, hashing algorithms can be told how the set of backend will change in advance and the number of simultaneously failed backends can be kept down [18], [19], [25].

SACH uses two original approaches. One is providing distinct algorithms for updating the hashing function, depending on whether the update is caused by backend failures/recoveries or by designed additions/removals. This distinction enables the algorithms for designed updates to take enough time to consider various requirements while enabling backend failures to be handled quickly. Existing algorithms do not distinguish occasions of function updates. The other one is structuring an allocation of the hash space so as to increase the load balancing performance under zero or few backend failures. CH achieves a certain level of load balancing regardless of the number of backend failures, as a result of its random allocating algorithm. In other words, CH equally treats the cases of an arbitrary number of backend failures. On the other hand, SACH emphasizes the case of zero or few backend failures, because the number of simultaneously failed backends can be assumed to be small.

After describing related works (Section 2), the paper makes its main contributions: (I) proposing a novel CH architecture where function-update algorithms differ between backend failures/recoveries and designed additions/removals (Section 3), (II) modeling and formulating a hash space allocation problem as a backend sequencing problem in discrete mathematics (Section 4), (III) proposing novel algorithms to solve the formulated problem (Section 5), and (IV) evaluating the proposed algorithm in comparison with existing ones with experimental results (Section 6). Section 7 discusses the results and limitation of SACH, and Section 8 concludes the paper.

2 Related Work

CH [11], [12], [23] assigns multiple virtual nodes on the cyclic hash space for each node (backend), and each backend is responsible for all the segments that end with its virtual nodes. This segmentation algorithm leads to consistency. Insertions/removals of backends cause only insertions/removals of their virtual nodes, so the segments not related to the moved backends are not affected at all. CH also achieves fault-tolerance and high lookup performance because the update process is lightweight, and there is only one hash calculation. However, CH has a trade-off between load balancing and memory usage, i.e., keeping the load ratio deviation under $1 + \epsilon$ requires $\mathcal{O}(1/\epsilon^2)$ numbers of virtual nodes [26], so strict load balancing as in cloud infrastructure leads to impractical memory usage.

Some papers [3], [17], [22] proposed hashing algorithms derived from CH to improve load balancing. However, although they overcame the trade-off of original CH between load balancing and memory usage, these algorithms still have trade-offs between load balancing and other properties such as the lookup rate or scalability. Multi-probe consistent hashing [3] improved load balancing without increasing the number of virtual nodes. When looking up a backend for a given key, it calculated multiple hash values of the key and mapped the key into the backend that was closest to one of the hash values. Load balancing can be improved by increasing the number of hash calculations. However, this hashing is difficult to deploy in applications requiring a high lookup rate such as cloud infrastructure applications, because the number of the hash calculations must be $\mathcal{O}(1/\epsilon)$ to keep the load ratio deviation less than $1 + \epsilon$. Another paper [17] presented a scheme that directly bounds the maximum load. When mapping a new key, it searches the hash space clockwise for a backend whose load has not reached the maximum, while skipping the backends with maximum load. Although this scheme prevents overloaded backends, it reduces its lookup rate because of the increased number of the backend skips for small ϵ. Therefore, this scheme alone is not ideal for improving the load balancing of CH, although it is beneficial to prevent overloaded backends in combination with other hashing algorithms including SACH.

Perfect consistent hashing [22] uses an approach similar to SACH. It analyzes how bucket allocation changes for insertions/removals of backends and shows a formulated problem and its solution. It achieves perfect load balancing even when backends are arbitrarily removed (this is the special case of SACH with a parameter $d_f = n_{max} - 2$ as described later). However, the number of buckets, i.e., memory usage, is too high as $n_{max}!$ to apply perfect consistent hashing to real problems. This is because clusters often consist of 100 or more backends [6], [7], while an exascale storage area is required even with $n_{max} = 20$. Also, it analyzes only the strict condition where perfect load balancing is achieved for an arbitrary number of backend failures, but does not analyze the condition, the key of SACH, where the load is evenly balanced only for a small number of backend failures.

A representative hashing algorithm not derived from CH is HRW [24], which utilizes per-backend hash functions instead of allocating a hash space to backends. HRW maps a given key to the backend whose hash value is the maximum among those of the backends. This hashing achieves even load balancing when the number of keys is large, and its memory usage is very low. However, HRW needs hash calculations equal in number to the backends for one lookup, so it is not appropriate for applications requiring a high lookup rate such as cloud infrastructure applications.

Jump hashing [15] and AnchorHash [16] are algorithms that combine hash functions in multiple layers and reduce the number of hash calculations compared to HRW. Jump hashing [15] utilizes per-backend hash functions as well as HRW but reduces the number of per-lookup hash calculations to $\mathcal{O}(\log(n))$ on average by arranging the order of hash value evaluation. However, the arrangement restricts the order of backend removals, so Jump hashing is not appropriate in environments like cloud infrastructure where the backend removal order is not controllable because of backend failures. With AnchorHash [16], the number of hash calculations can be reduced to $1 + \ln(n_{max}/n)$ on average and $n_{max} - n$ at the maximum. Here, n is the number of active backends, and n_{max} is set to exceed the maximum number of backends, considering future needs. Although AnchorHash is approximately ideal in the sense that it achieves the five properties as long as n_{max} is close to n, it is not necessarily ideal for cloud infrastructure applications. Since cloud infrastructure applications start with a small number of backends and scale fast, n_{max} must be much larger than the initial n, so the lookup rate of AnchorHash unavoidably deteriorates.

Other noteworthy algorithms are Maglev [7] and Beamer [20]. Maglev [7] allocates or re-allocates buckets of a lookup table evenly to the backends with a distributed autonomous process and high computing efficiency. It first generates a permutation table that holds a distinct permutation for every backend, and backends take turns at accepting an empty bucket one by one in ascending order of its permutation. Maglev hashing can be interpreted as a very efficient HRW for the buckets without hash calculations, because its permutations are generated through a special series of linear congruential generators. However, as its re-allocation process is exploratory with the permutation table, its re-allocation process is still time-consuming in the case of backend failures (see 6.4). A non-exploratory re-allocation process is necessary for fault-tolerance. Also, Maglev still has a slight trade-off between consistency and load balancing (see Section 6.2). Beamer [20] has demonstrated that distributed systems in cloud environments do not necessarily need to be autonomous and that muxes may coordinate in bucket allocation. Therefore, in Beamer, the allocation of the hash space is managed centrally by a controller, and muxes only retrieved the allocation from the controller. The controller divides the hash space into a small number of contiguous buckets, so that a high lookup rate, efficient memory usage, and even load balancing can be achieved. However, since muxes ask the controller for an updated allocation even in the case of backend failures, the fault-tolerance property is insufficient. In cloud infrastructure environments, a central machine that manages the bucket allocation can be assigned, but a fast and autonomous update mechanism appears to be necessary for fault-tolerance.

Compared with the related works above, SACH takes advantage of its characteristics to meet the five requirements better. First, unlike algorithms that use multiple hash functions such as HRW and AnchorHash, SACH allocates one hash space to the backends like CH and Maglev so that it can quickly complete the lookup with only one hash calculation. On the other hand, in related works allocating hash space to the backends such as CH and Maglev, the load balancing performance was traded off against memory usage or update time. Especially, Maglev meets the memory usage requirement but takes much update time to recalculate the allocation that gives good load balancing. On the other hand, SACH uses two different algorithms, fast-update and slow-update. While SACH updates the allocation with the CH-like fast-update algorithm against backend failures/recoveries, it structured the allocation with the slow-update algorithm for other cases. The slow-update algorithm considers the characteristics of the fast-update so that the load balancing is maintained with it. This approach can eliminate practical trade-offs.

Finally, this paragraph explains the relationship with load balancing algorithms. SACH is classified as a static algorithm, which does not utilize current states of backends [13]. Static algorithms represented by the round-robin algorithm are widely used because they are simple and easily achieve good performance [27]. However, general static algorithms do not scale well because they require shared session information between load balancers when they are applied to distributed network load balancers with session-affinity. On the other hand, hashing algorithms with consistency enable scalable distributed architecture by reducing the need for session information sharing. Especially, SACH is a hashing algorithm with consistency that simultaneously satisfies the five requirements.

3 Overview

SACH is similar to CH in many respects. The mapping of a key is indirect through a cyclic hash space. The hash space is divided into buckets, and each bucket is allocated to a backend. Buckets greatly outnumber backends. When looking up the backend corresponding to a given key, SACH hashes the key into one of the buckets and maps it to the backend related to the bucket. In particular, as SACH divides the hash space into equal-sized buckets, SACH uses a hash table equivalent to the hash space allocation. Also, one of SACH's update algorithms follows the update algorithm of CH.

To update an allocation of the hash space, SACH uses different approaches depending on whether backend failures/recoveries cause the updating or not, whereas CH performs the same processes. All CH does is insert virtual nodes for backend additions randomly and delete the virtual nodes for backend removals. These processes are so fast that CH can be fault-tolerant, although its load balancing is not necessarily uniform. On the other hand, SACH uses different updating algorithms between designed additions/removals and failures/recoveries. Cloud infrastructure usually controls the designed backend additions/removals, so that we can spend enough time updating allocations for the designed additions/removals. Therefore, we should prepare a fast updating algorithm for backend failures/recoveries and develop another updating algorithm for other cases considering characteristics of the fast-update algorithm. This approach may enable to design hashing algorithms that achieve desirable properties without sacrificing fault-tolerance.

3.1 Architecture

SACH includes four algorithms. *Fast-update* quickly updates an allocation of the hash space through CH-like processes. *Initial allocating* and *slow-update* respectively generate and update an allocation to be well-structured, considering characteristics of fast-update. A well-structured allocation achieves uniform load balancing through zero or few fast-updates. Lastly, *lookup* looks up a backend that corresponds to a given key with a well-structured allocation or its fast-updated one. These details are described later.

Next, the architecture of SACH in Fig. 1 is explained. SACH consists of two functions. One is a *controller*, which manages well-structured allocations with initial allocating and slow-update. The other is a balancer (*mux*), which is equipped with fast-update and lookup and maps the keys

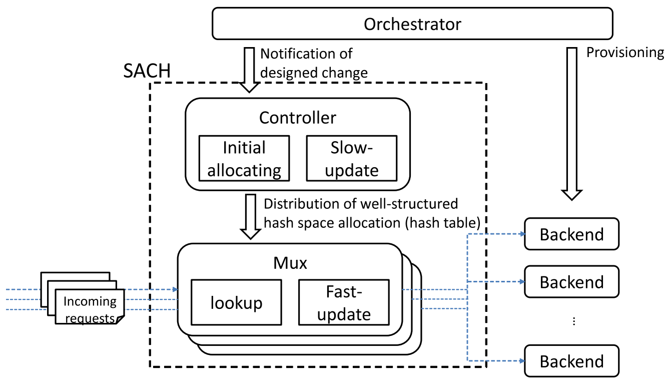

Fig. 1. SACH architecture.

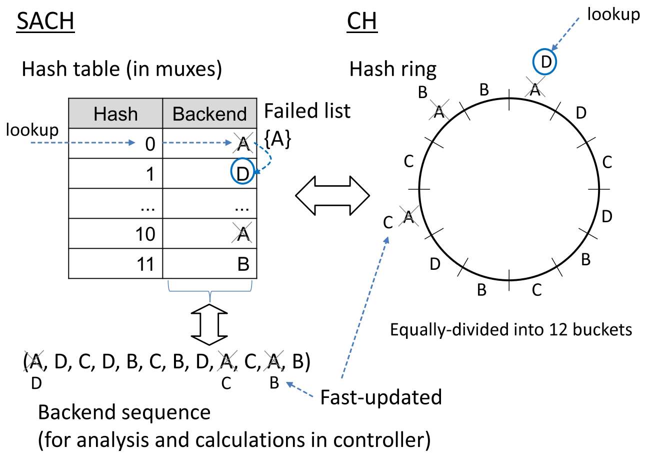

Fig. 2. Relations between a backend sequence, a hash table of SACH, and hash space allocation of CH (with fast-updated ones).

of incoming requests into backends by using well-structured allocations distributed from the controller.

The controller is located around so-called orchestrators, such as Kubernetes [1] and OpenStack Heat [2], which are responsible for provisioning resources. When creating a distributed system and changing the number of its backends, the controller is notified of the events and initializes or slow-updates a well-structured allocation, regardless of backend failures. Then the controller distributes the obtained well-structured allocation to the muxes. After the distribution, the controller does nothing more unless the configuration of the distributed systems is changed.

Once receiving a well-structured allocation, the muxes can autonomously operate. Even when some backends fail, each mux can continue to operate without communicating with the controller or other muxes. All that muxes have to do is to fast-update the allocation as soon as they detect backend failures. In other words, the distributed systems can satisfy the fault-tolerance property.

3.2 Fast-Update and Lookup

The rest of this section presents the details of the CH-like algorithms, fast-update and lookup. The following sections describe what well-structured allocations are and how to initialize and slow-update them.

Fast-update closely resembles CH's updating algorithm but somewhat differs in response to backend recoveries. A recovered backend is reassigned to the buckets it was in charge of before its failure. In the case of backend failures, a bucket that was mapped to a failed backend is remapped to the backend whose bucket is next to the remapped bucket. When implementing this fast-update, mux updates a list of failed backends, but not the hash table itself, as shown on the upper left in Fig. 2. A failed backend is added to the list and removed from the list when it recovers. When looking up the backend corresponding to a given hash value, SACH searches for a backend not in the list in order from its corresponding bucket. With this implementation, SACH does not need to store the updated hash table for recovery, while the result of the lookup is the same as that obtained by updating the table itself. In cloud infrastructure environments, this process hardly affects the lookup performance because the number of failed backends is assumed to be small.

In looking up the backend for a given hash value, SACH with a hash table does not need time-consuming processes such as binary search in CH. However, the hash table size l is determined independently of the range of hash functions. Therefore, the lookup algorithm scales hash values to $0, \dots, l-1$ to select the corresponding bucket. If the selected bucket is assigned to a failed backend, SACH searches the following buckets in order until it finds the first active backend.

4 Problem Formulation

As described in the previous section, SACH architecture is composed of the controller that maintains a *well-structured allocation* and muxes that autonomously map incoming requests into backends using the well-structured allocation. A well-structured allocation leads to proper load balancing under zero or few backend failures with fast-update. The goal of this section and the next is to present algorithms that create or update a well-structured allocation. This section clarifies what a well-structured allocation is: a sufficient condition that the hash space allocation should satisfy to achieve proper load balancing with some fast-updates. The next section provides algorithms to obtain allocations to satisfy the condition.

To clarify and formulate the condition where proper load balancing is achieved under zero or few backend failures with fast-update, we introduce a positive parameter d_f, which indicates how many backend failures we consider. Then we discuss the condition where load deviation is kept small against $n_f (0 \le n_f \le d_f)$ fast-updates. Here, n_f corresponds to the number of failed backends.

In the rest of this paper, we model a hash table as a backend sequence to be handled mathematically. Let the number of backends or buckets be n or l, respectively, and denote the set of backends as $B := \{b_1, \dots, b_n\}$. Then a hash table can be transformed into a cyclic backend sequence $s \in S := \{(s_1, \dots, s_l), s_i \in B\}$ without loss of generality. For example, the hash table in the upper left of Fig. 2 and the lower left's backend sequence are interconvertible.

The fast-update of a backend sequence model is defined as follows. As in the previous section, muxes do not update the hash table itself, but the lookup result is the same as the one obtained with the updated hash table. Therefore, the fast-update of a backend sequence is defined to directly update the sequence to analyze load balancing performance only with backend sequences. Let B_f be the set of the failed backends and $B_a := B \backslash B_f$ be the set of the active backends. Then fast-update replaces every failed backend in a given backend sequence with the first active backend in the following sequence. That is, $s_i \in B_f$ is replaced with $s_j \in B_a$ s.t. $\forall k \in [i, j), s_k \in B_f$ and $s_j \in B_a$. As in the lower left of Fig. 2, A's buckets are updated to the next buckets' backends when backend A fails.

Furthermore, we deal with the following subset $S_{P_{d_f+1}}$ instead of S, so that fast-update becomes well-defined and load shifts by fast-update become simple when $n_f \leq d_f$. Here, P_i denotes the set of the i-length permutations of B.

$$S_{P_{d_f+1}} := \{(s_1, \ldots, s_l) \in S \,|\, \forall i, (s_i, \ldots, s_{i+d_f}) \in P_{d_f+1}\}.$$

For any $s \in S_{P_{d_f+1}}$, its arbitrary subsequence of length $d_f + 1$ is composed of distinct backends. Therefore, when discussing $n_f \leq d_f$ backend failures, the maximum length of the subsequences composed of only failed backends is less than or equal to n_f. In other words, $s_i \in B_f$ is replaced with $s_j \in B_a, j \leq n_f + i$ in fast-update on $S_{P_{d_f+1}}$.

The bottom left of Fig. 2 shows an example of S_{P_2}. All its subsequences of length 2 such as (A, D) and (D, C) belong to P_2. If backend A fails, backends B, C, and D receive one bucket each in accordance with the subsequences (A, B), (A, C), and (A, D). That is, the backend sequence (A, D, C, D, B, C, B, D, A, C, A, B) is fast-updated to ($\underline{\text{D}}$, D, C, D, B, C, B, D, $\underline{\text{C}}$, C, $\underline{\text{B}}$, B) when backend A fails. After being fast-updated, backend sequences are not in $S_{P_{d_f+1}}$, but in S.

For $S_{P_{d_f+1}}$, this paper introduces the following metrics, which indicate a deviation of appearance frequency of permutations in P_{d_f+1}. The metrics have a monotonicity property shown in lemma 2. Its proof is given in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3058963.

Definition 1 (Metrics for backend sequences). *For positive $i \leq d_f + 1, s \in S_{P_{d_f+1}}$ and $p \in P_i$, let $num(s, p)$ be the number of p in s. Then this paper defines the following metrics $M_i^{(d_f)}(s)$ for $s \in S_{P_{d_f+1}}$.*

$$M_i^{(d_f)}(s) := \sum_{p \in P_i} \left| num(s, p) - \frac{l}{|P_i|} \right|. \tag{1}$$

Lemma 2 (Monotonicity of metrics). *The metrics $M_i^{(d_f)}(s)$, defined in definition 1, monotonically increase for i. i.e.,*

$$M_1^{(d_f)}(s) \leq M_2^{(d_f)}(s) \leq \cdots \leq M_{d_f+1}^{(d_f)}(s).$$

With the metrics, the load of the most heavily loaded backend can be bounded. Let $L_{n_f}^{Peak}$ be the ratio of the maximum load to the average one when n_f backends fail. Then the following theorem about $L_{n_f}^{Peak}$ holds. Here, ${}_nP_i$ is the number of i-permutations of n elements. The proof of this theorem is in Appendix B, available in the online supplemental material.

Theorem 3 (Peak-load boundary). *For any $n_f, 0 \leq n_f \leq d_f$, the following inequality holds.*

$$L_{n_f}^{Peak}(s) \leq 1 + \frac{n - n_f}{l} \sum_{i=0}^{n_f} {}_{n_f}P_i \cdot M_{i+1}^{(d_f)}(s) \tag{2}$$

$$\leq 1 + \frac{n - n_f}{l} \left(\sum_{i=0}^{n_f} {}_{n_f}P_i \right) M_{n_f+1}^{(d_f)}(s). \tag{3}$$

Our objective (obtaining a well-structured allocation with which load deviation is kept small against $n_f (0 \leq n_f \leq d_f)$ fast-updates) can be reduced to the following problem with this theorem. The next section presents algorithms to solve the problem.

Problem 4. *[Allocation structuring problem] Give an optimal backend sequence $s_* \in S_{P_{d_f+1}}$ that satisfies the following equation.*

$$s_* = \underset{s \in S_{P_{d_f+1}}}{\arg\min}\, M_{d_f+1}^{(d_f)}(s). \tag{4}$$

5 Allocation Algorithm

This section presents two algorithms for obtaining backend sequences that either solve or approximate Problem 4 described in the previous section. One is *backend-full sequencing*, which is an algorithm to efficiently obtain a backend sequence s that contains all n_{max} backends for a given capacity parameter n_{max}, and that satisfies $M_{d_f+1}^{(d_f)}(s) = 0$ (i.e., the obtained backend sequence is obviously a strict solution for Problem 4). The other is *slow-update*, which increases or decreases the number of backends under n_{max} for a given backend sequence (including a backend sequence generated by backend-full sequencing). The slow-update algorithm outputs a strict or approximate solution for Problem 4 by updating backend sequences while keeping down $M_{d_f+1}^{(d_f)}(s)$.

Relations between backend-full sequencing and slow-update and algorithms mentioned but not yet explained in Section 3.1 (*initial allocating* and *slow-update*) are described below with Fig. 3. Slow-update corresponds to that in this section. Initial allocating is a combination of backend-full sequencing and slow-update. When applying initial allocating, the controller combines slow-update and backend-full sequencing as shown in Fig. 3. First, the controller sets the backend capacity n_{max} as equal to or more than the initial number of the backends n and generates a backend sequence for n_{max} with backend-full sequencing. The example in Fig. 3 sets $n_{max} = 5$ and $n = 4$. Backend-full sequencing is performed for backend set {A, B, C, D, E}, composed of initial backend set {A, B, C, D} and reserved backend(s) {E}. Next, the controller updates the obtained backend sequence with slow-update to remove the reserved backend (s) in the sequence. The updated backend sequence is the initial well-structured backend sequence, which will be distributed to the muxes.

5.1 Backend-Full Sequencing

Backend-full sequencing efficiently calculates an optimal backend sequence s that satisfies $M_{d_f+1}^{(d_f)}(s) = 0$, which is an optimal solution for the problem 4. More concretely, it gives a backend sequence that includes every element in P_{d_f+1} exactly once. In the rest of this section, we refer to such backend sequences as *backend-full sequences*. Here, the length

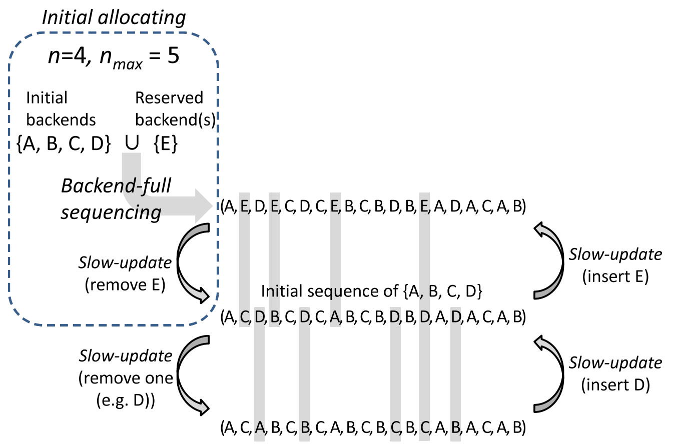
Initial allocating
n=4, n_max = 5
Initial backends
Reserved backend(s)
{A, B, C, D} ∪ {E}
Backend-full sequencing
(A, E, D, E, C, D, C, E, B, C, B, D, B, E, A, D, A, C, A, B)
Slow-update (remove E)
Slow-update (insert E)
Initial sequence of {A, B, C, D}
(A, C, D, B, C, D, C, A, B, C, B, D, B, D, A, D, A, C, A, B)
Slow-update (remove one (e.g. D))
Slow-update (insert D)
(A, C, A, B, C, B, C, A, B, C, B, C, B, C, A, B, A, C, A, B)

Fig. 3. Relation between initial allocating, backend-full sequencing, and slow-update.

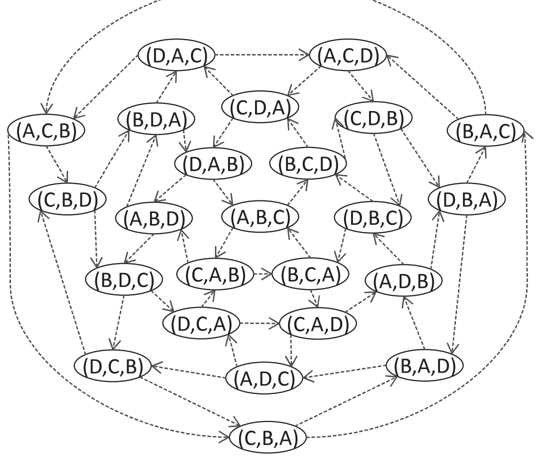

Fig. 4. Example of $G_B^{(L)}(k=3, n=4)$, $B=${A, B, C, D}.

of a backend-full sequence l equals $|P_{d_f+1}| = {}_{n_{max}}P_{d_f+1}$. Therefore, it is impractical to obtain backend-full sequences via a naive search, because the number of possible backend sequences is n to the power $|P_{d_f+1}|$. On the other hand, backend-full sequencing can be done in polynomial time.

The algorithm resembles the construction of de Bruijn sequences [5]. A de Bruijn sequence of order k on a set is a cyclic sequence where every possible length-k sequence of elements in the set appears exactly once as a subsequence. The difference between de Bruijn sequences and backend-full sequences is the difference in the set to which their subsequences belong: every possible sequence for de Bruijn sequences and every permutation for backend-full sequences. This paper modifies the construction method of de Bruijn sequences considering this difference.

Let us begin by briefly explaining how to construct de Bruijn sequences [5]. Here we denote the length of a subsequence as k and the number of elements in the set as n. First, we construct a de Bruijn graph $B(k,n)$, in which a node corresponds to one of the length-k possible sequences, and each node is linked to its adjacent nodes. The adjacency of nodes is defined as the match between the length-$(k-1)$ former subsequence of a node and the length-$(k-1)$ latter subsequence of another node. For example, in a de Bruijn graph of order 3 over {A, B, C}, the node with (A, $\underline{\text{B, C}}$) is linked to the nodes with ($\underline{\text{B, C}}$, A), ($\underline{\text{B, C}}$, B), ($\underline{\text{B, C}}$, C). Then we can construct de Bruijn sequences by lining up over $B(k,n)$ the first elements of the nodes along a Hamiltonian cycle, which is a cycle that visits every node exactly once. Although obtaining a Hamiltonian cycle of a graph G_1 is generally difficult (NP-complete), it can be reduced to a problem of obtaining an Eulerian cycle of another graph G_2 if G_1 is the line graph of Eulerian G_2 (G_2 is called an underlying graph of G_1). Since de Bruijn graphs have a good property that $B(k-1,n)$ becomes an underlying graph of $B(k,n)$ and Eulerian, we can construct a de Bruijn sequence of order k by lining up the first elements of the edge along an Eulerian cycle of $B(k-1,n)$, whose edges are related to sequences obtained by concatenating the sequences of the adjacent nodes. Now only $B(k-1,n)$, not $B(k,n)$, needs to be constructed.

Now we go back on backend-full sequencing. Steps of backend-full sequencing are the same as those of constructing de Bruijn sequences. We construct the following graph $G^{(L)}(k,n)$ instead of a de Bruijn graph $B(k,n)$, and line up the first elements of the nodes along one of its Hamiltonian cycles. The difference between $G^{(L)}(k,n)$ and $B(k,n)$ is the difference in the set of sequences related to their nodes. The nodes of $G^{(L)}(k,n)$ correspond to permutations P_k, not to all possible sequences like $B(k,n)$. For Problem 4, we set $k = d_f + 1$ and $n = n_{max}$. Fig. 4 shows an example of $G_B^{(L)}$, $B=${A, B, C, D} where $k=3$.

$$\begin{aligned}
G^{(L)}(k,n) &:= (V^{(L)}(k,n), E^{(L)}(k,n)),\\
V^{(L)}(k,n) &:= P_k\\
&= \{(b_1,\ldots,b_k),\ldots,(b_n,\ldots,b_{n-k+1})\},\\
E^{(L)}(k,n) &:= \{((v_1,\ldots,v_k),(v_2,\ldots,v_k,b))\\
&\qquad |\,(v_1,\ldots,v_k),(v_2,\ldots,v_k,b)\in V^{(L)}\}.
\end{aligned}$$

The problem here is how to obtain a Hamiltonian cycle of $G^{(L)}(k,n)$. Whereas a de Bruijn graph $B(k,n)$ is a line graph of $B(k-1,n)$, $G^{(L)}(k,n)$ is not a line graph of $G^{(L)}(k-1,n)$, so the problem remains NP-complete. However, for $k \geq 3 (d_f \geq 2)$, Jackson [10] proved that the following Eulerian $G^{(U)}(k,n)$ is an underlying graph of $G^{(L)}(k,n)$ and constructed the sequence. In addition, the process of obtaining a Hamiltonian cycle can be accelerated if $k = n-1$ [9], [21]. In fact, this $G^{(U)}(k,n)$, whose example is shown in Fig. 5, is quite similar to $G^{(L)}(k-1,n)$. Every node in $G^{(U)}(k,n)$ has one less edge than the node corresponding to the same permutation in $G^{(L)}(k-1,n)$. By comparing the two nodes connected by the removed edge, their permutations consist of the same set of backends. In other words, the concatenated backend sequence of the removed edge is not a permutation, while the other edges can be related to permutations, so one-to-one correspondence exists between the edges of $G^{(U)}(k,n)$ and the nodes of $G^{(L)}(k,n)$.

$$\begin{aligned}
G^{(U)}(k,n) &:= (V^{(U)}(k,n), E^{(U)}(k,n)),\\
V^{(U)}(k,n) &:= P_{k-1}\\
&= \{(b_1,\ldots,b_{k-1}),\ldots,(b_n,\ldots,b_{n-k+2})\},\\
E^{(U)}(k,n) &:= \{((v_1,\ldots,v_{k-1}),(v_2,\ldots,v_{k-1},b))\\
&\qquad |\,(v_1,\ldots,v_{k-1}),(v_2,\ldots,v_{k-1},b)\in V^{(U)},\\
&\qquad b \neq v_1\}.
\end{aligned}$$

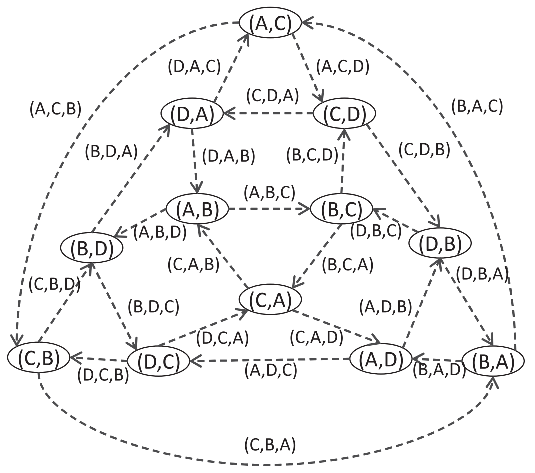

Fig. 5. Example of $G_B^{(U)}(k=3, n=4), B =$ {A, B, C, D}. This is an underlying graph of the graph shown in Fig. 4.

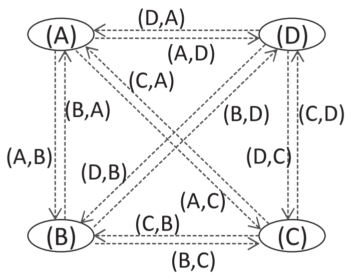

Fig. 6. Example of $G_B^{(U)}(k=2, n=4), B =$ {A, B, C, D}.

Since this underlying graph $G^{(U)}(k,n)$ is defined only if $k \geq 3 (d_f \geq 2)$, we have not obtained well-structured sequences for $k = 2 (d_f = 1)$ yet. Therefore, this paper naturally extends the definition of $G^{(U)}(k,n)$ for the case $k = 2 (d_f = 1)$ as described below. The extension only requires the common subsequence between the permutations of the connected nodes to be empty. From the conclusion, this extension for the case $k = 2 (d_f = 1)$ is significant for SACH as described later.

$$
\begin{aligned}
G^{(U)} &:= (V^{(U)}, E^{(U)}),\\
V^{(U)} &:= P_{k-1}\\
&= \{(b_1, \ldots, b_{k-1}), \ldots, (b_n, \ldots, b_{n-k+2})\},\\
E^{(U)} &:= \begin{cases} \{((v_1, \ldots, v_{k-1}), (v_2, \ldots, v_{k-1}, b)) \\ \mid (v_1, \ldots, v_{k-1}), (v_2, \ldots, v_{k-1}, b) \in V^{(U)}, \\ b \neq v_1\}, \text{if } k \geq 3, \\ \{((v_1), (b)) \mid (v_1), (b) \in P_1, b \neq v_1\}, \\ \text{if } k = 2. \end{cases}
\end{aligned}
$$

A node of $G^{(U)}(2,n)$ has a label of a permutation of length one such as (A) and (B), and the graph has an edge bidirectionally between every node-pair (an example is in Fig. 6). It can be easily proved that $G^{(U)}(2,n)$ is an underlying graph of $G^{(L)}(2,n)$ and that $G^{(U)}(2,n)$ is Eulerian. Therefore, we can also obtain well-structured sequences in the case of $k = 2 (d_f = 1)$ as well as the case of $k > 3$. First, an arbitrary edge $((v_i), (v_j)) \in G^{(U)}(2,n)$ is connected through node (v_j) to the edges $((v_j), (b)), b \neq v_j$, which are the whole set of the permutations beginning with v_j. The connections between the edges in $G^{(U)}(2,n)$ are the same as the connections between the nodes in $G^{(L)}(2,n)$ by definition. Therefore, $G^{(U)}(2,n)$ is an underlying graph of $G^{(L)}(2,n)$. Second, since every node in $G^{(U)}(2,n)$ is connected to the other nodes and the edges are bidirectional (i.e., the degree of each node is even (0)), $G^{(U)}(2,n)$ is Eulerian.

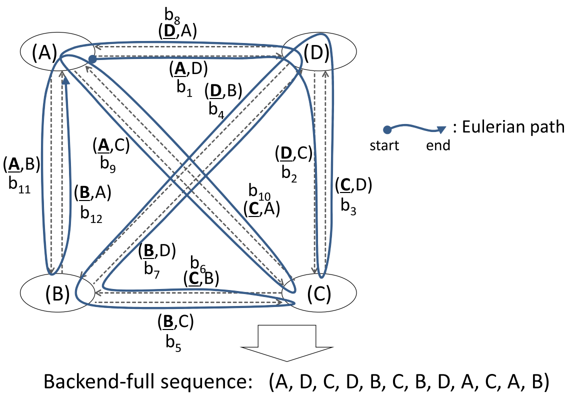

Fig. 7. Example of backend-full sequencing.

This section gave an algorithm obtaining a backend-full sequence s_* for $d_f \geq 1$, which includes every element in P_{d_f+1} exactly once. The obtained sequence s_* obviously satisfies $M_{d_f+1}^{(d_f)}(s_*) = 0$ and is one of the optimal solutions for Problem 4. This shows that even load balancing can be achieved with finite memory usage because the length l of s_* equals the number of the nodes in $G^{(L)}$ or $|P_{d_f+1}| = {}_{n_{max}}P_{d_f+1}$. The algorithm can be summarized as below.

(Backend-full sequencing algorithm)

1) Extracts an Eulerian cycle in $G^{(U)}(d_f + 1, n_{max})$.
2) Lines up the first elements of the edge permutations along the Eulerian cycle, as in Fig. 7.

More precisely, another solution of Problem 4 can be obtained with not a Hamiltonian cycle over $G^{(L)}$ (i.e., an Eulerian cycle over $G^{(U)}$) but with an Eulerian cycle directly over $G^{(L)}$. The obtained backend sequence $s_*^{(E)}$ includes every element in P_{d_f+1} exactly $n - d_f - 1$ times, so that it satisfies $M_{d_f+1}^{(d_f)}(s_*^{(E)}) = 0$ and is a solution of problem 4. However, the length of $s_*^{(E)}$ amounts to ${}_{n_{max}}P_{d_f+2}$, about n_{max} times the length of a well-structured backend sequence derived via a Hamiltonian cycle. Since the higher length directly causes an increase in memory usage, SACH derives a backend sequence with a Hamiltonian cycle as described in this subsection.

5.2 Slow-Update

This subsection gives the slow-update algorithm, which consistently updates backend sequences such as backend-full sequences introduced in the previous subsection while holding down $M_{d_f+1}^{(d_f)}(s)$. To achieve consistency, slow-update does not change the length of given backend sequences and remap buckets only between inserted or removed backends and other backends. If small $M_{d_f+1}^{(d_f)}(s)$ can be maintained via such processes and their computing time is short, we can continuously obtain well-structured backend sequences for the changing set of backends.

The basic idea of slow-update is to find the optimal backend sequence s that minimizes $M^{(d_f)}_{d_f+1}(s)$ among possible backend sequences against the change of the backend set. However, the number of the possible backend sequences is too huge to explore, so slow-update takes a greedy approach where backends are inserted/removed one by one and buckets are remapped one by one while being locally optimized.

5.2.1 Remove Backends

When removing a backend b^-, the slow-update algorithm must replace every b^- in a given backend sequence with another backend. The slow-update algorithm sequentially replaces all the buckets allocated to b^- with other backends, considering all of $M^{(d_f)}_k(s)(k \leq d_f + 1)$ to avoid an accumulation of approximation errors. The details are in Algorithm 1 and are described in the following with Fig. 8.

Algorithm 1 Remove Backend b^- in Backend Sequence s

```
Require:
    each function is given $s$, $B$, and $d_f$
1:  function REMOVEBACKEND($b^-$)
2:      for all $i$ s.t. $s_i = b^-$ do
3:          $B_{cand} \leftarrow B \setminus \{s_{i-d_f}, \dots, s_i = b^-, \dots, s_{i+d_f}\}$
4:          for $len = 1, \dots, d_f + 1$ do
5:              $score_{max} \leftarrow -|s|, B'_{cand} \leftarrow \emptyset$
6:              for all $b \in B_{cand}$ do
7:                  if $SCORE(i, b, len) \geq score_{max}$ then
8:                      $B'_{cand} \leftarrow \{b\}$
9:                      $score_{max} \leftarrow SCORE(i, b, len)$
10:                 else if $SCORE(i, b, len) = score_{max}$ then
11:                     $B'_{cand} \leftarrow B'_{cand} \cup \{b\}$
12:                 end if
13:             end for
14:             $B_{cand} \leftarrow B'_{cand}$
15:         end for
16:         $s_i \leftarrow B_{card}[0]$
17:     end for
18:     return $s$
19: end Function
20: ----------------------------------------------------------------
21: function SCORE($i, b, len$)
22:     $s_i \leftarrow b$
23:     $P^-_{len} \leftarrow P_{len} \setminus \{p \in P_{len} \mid \exists i, p_i = b^-\}$
24:     $l' \leftarrow \sum_{p \in P^-_{len}} num(s, p)$
25:     $\mu \leftarrow l' / |P^-_{len}|$
26:     $P^+ \leftarrow \bigcup_{j=1}^{len} \{(s_{i-len+j}, \dots, s_{i+j-1})\}$
27:     return $\sum_{p \in P^+} |\ num(s,p) - 1 - \mu\ | - |\ num(s,p) - \mu\ |$
28: end Function
```

For each b^- in a given backend sequence s, the loop beginning at line 2 finds a proper alternative backend. B_{cand} in line 3 indicates the set of temporal candidate backends and is to be narrowed down with $M^{(d_f)}_k(s)(k \leq d_f + 1)$. In initializing B_{cand}, not only b^- but also the backends that are responsible for the buckets near b^- (i.e., {A, D} in the first

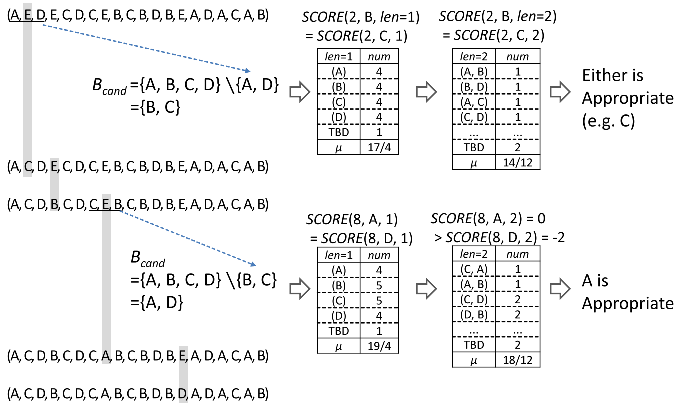
(A, E, D, E, C, D, C, E, B, C, B, D, B, E, A, D, A, C, A, B)
B_{cand} ={A, B, C, D} \{A, D} ={B, C}
SCORE(2, B, len=1) = SCORE(2, C, 1)
len=1 | num; (A) 4; (B) 4; (C) 4; (D) 4; TBD 1; μ 17/4
SCORE(2, B, len=2) = SCORE(2, C, 2)
len=2 | num; (A, B) 1; (B, D) 1; (A, C) 1; (C, D) 1; ... ...; TBD 2; μ 14/12
Either is Appropriate (e.g. C)
(A, C, D, E, C, D, C, E, B, C, B, D, B, E, A, D, A, C, A, B)
(A, C, D, B, C, D, C, E, B, C, B, D, B, E, A, D, A, C, A, B)
B_{cand} ={A, B, C, D} \{B, C} ={A, D}
SCORE(8, A, 1) = SCORE(8, D, 1)
len=1 | num; (A) 4; (B) 5; (C) 5; (D) 4; TBD 1; μ 19/4
SCORE(8, A, 2) = 0 > SCORE(8, D, 2) = -2
len=2 | num; (C, A) 1; (A, B) 1; (C, D) 2; (D, B) 2; ... ...; TBD 2; μ 18/12
A is Appropriate
(A, C, D, B, C, D, C, A, B, C, B, D, B, E, A, D, A, C, A, B)
(A, C, D, B, C, D, C, A, B, C, B, D, B, D, A, D, A, C, A, B)

Fig. 8. Example of slow-update execution.

example of Fig. 8) are removed from the set of backends B. With this initialization, backend sequences after slow-update also belong to $S_{P_{d_f+1}}$, so slow-update can be repeatedly applicable.

The sub-loop beginning at line 4 narrows down the candidate backends B_{cand} to finally determine an alternative backend, while considering suppressing the adverse effects of taking a greedy approach in replacing individual buckets. The narrowing process utilizes all the metrics $M^{(d_f)}_i(s)(1 \leq i \leq d_f + 1)$, whereas backend-full sequencing considers only $M^{(d_f)}_{d_f+1}(s)$. This is because small $M^{(d_f)}_i$ for small i is a prerequisite for small $M^{(d_f)}_i$ for large i, and $M^{(d_f)}_i$ for small i is easier to suppress as follows. The percentage of permutations generated by replacing one bucket for P_i decreases exponentially with respect to i due to the constraints of the backends before and after, so the greedy approach can not effectively control $M^{(d_f)}_i$ for large i. Also, emphasizing $M^{(d_f)}_i$ for small i is consistent with SACH's emphasis on a small number of backend failures.

Score function evaluates each candidate backend. As $M^{(d_f)}_i(s)$ is defined as the summation of the deviation of every permutation's appearance frequency, $M^{(d_f)}_i(s)$ difference caused by a b^- replacement is reduced to the summation of the small number of deviations, which are related to the permutations P^+ generated by the replacement (shown in lines 26 and 27). Here, when calculating the deviations, the average appearance frequency $\mu = l/|P_i|$ needs compensating. As shown in lines 23–25, the permutations including b^- must be removed from P_i, and the number of permutations including b^- is taken from l.

That is the slow-update algorithm for removing one backend. When removing multiple backends, all we have to do is repeat that process as many times as the number of the removed backends. As described above, that process can be repeatedly applicable, because slow-updated sequences in $S_{P_{d_f+1}}$ are also in $S_{P_{d_f+1}}$.

5.2.2 Insert Backends

To insert a backend, SACH restores the backend sequence to the one before the latest backend removal. Although we can conceptually use an exploratory update process like removing backends, the exploratory process takes much

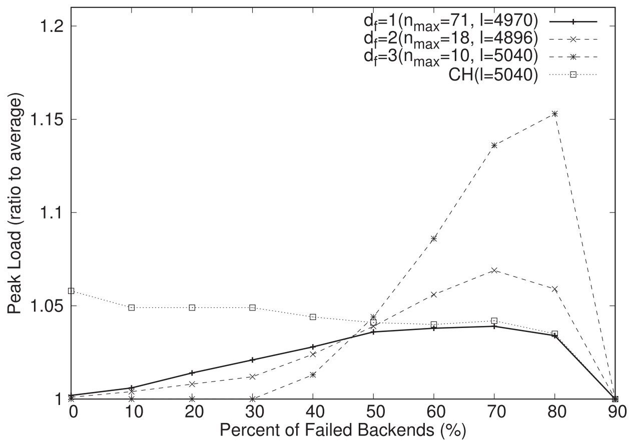

Fig. 9. Load Balancing of SACH for different $d_f (n = 10)$. The vertical line shows the load ratio of the peak-loaded backend to the average. Its smaller value indicates better load balancing and the bottom (1) is ideal (all backends have the same amount of load). Since l cannot be divided by the number of permutations for $d_f = 1, 2$, load balancing is not ideal in the cases.

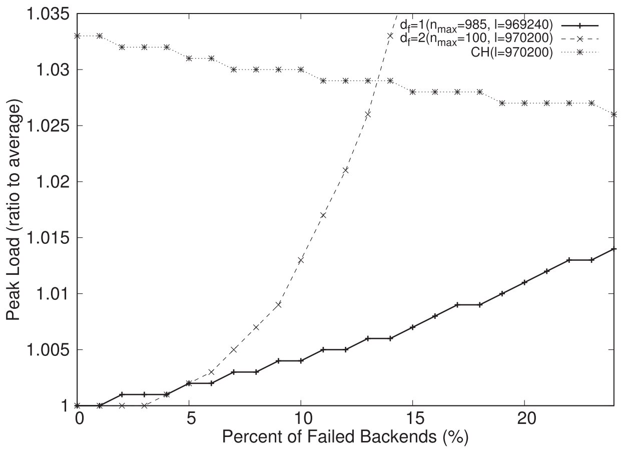

Fig. 10. Load Balancing of SACH for different $d_f (n = 100)$.

computing time and increases approximation errors because of the increased freedom of selecting buckets to replace. In return for taking a restore-based approach, n_{max} in backend-full sequencing becomes the maximum number of the backends. Therefore, the process of inserting backends is as described below. Fig. 3 also shows an example.

1) When applying backend-full sequencing, set n_{max} larger than the actual number of the backends.
2) Repeat RemoveBackend function until the distinct number of backends equals the actual number.
3) When inserting a backend, roll back the backend sequence.

6 Experimental Results and Evaluation

Although SACH is designed to satisfy consistency and fault-tolerance as described in Section 5, we have to assess whether it can satisfy other properties: load balancing, memory usage, and lookup performance. Besides, the computing time of backend-full sequencing and slow-update must be verified to clarify how much SACH can scale. Also, we investigated the effect of the setting of the SACH-specific parameter d_f.

First, we clarify the proper setting of the SACH-specific parameter d_f in Section 6.1. Second, we evaluate load balancing, memory usage, and lookup performance of SACH with the proper d_f compared with those of existing algorithms. In the comparison, memory usage in every algorithm was set the same, because load balancing and lookup performance vary depending on memory usage. Results for load balancing and lookup are in Sections 6.2 and 6.3, respectively. When measuring load balancing with failed backends, we show the average value of 100 trials where failed backends were selected at random. Finally, we evaluated the scalability of slow-update in Section 6.4.

Next, we explain how to equally set memory usage of every algorithm. In SACH, the number of buckets or the length of backend sequence l is determined to be ${}_{n_{max}}P_{d_f+1}$ for given n_{max} and d_f, as described in Section 5. Therefore, we fixed the memory usage of SACH first and approximated memory usage of the other algorithms to the fixed value l as far as possible. In concrete terms, the number of the buckets in Maglev, M, was set to the first prime after l, and the number of the virtual nodes for each backend was chosen to be the minimum integer equal to or greater than l/n in CH.

Finally, all the following experiments were conducted on a single core of a machine with an Intel i7-6700 processor running at 3.4 GHz and 32 GB of RAM on Windows 10 Pro. All algorithms were implemented in Python 3. SHA-1 (160 bits) is used in hashing keys. In particular, the lookup function of SACH specifies the corresponding bucket in the hash table by multiplying hashed keys by $l/2^{160}$ (rounded down to the nearest whole number).

6.1 Setting of Parameter d_f

To evaluate the effect of d_f setting, we compared load balancing for different values of d_f. Again we set memory usage approximately equal for different d_f as in the following, and lookup performance of SACH should not change and so is not worth evaluating. In SACH the length of a backend sequence l equals ${}_{n_{max}}P_{d_f+1}$. Therefore, we can keep memory usage approximately equal by setting n_{max} large for small d_f. First, for the maximum d_f, n_{max} was set to the number of backends n and its backend sequence was obtained via backend-full sequencing. For smaller d_f, $n_{max} > n$ was chosen to be the maximum so that ${}_{n_{max}}P_{d_f+1} < {}_nP_{\max(d_f)+1}$ and the obtained backend-full sequence was slow-updated $n_{max} - n$ times.

In this experiment, we set the number of the backends $n = 10$ and 100 and compared load balancing in the range of $d_f \in \{1, 2, 3\}, \{1, 2\}$ respectively. Figs. 9 and 10 show the results of the $n = 10$ and 100 cases, respectively. The vertical line shows the load ratio of the peak-loaded backend to the average. Its smaller value indicates better load balancing and the bottom (1) is ideal because the peak load equals the average, i.e., all backends have the same amount of load. The horizontal one shows the percent of the failed backends. The backend sequence is fast-updated for a backend failure. Also, the results with CH are shown in both figures.

First, we can see the expected effect of SACH algorithms; $d_f = 3$ and $d_f = 2$ cases in Figs. 9 and 10 indicate that SACH

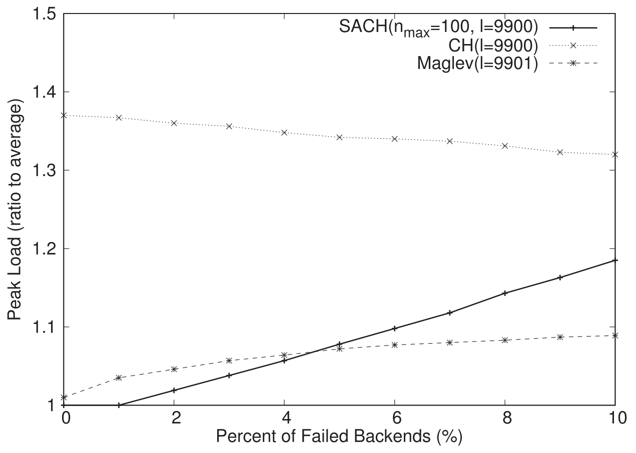

Fig. 11. Load balancing compared with CH and Maglev ($n, n_{max} = 100$).

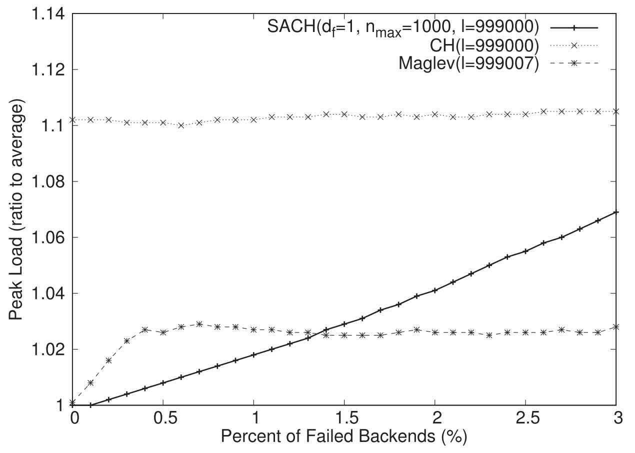

Fig. 12. Load balancing compared with CH and Maglev ($n, n_{max} = 1000$).

achieves ideal load balancing for the backend failures of up to 2 and 1, respectively. Load balancing is also expected to be good but not ideal with few failures for other cases. The micro imbalance is caused by indivisibility. The length of the backend sequence $l = {}_{n_{max}}P_{d_f+1}$ cannot be divided by the number of the permutations ${}_nP_{d_f+1}$. As many slow-updated backend sequences have this problem, truly ideal load balancing is not achievable even with large d_f.

Next, the effect of d_f setting in Fig. 9 is analyzed. For a small number of backend failures (fewer than about 50 percent), SACH performed better than CH regardless of d_f setting, and larger d_f led to better load balancing. On the other hand, the situation was reversed for a large number of backend failures (more than about 50 percent). SACH performed worse than CH, and large d_f led to worse load balancing. However, as backend failures do not frequently happen, Fig. 9 seems to suggest that d_f be as large as possible.

In contrast, Fig. 10 shows a different trend. Although it is true that larger d_f tended to be better for a small number of backend failures, the threshold of *small* noticeably decreased. Closer investigation revealed that large d_f was superior to smaller d_f only if there were fewer than five failed backends (not percent) regardless of the number of the backends n. In addition, smaller d_f also performed load balancing well in that case. Moreover, with large d_f, load balancing performance declined quickly when the number of the failed backends became larger than the threshold around five. The range in which SACH was superior to CH was narrowed as d_f increased. The graph of CH was somewhat flat, whereas SACH took a particular note on a small number of failures. Large d_f is supposed to strengthen local optimality around 0 failures and to deteriorate global balancing characteristics in return.

From these results, d_f should be one for cloud infrastructure, whose applications might scale greatly. When the number of backends is huge, the range in which large d_f outperforms small d_f is very narrow. Moreover, the difference between large and small d_f is minimal in the range, whereas large d_f is inferior to small d_f outside the range. Besides, with large d_f, high memory usage cannot be avoided, because the length of the backend sequence exponentially increases with d_f. For example, when $n_{max} = 1000$ and $d_f = 2$, l amounts to ${}_{1000}P_3 \approx 10^9$ (backends might number 1,000 in real applications [7]). We need no less than 4 GB memory to store such a backend sequence when allocating 4 bytes for one bucket. This high memory usage makes the hash space allocation (or a backend sequence) difficult to transfer between controllers and muxes. Therefore, this paper sets $d_f = 1$ in the following experiments.

6.2 Load Balancing

This subsection compares the load balancing performance of SACH with that of CH or Maglev hashing under various settings. For SACH parameters, d_f was set to 1 in accordance with the results in Section 6.1, and two patterns of n_{max} were chosen: $n_{max} = n$ and $n_{max} > n$. Whereas a backend sequence was obtained via only backend-full sequencing in $n_{max} = n$ cases, $n_{max} > n$ cases also utilized slow-update. For Maglev/CH, the size of the lookup table/the number of virtual nodes was set to be near the length of backend sequences of SACH, as mentioned at the beginning of this section.

In addition to the parameter settings, there is something to note about Maglev. To make the conditions equal, this paper operated a consistency-conscious version of Maglev hashing. Maglev hashing has a trade-off between load balancing and consistency in updating lookup tables. Load balancing-conscious Maglev hashing remaps some buckets among backends that are neither removed nor inserted; i.e., it can enhance load balancing by sacrificing some consistency. Inversely consistency-conscious Maglev hashing generates some load imbalance in updating lookup tables. As both SACH and CH strictly satisfy consistency, this paper used consistency-conscious Maglev for comparison. In particular, the experiments were conducted while avoiding the deterioration of the load balancing performance of Maglev. The load balancing performance of consistency-conscious Maglev drops every time its lookup table is updated. Therefore, the experiments in this section used lookup tables that had not been updated since their initialization. For example, in $M = 65,537$ setting [7], when a lookup table was initialized with 10 backends and updated for 100 backends, the average peak-load ratio amounted to 104 percent without backend failures.

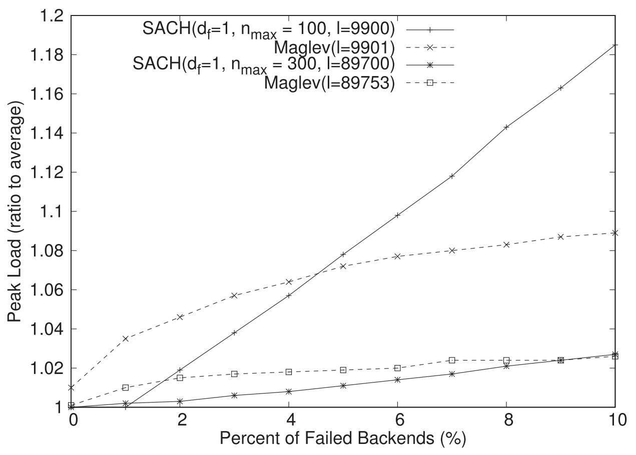

Fig. 13. Load balancing compared with Maglev ($n = 100 \leq n_{max}$).

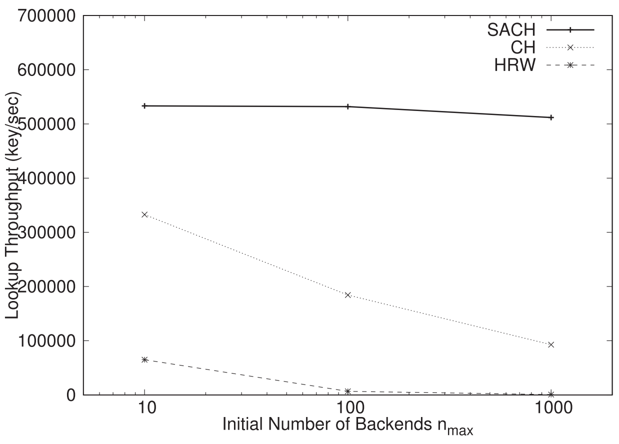

Fig. 14. Lookup throughput compared with CH and HRW.

For $n_{max} = n$ cases (without slow-updated backend sequences), Figs. 11 and 12 show the verified results for $n = 100$ and 1000, respectively. Both figures indicate the following:

- SACH and Maglev hashing achieved better load balancing than CH for a small number of failed backends.
- SACH reached the same level of load balancing as Maglev hashing for a small number of backend failures, even though fast-update of SACH did not include exploratory processes, on which Maglev hashing depended.
- Especially for around zero backend failures, load balancing of SACH was better than that of Maglev hashing. (The curve of SACH was downwardly convex, so that its load balancing was almost ideal around zero backend failures.)

Therefore, for $n_{max} = n$ cases, SACH can be said to achieve both fault-tolerance and load balancing and provide the best load balancing for a small number of backend failures.

The results of $n_{max} > n$ cases (with backend sequences that were slow-updated) are shown in Fig. 13, which also includes the result of the $n_{max} = n = 100$ case in Fig. 11 for comparison. As a consequence of slow-update, load balancing of SACH ($n_{max} = 300$) became non-ideal against just one failed backend, while the number of failed backends equaled $d_f (= 1)$. However, the load balancing of SACH was sufficiently even and was still better than that of Maglev hashing (and CH, of course) for a small number of failed backends (under around 10 percent of failed backends). Therefore, SACH can be again said to achieve both fault-tolerance and load balancing and provide the best load balancing for a small number of backend failures.

Moreover, the results in Fig. 13 suggest other pros of SACH. The load balancing of SACH could be enhanced more than that of CH and Maglev hashing as memory usage increases. In the $n_{max} = n = 100 (l \approx 10,000)$ cases, the advantage and disadvantage for load balancing of SACH and Maglev hashing reversed at about 5 percent of backends failed. The threshold increased to about 9 percent in the $n_{max} = 300 (l \approx 90,000)$ cases. This relationship is applicable between SACH and CH (not shown in the figures). Whereas the threshold was around 14 percent for $n_{max} = 100 (l \approx 10,000)$ cases, it reached around 24 percent for $n_{max} = 300 (l \approx 90,000)$ cases. This is thought to be caused by injecting a certain good randomness via slow-update, whereas backend-full sequencing is quite deterministic. As CH has shown, random allocation of the hash space makes the load balancing curve somewhat flat; i.e., load balancing does not deteriorate for the increased backend failures. For example, when $n = n_{max}, d_f = 1$, a subsequence (A, B) appears exactly once in a backend-full sequence. Therefore, if both A and B fail, the two buckets are replaced with another backend in bulk, and such replacement directly generate load imbalance. If l increases with n_{max}, the number of appearances of subsequence (A, B) rises and the variation of the following backends increases almost linearly (when the number of the failed backends is much smaller than the number of all the backends). Therefore, the load balancing of SACH could be more enhanced as memory usage increases.

6.3 Lookup

When looking up a backend responsible for a given key, SACH does not require exploratory processes in contrast to CH. As the hashing space of SACH is evenly divided into buckets, the allocation of the space is equivalent to a hash table. Therefore, SACH specifies the responsible backend with just one hashing calculation, as well as Maglev hashing. This subsection compares the lookup performance of SACH with those of CH and HRW, which have exploratory processes or multiple hashing calculations.

Fig. 14 shows the lookup performances for different sizes of lookup tables. The vertical line shows the average number of lookup processes completed in a second (100 trials). The horizontal one shows n_{max} in SACH instead of the size of lookup tables. The number of the virtual nodes in CH was set equal to the length of backend sequences in SACH; i.e., there were 90, 9900, and 999,000 virtual nodes for $n_{max} = 10, 100$, and 1000, respectively. The number of the virtual nodes or the length of backend sequences was about n_{max}^2.

From the results, the performance of SACH was almost independent of n_{max}, whereas those of CH/HRW decreased as n_{max} increased. Therefore, as well as SACH performing

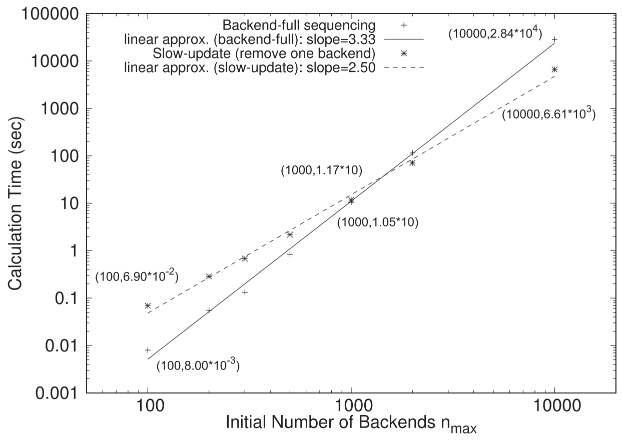

Fig. 15. Calculation time of structured-allocation maintenance.

the best regardless of n_{max}, the difference was considerable for larger n_{max}. With regard to HRW, its performance was inversely proportional to n_{max} owing to n_{max} hashing calculations, so that the performance of SACH amounted to about 100 times that of HRW for $n_{max} = 100$. When it comes to CH, its performance seemed inversely proportional to $\log(n_{max})$ and less than half of SACH's performance, as a consequence of binary searches in the hash space. The above has proved that SACH achieves the lookup property.

6.4 Scalability

Finally, we evaluated the scalability of SACH. Although SACH achieves both fault-tolerance and load balancing by generating well-structured backend sequences, SACH cannot scale if it takes much time to generate or update backend sequences. This subsection presents the computation time of backend-full sequencing and slow-update to verify how much SACH can scale.

Fig. 15 provides the computation time of backend-full sequencing and slow-update for some n_{max}. Each computation time was almost proportional to n_{max} with a double-logarithmic scale. For backend-full sequencing and slow-update, the slopes of linear approximations were 3.3 and 2.5 and the computation times for $n_{max} = 1000$ were 10.5 and 11.7 sec, respectively (the reason slow-update is slower than backend-full sequencing for $n_{max} < 1000$ is that $num(s, p)$ calculations for all permutations are time-consuming before the following exploratory process). These results indicate that n_{max} in SACH should be less than around 10^5. As the number of backends was supposed to be less than 1,000 even in Google data centers [7], SACH can be sufficiently feasible.

The following describes a comparison with the update time of Maglev hashing. With $n = 1000$ and $M = 999,001$, it took about 5 seconds to update if a large-scale permutation table (about 4GB) necessary for the update process is stored in memory in advance. Otherwise, it took as long as about 165 seconds. While the former is faster than slow-update, it is insufficient for fault-tolerance and is memory usage-intensive (always 4 GB). In contrast, SACH is fault-tolerant because it uses the fast-update algorithm to update the hash table against backend failures. On the other hand, the latter takes considerably longer than slow-update. Since this tendency is the same for other n, the update process of Maglev hashing has a problem of resource-intensity, and SACH is expected to able to improve it.

7 Discussion

The experimental results suggest that SACH could replace CH in many cloud infrastructure applications. SACH is the first hashing algorithm that satisfies all the five requirements (consistency, load balancing, lookup time, memory usage, and fault-tolerance), whereas previous methods sacrifice one or more of them. The applicable condition of SACH is that the target system can distinguish designed insertion or removal of backends from unexpected ones such as backend failures. The condition is already satisfied in many data centers, most of which manage computing resources with a certain orchestrator.

However, SACH might not necessarily be sufficiently scalable in the future. SACH sets a capacity parameter n_{max} that bounds the maximum number of the backends. Its existence is not essential, because other algorithms substantially set similar parameters; they initially fix the number of buckets, so that load balancing is deteriorated for a larger number of backends. However, the existence of the limit of n_{max} is not desirable. As Section 6.4 has shown, n_{max} has to be up to 10,000. Even though the current version of SACH can be applied to existing Google-scale clusters [7], SACH will need to be improved in the future. Slow-update will be speeded up by parallelizing the bucket replacement processes or improving the efficiency of each bucket's replacement process that currently includes a linear search. In either case, load balancing performance will slightly decrease. However, as with the load balancing-conscious Maglev, the demand for consistency can be slightly relaxed and load balancing performance restored.

SACH has also a problem in memory usage. SACH uses sufficiently small memory to be implemented as software but too much memory to be implemented as hardware in routers and other devices. For example, its memory usage amounts to about 4 MB for $d_f = 1, n_{max} = 1000$. As this is because the memory usage of SACH goes up to $\mathcal{O}(n_{max}^{d_f+1})$, memory usage is expected to be lowered. Memory usage is determined by the length of the backend-full sequence. Therefore, we can decrease it by relaxing the condition that backend-full sequencing uses every permutation exactly once. Specifically, by removing some edges in $G^{(U)}$ while considering the metrics $M_i^{(d_f)}$ and keeping the graph Eulerian, we will reduce the memory usage.

In addition to discussing the drawbacks of SACH, the following discusses two assumptions of SACH: the number of simultaneous failures is small, and the keys uniformly distribute over the hash space.

Regarding the number of simultaneous failures, as cloud dependability analyses [18], [19], [25] show, the rate of simultaneously failed backends seems to be low (under two percent) as long as distributed systems operate normally. However, when a large-scale shared resource is lost due to a disaster or power supply loss, an increased number of backend failures must happen. This will significantly reduce the load balancing performance of SACH. However, when simultaneous failures occur due to loss of shared resources, the combination of backends that are likely to fail at the same time can be known in advance. Extending SACH by utilizing this prior information could suppress the load imbalance. For example, in generating a backend sequence, it might be useful to avoid adjoining backends that can simultaneously fail.

Next, we will discuss the uniformity of the key distribution. With a large number of hash keys, their distribution can be considered uniform. On the other hand, if the number of keys is not enough, their distribution has a non-negligible size of bias. Therefore, the following describes how many keys are required for SACH to perform as desired. Let n be the number of backends, m be the number of keys, and P be the allocated hash space ratio to the average associated with a specific backend. Then the distribution of the number of keys related to the backend can be approximated by the Poisson distribution of m/nP when both n and m are large to some extent. Therefore, about its ratio to the average number of allocated keys m/n, the expected value $E = P$, and the variance $\sigma = \sqrt{n/mP}$. Since the expected value is P, P still needs to be reduced as in SACH. On the other hand, if σ is not small, a larger load imbalance will occur. Specifically, it is desirable that the number of keys per backend $m/n > 1000$. In this case, the load ratio deviation can be suppressed within 10 percent with high probability. Considering the number of connections per web-server and the number of keys in key-value stores, $m/n > 1000$ will hold in many cases.

8 Conclusion

This paper proposed Structured Allocation-based Consistent Hashing (SACH), a consistency-equipped hashing algorithm that simultaneously satisfies all five properties for cloud infrastructure: consistency, load balancing, lookup time, memory usage, and fault-tolerance. SACH is designed for cloud infrastructure-like environments, where backend provisioning is under control and the percentage of simultaneously failed backends is small.

First, SACH uses an architecture that uses different sub-algorithms between designed backend additions/removals and backend failures/recoveries. This architecture makes it possible to take enough time to reallocate the hash space for a designed update, while the allocation can be updated so fast that the cluster maintains fault-tolerance. Next, SACH models an allocating problem as a backend sequencing one to capacitate mathematical problem formulation and provides polynomial-time algorithms for the formulated load balancing problem.

The experimental results showed that SACH with equivalent memory usage performs equally to or better than existing algorithms in terms of both load balancing and lookup rate as long as the percentage of simultaneously failed backends is small. Here, the threshold of failure percentage under which SACH performs better was about 10 and 1.5–10 percent against CH and Maglev, respectively. Since SACH is designed to satisfy consistency and fault-tolerance properties, the results demonstrated that it satisfies all the five properties.

SACH will contribute both industrially and academically. Since SACH can increase load balancing performance without sacrificing other properties in cloud infrastructure environments, many industrial applications with CH could improve their load balancing by using SACH. Besides, the allocation modeling that this paper introduced will make it possible to achieve other types of consistency-equipped hashing algorithms for different requirements. However, SACH still has many issues. It will be improved for example, by adapting it to systems with heterogeneous backends, reducing its memory usage, and increasing its scalability.

Acknowledgments

The author would like to thank members of Server Network Innovation Project in NTT Network Service Systems Laboratories for their helpful discussions and comments on the manuscript.

References

[1] Kubernetes. Accessed: Feb. 8, 2021. [Online]. Available: https://github.com/kubernetes/kubernetes
[2] OpenStack Heat. Accessed: Feb. 8, 2021. [Online]. Available: https://wiki.openstack.org/wiki/Heat
[3] B. Appleton and M. O'Reilly, "Multi-probe consistent hashing," 2015, *arXiv:1505.00062*.
[4] J. T. Araújo, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the edge: Transport affinity without network state," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 111–124.
[5] N. G. De Bruijn and P. Erdös,"On a combinatorial problem," *Indagationes Math*, vol. 10, pp. 421–423, 1948.
[6] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.
[7] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. 13th Usenix Conf. Netw. Syst. Des. Implementation*, 2016, pp. 523–535.
[8] D. Halperin *et al.*, "Demonstration of the Myria big data management service," in *Proc. 2014 ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 881–884.
[9] A. Holroyd, F. Ruskey, and A. Williams, "Faster generation of shorthand universal cycles for permutations," in *Proc. Int. Comput. Combinatorics Conf.*, 2010, pp. 298–307.
[10] B. W. Jackson, "Universal cycles of k-subsets and k-permutations," *Discrete Math.*, vol. 117, no. 1–3, pp. 141–150, 1993.
[11] D. Karger, E. Lehman, T. Leighton, T. Levine, D. Lewin, and R. Panigraphy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[12] D. Karger *et al.*, "Web caching with consistent hashing," *Conputere Netw.*, vol. 31, no. 11–16, pp. 1203–1213, 1999.
[13] P. Kumar and R. Kumar, "Issues and challenges of load balancing techniques in cloud computing: A survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–35, 2019.
[14] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[15] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," 2014, *arXiv:1406.2294*.
[16] G. Mendelson, S. Vargaftik, K. Barabash, D. H. Lorenz, I. Keslassy, and A. Orda, "AnchorHash: A scalable consistent hash," *IEEE/ACM Trans. Netw.*, to be published, doi: 10.1109/TNET.2020.3039547.
[17] V. Mirrokni, M. Thorup, and M. Zadimoghaddam, "Consistent hashing with bounded loads," in *Proc. ACM 29th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2018, pp. 587–604.
[18] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, "Cloud dependability analysis: Characterizing Google cluster infrastructure reliability," in *Proc. 3th Int. Conf. Web Res.*, 2017, pp. 56–61.
[19] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, "Highly reliable architecture using the 80/20 rule in cloud computing datacenters," *Future Gener. Comput. Syst.*, vol. 77, pp. 77–86, 2017.

[20] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 125–139.
[21] F. Ruskey and A. Williams, "An explicit universal cycle for the (n-1)-permutations of an n-set," *ACM Trans. Algorithms*, vol. 6, no. 3, 2010, Art. no. 45.
[22] M. Sackman, "Perfect consistent hashing," 2015, *arXiv:1503.04988*.
[23] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003.
[24] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Trans. Netw.*, vol. 6, no. 1, pp. 1–14, Feb. 1998.
[25] K. V. Vishwanath, and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 193–204.
[26] X. Wang and D. Loguinov, "Load-balancing performance of consistent hashing: Asymptotic analysis of random node join," *IEEE/ACM Trans. Netw.*, vol. 15, no. 4, pp. 892–905, Aug. 2007.
[27] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, "Load balancing in data center networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 2324–2352, Third Quarter 2018.

Yuichi Nakatani received the BS and MS degrees in informatics from Kyoto University, in 2003 and 2005, respectively. Since joining NTT in 2005, he has been engaged in researching distributed systems and network security and developing of network services. He is a member of IEICE.