PAPER

Suzaku: A Churn Resilient and Lookup-Efficient Key-Order Preserving Structured Overlay Network

Kota ABE[†,††a)] ***and*** **Yuuichi TERANISHI**[††,†††], ***Members***

SUMMARY A *key-order preserving structured overlay network* is a class of structured overlay network that preserves, in its structure, the order of keys to support efficient range queries. This paper presents a novel key-order preserving structured overlay network "*Suzaku*". Similar to the conventional Chord$^\#$, Suzaku uses a periodically updated finger table as a routing table, but extends its uni-directional finger table to bi-directional, which achieves $\lceil \log_2 n \rceil - 1$ maximum lookup hops in the converged state. Suzaku introduces active and passive bi-directional finger table update algorithms for node insertion and deletion. This method maintains good lookup performance (lookup hops increase nearly logarithmically against n) even in churn situations. As well as its good performance, the algorithms of Suzaku are simple and easy to implement. This paper describes the principles of Suzaku, followed by simulation evaluations, in which it showed better performance than the conventional networks, Chord$^\#$ and Skip Graph.

key words: *peer-to-peer systems, key-order preserving structured overlay network, churn resilience*

1. Introduction

Structured overlay networks provide a scalable lookup service to locate target nodes (e.g., computers, network devices), specified by a key, using autonomous and cooperative routing among nodes on the application layer.

Interest has been increasing for a type of structured overlay called *Key-order preserving structured overlay network* (hereafter, KOPSON), which preserves the order of keys in its structure. In KOPSON, when the keys of two nodes are adjacent, these two nodes are also adjacent in the overlay network. This property enables the execution of the bulk lookup of nodes in arbitrary key intervals, which is necessary for range queries. Such queries are useful for a variety of distributed applications such as application layer multicasts (ALMs), on-line games [1], distributed pub/sub systems [2], and distributed spatial-temporal databases [3].

Existing structured overlays for distributed hash tables such as Chord [4] use hash functions to distribute unevenly distributed keys to nodes uniformly. However, because hashing scatters adjacent keys to distant, unrelated nodes, overlays of this type do not support range queries. KOPSONs tackle the unevenly distributed keys issue through different approaches. To preserve key-order in the network, they do not use hash functions. Instead, they construct multi-level routing tables whose pointers point to distant nodes to achieve scalable lookup.

KOPSONs often suffer from node churn, i.e., a lot of nodes are inserted or deleted in a short time. For example, in a distributed pub/sub system, many subscribers may subscribe to particular content in a short time (flash crowd), and in ALM, all receivers may leave just after the session finishes. Therefore, KOPSONs should have high churn robustness; they should keep good lookup performance even when churn occurs.

There are roughly two approaches for implementing a KOPSON. One approach is the *periodic approach*, which gradually and continuously improves routing table entries by periodic data collections. Mercury [5] and Chord$^\#$[6] are of this type. This type of approach has a drawback in churn situations; the maximum number of lookup hops becomes $O(n)$ (where n is the number of inserted nodes) after a large number of nodes are inserted in a short time because several data collection periods are required to obtain a mature routing table.

The other approach is the *active approach*, in which each node actively updates routing table entries of itself and other nodes on its insertion or deletion. Skip Graph [7] is a well-known KOPSON of this type. The structure of Skip Graph is determined by random values that each node generates. Skip Graph keeps a scalable average lookup performance ($O(\log n)$ hops) with a high probability even in churn situations. However, because random values are used to determine the structure of the network, the maximum number of hops is unbounded and typically is as many as several folds of $\log_2 n$. In addition, as described later, implementing Skip Graph is not straightforward.

In this paper, we propose a novel KOPSON *Suzaku* that addresses drawbacks in the existing schemes. Suzaku has the following notable properties:

1. When routing tables are converged, Suzaku achieves $\lceil \log_2 n \rceil - 1$ maximum lookup hops.
2. Even in the worst case, the number of lookup hops is nearly logarithmically increased and faster than Skip Graph when $n \leq$ around 100k.
3. The network traffic to maintain a Suzaku network is small.
4. The algorithm is simple and easy to implement.

Manuscript received April 23, 2018.
Manuscript revised January 12, 2019.
Manuscript publicized March 5, 2019.

[†]The author is with Graduate School of Engineering, Osaka City University, Osaka-shi, 558-8585 Japan.
[††]The authors are with National Institute of Information and Communications Technology, Koganei-shi, 184-8795 Japan.
[†††]The author is with Cybermedia Center, Osaka University, Suita-shi, 565-0871 Japan.
a) E-mail: k-abe@osaka-cu.ac.jp

DOI: 10.1587/transcom.2018EBT0001

The approach of Suzaku is a hybrid of the previously mentioned approaches; routing table entries are updated both periodically and actively. In addition, Suzaku introduces and integrates several techniques such as bi-directional finger tables, passive updates, and reverse pointers, whose details are described in the following sections.

This paper is organized as follows. In Sect. 2, we review related work. Section 3 describes Suzaku. Section 4 evaluates Suzaku by simulations and provides some discussion. Finally, Sect. 5 summarizes this work.

2. Related Work

There have been many studies for efficient KOPSONs. In this section, we describe three well-known and typical conventional KOPSONs, namely, Mercury, Chord$^\#$, and Skip Graph.

These KOPSONs do not use a hash function to disperse keys. Nodes are sorted in the network by their keys to preserve key-order. Each node has a *successor* pointer to the next-key node and a *predecessor* pointer to the previous-key node. Furthermore, in Mercury and Chord$^\#$, the successor of the max-key node points to the min-key node, and the predecessor of the min-key node points to the max-key node. Thus, they construct a doubly linked ring (hereafter, "level 0 ring"). We assume that the clockwise direction is the key-increasing direction.

The use of only successor and predecessor pointers is insufficient to achieve a good lookup (routing) performance, so each node also maintains long distance pointers, which are stored in a routing table with multiple levels. Lower levels of the table contain pointers to closer nodes in the key space, and higher levels contain pointers to more distant nodes.

2.1 Mercury

Mercury is a periodic approach-based KOPSON. In Mercury, each node estimates the density of keys in the network by sampling nodes using random walks. Each node reconstructs its routing table on the basis of the node counts computed from the recent estimation. Such reconstruction is initiated only when the number of nodes in the system changes dramatically.

2.2 Chord$^\#$

Chord$^\#$ is another periodic approach-based KOPSON, but it does not use density estimation. The routing table of Chord$^\#$ is called a *finger table*. It is a one dimensional array whose element (*finger table entry* [FTE]) is determined with the following expression.

$$\text{finger}[i] = \begin{cases} \text{successor} & (i = 0) \\ \text{finger}[i-1] \rightarrow & \\ \qquad \text{getFinger}(i-1) & (i > 0) \end{cases}$$

For any node u, u.finger[0] is equal to u's successor pointer. u.finger[i] ($i > 0$) is updated by fetching finger[$i-1$] of the node that is referenced by u.finger[$i-1$]. To fetch finger[$i-1$], u sends a request message and receives a reply message. All nodes update their finger tables periodically. When no node insertions or deletions occurs, the finger tables of all nodes are eventually converged. When converged, finger[i] of each node points to the 2^i distant node in the clockwise direction and the maximum number of lookup hops is bounded to $\lceil \log_2 n \rceil$.

2.3 Skip Graph

Skip Graph is an active approach-based KOPSON. Each Skip Graph node has a random value called *membership vector* (MV). A routing table of Skip Graph is composed of multiple levels of doubly linked lists. At level 0, each node has pointers to its successor and predecessor nodes. At level i ($i > 0$), each node u has pointers to the closest nodes in both directions, in which the first i digits (prefix) of MV are equal to u's one. These doubly linked lists are actively updated on node insertions and deletions.

2.4 Discussion

2.4.1 Number of Lookup Hops

In Chord$^\#$, when its routing tables are converged, the maximum number of hops is $\lceil \log_2 n \rceil$. However, when routing tables are not converged, the number of lookup hops may be much larger than this bound. When a lot of nodes are inserted between two adjacent nodes in a short time (churn situation), the number of hops may be increased to $O(x)$, where x denotes the number of newly inserted nodes, because the finger tables of newly inserted nodes are not so quickly updated.

The average number of lookup hops of Skip Graph is $O(\log n)$ even in a churn situation, but in general, it is larger than $\log_2 n$ and the maximum number of hops is severalfold higher than $\log_2 n$ due to the randomness used for constructing routing tables. There are several proposals to improve lookup hops that refine the structure of a Skip Graph instance using a self-stabilizing approach [8], [9]. However, these proposals take time to reduce the number of hops to the level comparable to Chord$^\#$ and also incurs complexity.

The lookup performance of Mercury depends on the accuracy of the estimated density of keys [6] and does not excel converged results of Chord$^\#$ in general.

2.4.2 Routing to Deleted Nodes

In Mercury and Chord$^\#$, because their routing tables are *asymmetric* (a network is symmetric if, for any node p, all nodes pointed to by any routing table entries of p have a pointer to p), a leaving node cannot notify nodes whose routing table entry points to the deleting node of its leaving. Thus, in these networks, messages may be routed to an invalid node even if there is no failure. In this case, the query takes a longer time to complete because it requires retransmission after waiting for a timeout. On the other hand, Skip Graph is free from such cases because its routing table is symmetric.

2.4.3 Implementation Complexity

An instance of Skip Graph consists of multiple levels of doubly linked lists. Keeping the consistency between the multiple levels of doubly linked lists is a complicated task in a distributed environment where node insertion, deletion, and failure occur unpredictably and concurrently. As far as the authors know, no existing Skip Graph implementation supports failure recovery at a practical level. Skip Graph variants such as Rainbow Skip Graph [10], Skip Tree Graph [11], and p-Skip Graph [12] have similar implementation complexity issues. On the other hand, the structure of Chord$^{\#}$ is simple; pointers are unidirectional and just stored in finger tables. Thus, the implementation complexity of Chord$^{\#}$ is smaller than that of Skip Graph. While Mercury also uses unidirectional pointers, it requires an additional random sampling process to estimate the density of keys, which leads to a larger complexity for its implementations.

3. Suzaku

In this section, we describe the design and algorithms of *Suzaku*.

3.1 Design Principles

The design principles of Suzaku are as follows:

- **Use finger tables:** Suzaku uses finger tables like Chord$^{\#}$ as its routing table.
- **Introduce backward finger table:** Suzaku has finger tables for both directions. The conventional, clockwise direction finger table is called *forward finger table* (FFT), and the new, counterclockwise direction one is called *backward finger table* (BFT).
- **Introduce passive updates:** In Suzaku, like Chord$^{\#}$, each node p updates an FTE by sending a request message, which we call a *getEnt* message, to another node q. This type of updates is called *active update*. In addition, q also updates its opposite direction finger table. This type of updates is called *passive update*.
- **Actively update finger tables immediately after node insertion:** In Chord$^{\#}$, when a node is inserted, its finger table is empty (except in level 0). In Suzaku, an inserted node actively updates all its FFT and BFT entries immediately after being inserted at the level 0 ring. In conjunction with passive updates, this policy reduces performance degradation in the case where a lot of nodes are inserted in a short time.
- **Adopt active ring management algorithm:** To correctly update FFT and BFT entries, successor and predecessor pointers should point to the correct nodes. To manage these pointers, Chord$^{\#}$ adopts Chord [4]'s stabilization algorithm. However it requires a long time to converge when many nodes are inserted in a short time [13]. Suzaku adopts a ring management algorithm that actively updates successor and predecessor pointers upon node insertion and deletion (such as DDLL [13]).
- **Delayed finger table updates:** In Chord$^{\#}$, when updating an FTE, a node fetches a pointer from a remote node and updates the FTE immediately without checking whether the referenced node is alive or not. In Suzaku, the updating of an FTE is delayed until the referenced node is confirmed to be alive. This policy reduces the possibility of routing a message to invalid nodes.
- **Introduce reverse pointers:** In Suzaku, to avoid routing messages to already-deleted nodes, each node p has a *reverse pointer set* ($p.R$), whose element points to a node that has a pointer to p in its finger table at level 1 or above. When node p is deleted, *deletion notifications* are sent to each node in $p.R$. A deletion notification contains an alternative node (p's predecessor) and the recipient node replaces its FTEs that points to p with the alternative one.

We denote node p's FFT and BFT as p.FFT and p.BFT, respectively. p.FFT[0] and p.BFT[0] are respectively equal to the successor and predecessor pointers that are managed by the ring management algorithm.

3.2 Node Insertion

When node p is being inserted, it asks an introducer node to find its adjacent key nodes in the overlay network and inserts p into the level 0 ring using the ring management algorithm. Then, FFT and BFT entries of level 1 and above are actively updated in turn; p updates p.FFT[1], p.BFT[1], p.FFT[2], p.BFT[2] ... in this order. This initial finger table update continues until the fetched pointer reaches or passes p. (See also the message sequence diagram (Fig. 2) and the pseudo code (Fig. 3).)

The following is the procedure to update p.FFT[$i + 1$] ($i \geq 0$). Updating p's BFT is similar; substitute FFT for BFT and vice versa.

- Let q denote the node pointed to by p.FFT[i]. p sends a getEnt request to node q. Let x denote the node pointed to by q.FFT[i] (Fig. 1 left).
- When q receives the getEnt request, q sends back q.FFT[i] to p. Also p updates q.BFT[i] to p except in the case of $i = 0$ (because the successor and predecessor pointers are managed by the ring management algorithm). This update is called *Passive Update 1* (PU1).
- At this point, it is uncertain from the perspective of p whether x is alive or not. This is determined later when p fetches x.FFT[$i + 1$] from x and receives a response message from x. Thus, p delays updating p.FFT[$i + 1$]

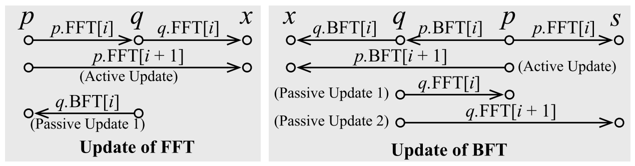


Fig. 1 Explanatory figures of FFT and BFT updates. p updates p.FFT[i+1] (left) and p.BFT[$i+1$] (right). p sends a getEnt request to q to fetch the pointer to x.

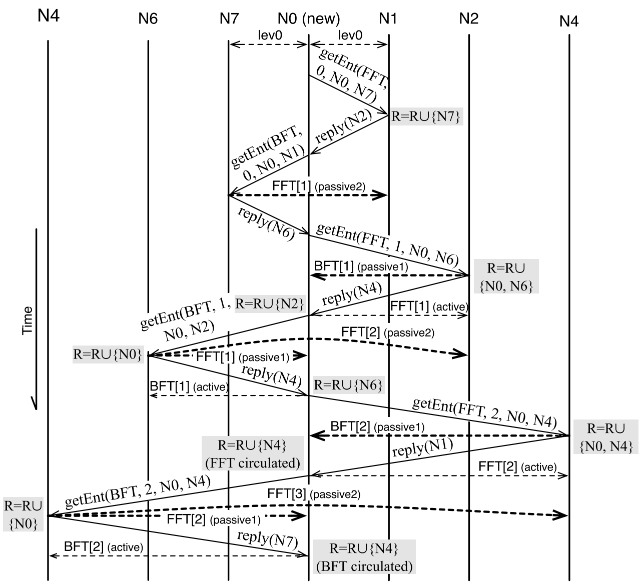


Fig. 2 Message sequence diagram of initial finger table updates. Node N0 is inserted into a network that consists of nodes N1 to N7 whose finger tables have been converged. Slanted lines show message transmission and horizontal dashed lines show finger table updates (thin ones are active and bold ones are passive). RemoveRev messages are omitted for simplicity.

```
const FFT = 0, BFT = 1;
// ft[FFT] is FFT, ft[BFT] is BFT
// ft[FFT][0] = successor, ft[BFT][0] = predecessor
var ft[2][]: Array of {Array of NodeAndKey};
// NodeAndKey is a type that contains the locator and
//   the key of a node
var R: Set of NodeAndKey; // reverse pointer set
n.initialUpdate(dir, i, n1, n2) {
  tab ← ft[dir];
  if (i = 0) n1 ← ft[dir][0]; // succ or pred
  if (n1 = null or n1 = n) {
    x ← null; goto next;
  }
  // y is used by n1 for PU2.
  if (dir = FFT and n2 ∉ (n, n1))
    y ← n2;
  else (dir = BFT and ft[FFT][i] ∉ (n1, n))
    y ← ft[FFT][i];
  else y ← null;
  x ← n1.getEnt(dir, i, n, y);
  if (n1 failed) { we omit the detail of this case }
  if (i ≠ 0) {
    // active and delayed update
    change(tab, i, n1, true);
  }
  if ((dir = FFT and x ∈ [n, tab[i]]) or
    (dir = BFT and x ∈ [tab[i], n]))
      x ← null; // circulated
next:
  if (x = null and n2 = null) return;
  if (dir = FFT) { n.initialUpdate(BFT, i, n2, x); }
  else { n.initialUpdate(FFT, i + 1, n2, x); }
}
n.getEnt(dir, level, p, q) {
  if (level ≠ 0) {
    change(ft[1 - dir], level, p, true); // PU1
  }
  if (q ≠ null) {
    if (dir = BFT) {
      change(ft[1 - dir], level+1, q, false); // PU2
    } else R ← R ∪ {q};
  }
  return ft[dir][level];
}
n.change(tab, level, new, addtorev) {
  old ← tab[level];
  tab[level] ← new;
  if (addtorev and new ≠ null) R ← R ∪ {new};
  if (old ≠ null and (FFT and BFT do not contain old)) {
    old.removeRev(n);
  }
}
n.removeRev(p) {
  R ← R \ {p};
}
```

Fig. 3 The initial finger table update algorithm.

until that point.

Let us consider the case where p fetches q.BFT[i] where q is the node pointed to by p.BFT[i] (Fig. 1 right). In this case, q.FFT[i] is updated to p by PU1. Also, because the node pointed to by p.FFT[i] (node s in the figure) is the 2^{i+1} clockwise distant node from q (when finger tables are converged) and has been confirmed to be alive, q.FFT[$i+1$] is updated to p.FFT[i]. This update is called *Passive Update 2* (PU2).

PU2 is not used when p updates its FFT. This is because when p updates p.FFT[i], p.BFT[i] is not yet updated because of delayed finger table updates (the node pointed to by the previously fetched entry for p.BFT[i] is not confirmed to be alive). Also, PU2 is not used if s is located between q and p. Such cases occur when q is farther than 1/2 in the ring from p.

Figure 2 is an example of a message sequence. A getEnt request message has four parameters. The first two indicate which entry to fetch. The third one is the locator of the request sender node and used for PU1. The last one is the same level pointer of the opposite side finger table of the sender node (i.e., BFT[i] (resp., FFT[i]) if the getEnt request is to fetch an FFT[i] (resp., BFT[i])). This pointer is used either for PU2 or for adding to a reverse pointer set.

In the figure, N0 receives a pointer to N2 from N1 but it is stored in $N0$.FFT[1] after receiving a reply message from N2 (delayed finger table updates). N6 updates $N6$.FFT[1] to N0 by PU1 and $N6$.FFT[2] to N2 by PU2. Note that N1 (resp., N7) does not update its BFT[0] (resp. FFT[0]) on receiving a getEnt request because level 0 is managed by the ring maintenance algorithm.

The algorithm of initial finger table updates is shown in Fig. 3 (Note that this figure contains algorithms described later in Sect. 3.4). When the ring management algorithm completes insertion into the level 0 ring, `initialUpdate(FFT, 0, null, null)` is executed to start the initial finger table update procedure.

3.3 Periodic Finger Table Updates

After node p finishes initial finger table updates, p periodically updates the finger tables (*periodic update mode*). In this mode, p updates p.FFT[i] every T_{ft}, increasing i from 1. When i reaches the max level, i is reset to 1. While PU1 is still used in this mode (i.e., a node that receives a getEnt

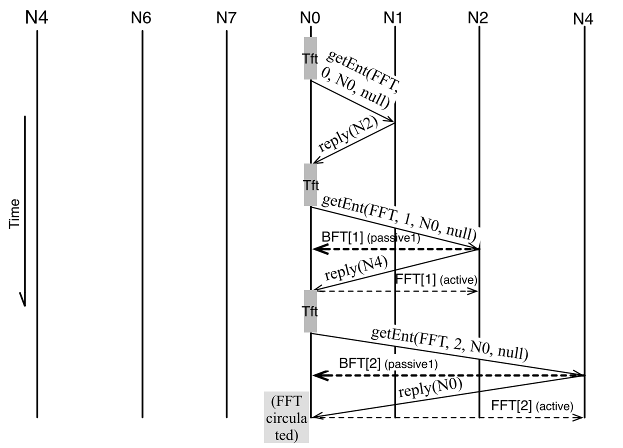


Fig. 4 Message sequence diagram of periodic finger table updates of node N0.

```
// i: level to fetch
// n1: previously fetched entry for n.FFT[i]
n.periodicUpdate(i, n1) {
  if (i = 0) n1 ← ft[FFT][0]; // successor
  else if (ft[FFT][i] is updated within past $T_{ft}$)
    // if ft[FFT][i] is passively updated after fetching n1,
    // use ft[FFT][i] instead of n1 because it is newer.
    n1 ← ft[FFT][i];
  x ← n1.getEnt(FFT, i, n, null);
  if (n1 failed) { we omit the detail of this case }
  if (i ≠ 0) {
    // active and delayed update
    change(ft[FFT], i, n1, true);
  }
  if (x ∈ [n, ft[FFT][i]]) { // circulated
    // truncate FFT and BFT to size i
    for (j←i to ft[FFT].length−1)
      change(ft[FFT],i,null,false);
    for (j←i to ft[BFT].length−1)
      change(ft[BFT],i,null,false);
    sleep($T_{ft}$);
    n.periodicUpdate(0, null);
  } else {
    // fetched entry (x) is stored later (delayed update)
    sleep($T_{ft}$);
    n.periodicUpdate(i + 1, x);
  }
}
```

Fig. 5 The periodic finger table update algorithm.

request updates its BFT), PU2 is not used.

Figure 4 is an example of a message sequence. The algorithm is shown in Fig. 5. When initial finger table updates are completed, `periodicUpdate(0, null)` is executed after waiting T_{ft} to start periodic finger table updates.

This algorithm is essentially the same to that of Chord$^{\#}$ with regard to FFT, so when no node insertion and deletion occurs, an FFT[i] of any node converges so that it points to the 2^i distant node in the clockwise direction. Likewise, a BFT[i] of any node converges so that it points to the 2^i distant node in the counterclockwise direction because PU1 ensures that when p.FFT[i] points to q, q.BFT[i] points to p.

3.4 Managing Reverse Pointers

When node p actively updates its finger table and obtains a pointer to q, q also obtains p in its finger table using PU1. Therefore, pointers obtained using active update and PU1 are added to the reverse pointer set.

As for PU2, a node s, which is pointed to by p.FFT[i], is pointed to by q.FFT[$i + 1$], where q is a node pointed to by p.BFT[i]. Therefore, q is added to s's reverse pointer set. This is done as follows: before p sends a getEnt request to q, p sends another getEnt request to s (to obtain r.FFT[i]). This request sent to s is used for adding q to s's reverse pointer set (cf. "$R = R \cup \{N0, N6\}$" at N2 in Fig. 2, where $s = N2$ and $q = N6$).

When an FTE of p is actively or passively updated from x to y and if p has no more FTEs that points to x, p sends a *removeRev* request to x to remove p from $x.R$.

3.5 GetEnt Received by Inserting Node

Let us consider the case where a node p sends a getEnt request to another node q to fetch q.FFT[i] (same discussion can be applied to BFT). If q has not finished its initial finger table updates, q.FFT[i] may be not initialized (null). In this case, p resends the getEnt request after a defined amount of time. (This procedure is omitted in Figs. 3 and 5.)

Note that this scheme is not susceptible to livelock. As the level 0 ring is managed by a ring management algorithm, a getEnt request to fetch FFT[0] always succeeds and thus FFT[1] of any node is eventually initialized. This in turn enables FFT[2] and subsequent FFTs of any node to eventually be initialized.

3.6 Node Deletion

When node p is being deleted, the ring management algorithm first removes p from the level 0 ring. Then, FTEs of other nodes that point to p are changed to p's predecessor q. This is done as follows: p sends p's reverse pointer set ($p.R$) to q. q sends a *deletion notification* message to each node in $p.R$, which requests the recipient node to change all the FTEs that points to p to q. Also, q merges $p.R$ into $q.R$. As p may receive messages for a while after p is removed from the level 0 ring, p continues its routing task for a while after its deletion.

3.7 Lookup

Suzaku uses greedy routing in the clockwise direction to look up a target node. When node p looks up a node whose key is equal to the given k, p sends a query message to x, which is a node in p's finger table whose key is closest to k in the clockwise direction. This procedure is repeated by query recipient nodes until the query reaches the destination node.

As each node has finger tables for both directions, the upper bound of lookup hops is $\lceil \log_2 n \rceil - 1$ when finger tables of all nodes are converged[†]. Considering unconverged states, the number of lookup hops peaks when a lot of nodes are

[†]Each FFT and BFT covers half of the node space and the fingers in the tables point to exponentially distant nodes. Thus, the upper bound of lookup hops is $\lceil \log_2 (n/2) \rceil = \lceil \log_2 n \rceil - 1$.

inserted in a short time and the newly inserted nodes have yet to start periodical finger table updates (PFTUs). The lookup performance in such situations are evaluated with simulations in Sects. 4.2 and 4.3.

3.8 Handling Failures

When a node fails, both the level 0 ring and finger tables need be repaired. The former is handled by the ring management algorithm. We describe how to address the latter.

In Suzaku, an FTE that points to node x also contains x's successor list as its *backup pointers*. Backup pointers are directly obtained from x by finger table updates (with the reply messages of getEnt). When node p sends a message to node q, which is taken from an FTE e, and detects q's failure (by response timeout, etc.), p picks up an alternative node q' from the backup pointers of e and resend the message to q'. (q' provides its successor list to p to refill backup pointers.)

4. Evaluation and Discussion

In this section, we evaluate Suzaku by comparing it with Chord$^{\#}$ and Skip Graph (We choose Chord$^{\#}$ as a comparison target among periodic KOPSONs, because Suzaku is based on Chord$^{\#}$ and thus its comparison is essentially important).

4.1 Simulator

We implemented a discrete, message level event simulator that simulates Suzaku, Chord$^{\#}$, and Skip Graph. All algorithms use DDLL [13] as the ring management algorithm.

In the simulations, T_{ft}, the finger table update period of Chord$^{\#}$ and Suzaku, is 1 minute. All end-to-end latency is set as 20ms regardless of message size[†].

In the Chord$^{\#}$ implementation, we applied the following improvements for performance. On node insertion, a node copies the finger tables of the predecessor node. On node failure, finger tables are recovered using the same algorithm of Suzaku.

In the Skip Graph implementation, binary MV is used. The implementation has several differences from the original paper [7]. (1) In the original version, linked lists at each level are not circular, but ours are circular for fair comparison. (2) To simplify the evaluation, insertion to or deletion from linked lists at level 1 or above is instantly completed and does not conflict with the insertion or deletion of other nodes. (3) To simplify the evaluation, routing tables are instantly repaired when a failure is detected. Note that these modifications are advantageous for Skip Graph.

All experiments started after the first node (initial node) was inserted. All other nodes are inserted in random order.

[†]End-to-end latency is small compared with T_{ft} and does not affect much on the simulation results. 20ms (40ms in RTT) is typical value used in several Internet research (such as [14], [15]). Also, all messages used in the experiments are control messages so their sizes do not vary much.

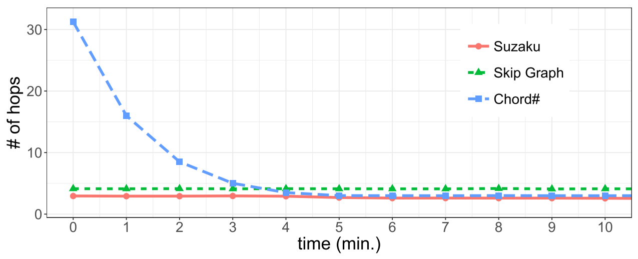
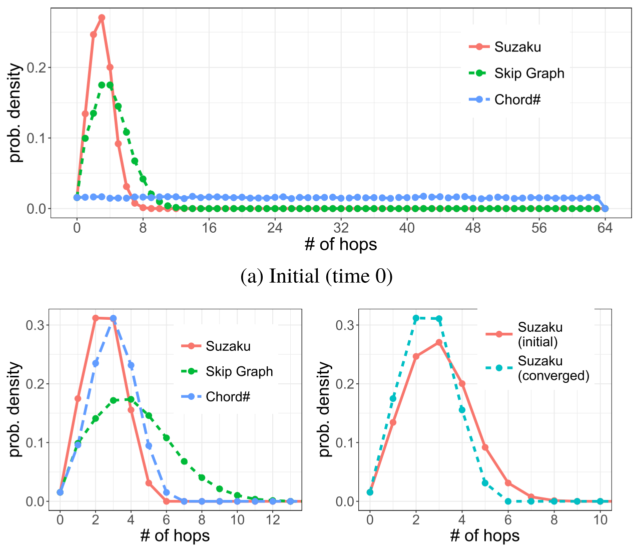


Fig. 6 Average number of lookup hops for elapsed time. Number of nodes is 64.



(a) Initial (time 0)

(b) Converged (time 64)

(c) Comparison (Suzaku)

Fig. 7 (a) and (b): Probability distribution of number of lookup hops at initial state (a) and at converged state (b). (c): Comparison between the initial state and the converged state of Suzaku.

4.2 Number of Hops in Massive Node Insertion (1)

For Suzaku, the worst situation for lookup performance is when a lot of nodes are inserted in a short time and no node has started periodic finger table updates. Thus, we evaluated the following case. We inserted 64 nodes in a short time. After insertion finished, we executed 4000 lookups between two random nodes every 60 seconds. We repeated this experiment 10 times and merged the results.

Figure 6 shows the average number of lookup hops from time 0 (when all nodes were inserted) to time 10 (min). Figures 7(a) and 7(b) show the probability distributions of the number of lookup hops at time 0 (initial state) and time 64 (when the routing tables of all Chord$^{\#}$ or Suzaku nodes were converged). Figure 7(c) shows the difference between hops distribution of Suzaku at time 0 and time 64.

In Chord$^{\#}$, the number of lookup hops at time 0 is large and the average number of hops is $O(n)$ because no node has updated its finger table. The lookup performance of Skip Graph does not change because it does not update its routing tables. In Suzaku, the number of lookup hops is small (average 2.95) from the time of insertion, and it is close to the converged value (2.50). According to Fig. 7(a), the lookup performance of Suzaku seems to exceeds Skip Graph and Chord# even without any PFTUs. This is true for Chord# but for Skip Graph, it depends on the number of nodes. We discuss this in the next subsection.

When finger tables are converged (Fig. 7(b)), the maximum number of hops of Suzaku is settled to $\lceil \log_2 n \rceil - 1$ (5 in this case). Figure 7(c) indicates that the number of lookup hops at the initial state is very close to the converged one.

4.3 Number of Hops in Massive Node Insertion (2)

To examine the churn resiliency of Suzaku in more detail, we evaluated the lookup performance, by varying the number of nodes. As we were interested in the worst case performance, PFTUs were turned off in this experiment.

We also measured how PU2 contributed to lookup performance. We compared original Suzaku (Suzaku) with Skip Graph, Suzaku without PU2 (Suzaku-nopu2), and Suzaku with bi-directional PU2 (Suzaku-pu2bid). Suzaku-pu2bid updates both FFT and BFT with PU2 whereas the normal PU2 updates only FFT (uni-directional)[†]. We omit Chord# because it requires $O(n)$ hops when PFTU is turned off.

The results shown in Fig. 8 are the average of 10 trials. Among Suzaku and its variants, Suzaku-pu2bid achieved the best performance, followed by normal Suzaku and Suzaku-nopu2. The average numbers of lookup hops of Suzaku-pu2bid, Suzaku, and Suzaku-nopu2 at $n = 32$ Ki were around 10.2, 11.2 and 12.7, respectively.

While the trend line of Skip Graph is straight because its average number of lookup hops is $O(\log n)$ [7], the lines of Suzaku and its variants slightly curve upwards. This implies that the average number of lookup hops of Suzaku and its variants (without PFTU) are larger than $O(\log n)$. However, if the number of nodes is fewer than around 100k, which is sufficient for many practical cases, Suzaku achieves *nearly logarithmic* lookup hops and is faster than Skip Graph, even if PFTU is turned off.

These results conclude that Suzaku is highly resilient against massive node insertion.

Next, we discuss the rationale for the number of lookup hops in this situation.

Figure 9 shows the average clockwise distance (in number of nodes) of FFT entries for each level when PFTU is turned off. (When computing the average in a network of n nodes, a node that lacks FFT[i] is treated as if it has a pointer of distance n.)

As successor pointers of each node are actively maintained by the ring maintenance algorithm, the distance of all pointers in FFT[0] is 1 (not shown in the figure). The distance of all pointers in FFT[1] is 2. This can be explained as follows. Let us assume node p is inserted, p.FFT[0] points to node q, and p.FFT[1] points to node r. At this point, the distance from p to r is 2. If the distance of p.FFT[1] changes later, another node x must be inserted between p and r. If x is inserted between q and r, x updates p.FFT[1] to x with PU1. If x is inserted between p and q, x updates p.FFT[1] to q with PU2. In either case, the distance of p.FFT[1] does not change. (Note that we excluded node deletion cases in this discussion. If a node deletion occurs, the distance of FFT[1] may drop to 1).

[†]Suzaku-pu2bid requires an additional message to update a BFT entry of a remote node. For example, in Fig. 3, N0 sends a *BFT-update request* to N1 to update N1.BFT[1] to N7 after receiving a reply message from N7 (which confirms that N7 is alive).

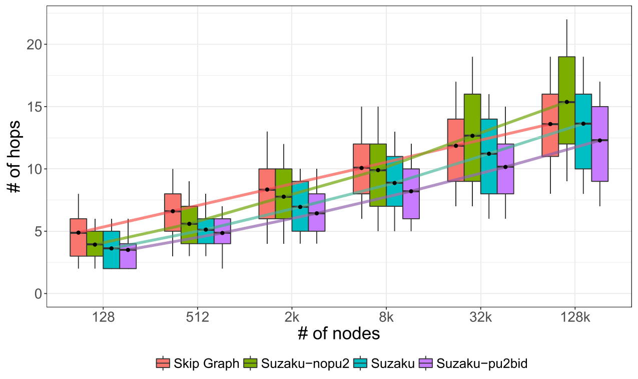


Fig. 8 Distribution of number of lookup hops for various numbers of nodes when periodical finger table updates are turned off. Suzaku, Suzaku without PU2, Suzaku with bi-directional PU2, and Skip Graph were evaluated. Boxes with whiskers represents the distribution of the number of hops. The bottom and upper edges of a box represent the 25th and 75th percentiles, respectively. The bottom and upper ends of the whiskers represent the 10th and 90th percentiles, respectively. The band inside a box is the average. The average points are connected with lines to observe the trend line.

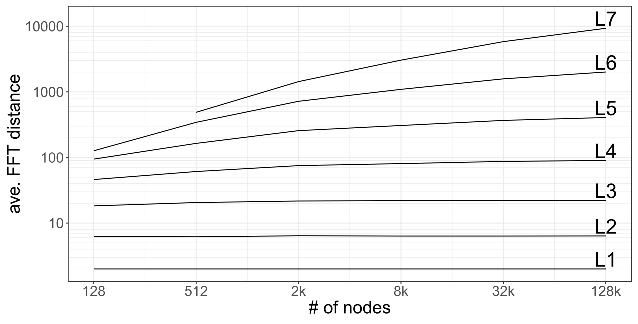


Fig. 9 Average distance (in number of nodes) of FFT entries for each level when periodic finger table updates are turned off. Both axes are in a log scale. Levels over L7 are omitted. When $n = 128$, no node has a L7 entry.

Above level 1, the average distance of each level gradually increases as the number of nodes increases, but it seems to converge to a certain distance; at $n = 128$ Ki, the approximate average distances of FFT[2] to FFT[7] are 6.37, 22.3, 90.0, 406.2, 2004.0, and 9306.0, respectively. The distance intervals between adjacent levels increase slightly larger than exponential. This observation agrees with the fact that the average lookup hops of Suzaku without PFTU is *nearly logarithmic* but larger than $O(\log n)$.

On the basis of the above results, the necessity of PFTU is questioned. There are several reasons to use PFTU aside from optimizing lookup hops.

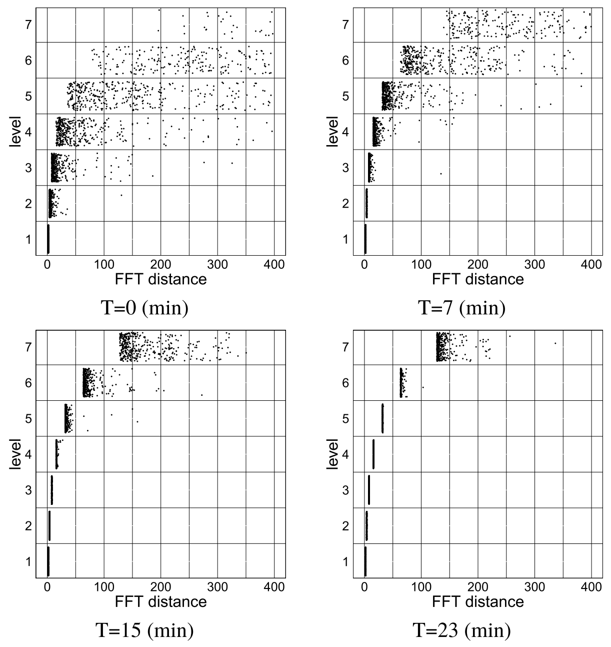


Fig. 10 Distributions of distances of FFT entries for each level. The x-axis is distance (in number of nodes) of FFT entries. The y-axis is the FFT level. Each dot corresponds to a single FFT entry. (a) Distributions immediately after 400 nodes are inserted. (b), (c), and (d) distributions after 7, 15, and 23 minutes, respectively, to observe convergence.

- As described in Sect. 3.8, an FTE has backup pointers. PFTU is useful for keeping backup pointers up-to-date.
- Let us consider the case where successive nodes $N1, N2, \ldots, N10$ are deleted. In this case, $N0$ is more pointed to by other nodes because the node deletion algorithm updates all the FTEs of nodes that point to $N1 \ldots N10$ to $N0$. (The node deletion algorithm updates all pointers to a deleted node to its predecessor node). PFTU effectively eliminates such imbalance.

4.4 Distribution and Convergence of FFT Entries

We inserted 400 nodes in a short time and observed the distance changes of FFT entries (Fig. 10). We observed that the variance of distribution is initially large especially at higher levels (Fig. 10(a)). (b), (c), and (d) respectively correspond to the result after the `periodicUpdate` procedure is executed 7, 15, and 23 times at each node. The distance of FFT[i] converges to 2^i as time elapses.

4.5 Retransmission Rate in Massive Node Deletion

To examine the behavior when a lot of nodes are deleted in a short time, we conducted the following experiment. We inserted 256 nodes (N0 to N255) and deleted N32 to N96 immediately after finger tables were converged. In parallel to deletion, we performed queries from a random node within N0 to N31 to a random node within N97 to N127. We measured the retransmission ratio, which is a ratio of how many queries were sent to a deleted node, and retransmission

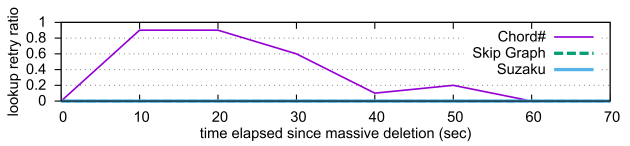


Fig. 11 Lookup retransmission ratio in massive node deletion. The x-axis represents elapsed time since node deletion was completed. Ratio is computed by aggregating lookup failure over 10-second periods.

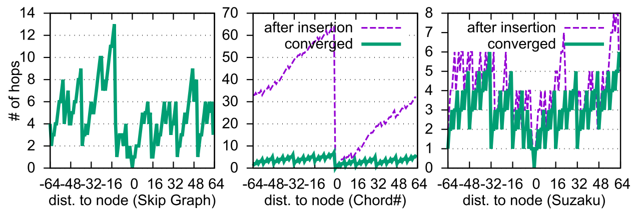


Fig. 12 Relation between distance to destination node and number of lookup hops. The x-axis represents the number of nodes from N64 to the destination node (N0 to N127).

was performed.

Figure 11 shows the results. While Chord# required frequent retransmissions (because it does not remove pointers to a deleted node), Suzaku and Skip Graph did not require any retransmissions. These results show that Suzaku (and Skip Graph) are resilient to massive node deletion.

4.6 Relation Between Location of Lookup Target and Number of Lookup Hops

Some KOPSON applications such as pub/sub systems require that routing to close nodes is fast regardless of its directions (bigger or smaller keys). Suzaku conforms to this property. To examine the performance of looking up close nodes, we inserted 127 nodes (N0 to N63 and N65 to N127) and then inserted N64. We measured the number of hops from N64 to all other nodes. Results are shown in Fig. 12.

We observed that immediately after a massive node insertion, Chord# requires a significant amount of time to look up close nodes in the counterclockwise direction (requires $O(n)$ hops), and Suzaku is fast regardless of lookup directions without finger table updates.

4.7 Height of Finger Tables

We evaluated the average height of finger tables of Suzaku, both in the state of after massive node insertion (where PFTU is off) and in the converged state (Fig. 13). The average height is, $\lceil \log_2 n \rceil$ in the converged state, and is lower than $\lceil \log_2 n \rceil$ in the massive node insertion case. This result indicates that the average finger table height of Suzaku is $O(\log n)$.

4.8 Node Insertion Traffic

To measure traffic required for node insertion, we conducted the following experiment. After inserting the initial node,

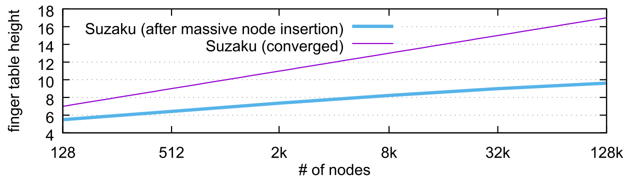


Fig. 13 Finger table height of Suzaku, both in the state of after massive node insertion (no PFTU) and in the converged state. The y-axis is the average of max(FFT height, BFT height).

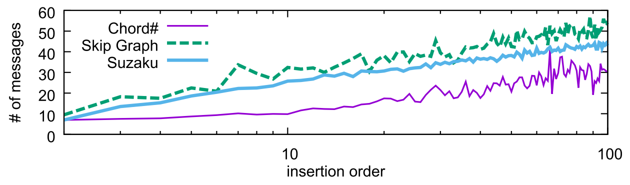


Fig. 14 Number of messages required for node insertion as a function of insertion order.

we inserted 100 nodes and counted the number of messages required for inserting the i'th node. We repeated the experiment 10 times to compute the average number.

Results are shown in Fig. 14. According to the graph, Suzaku requires fewer messages than Skip Graph. As Chord# does not construct finger tables on node insertion, it requires less messages than Skip Graph and Suzaku. The number of messages of Chord# increases as i increases because it required more messages for finding node insertion locations. If nodes are slowly inserted, fewer messages are required as their finger tables will be (at least partially) updated.

Node insertion requires three types of messages; messages for finding insertion location (m_1), those for inserting to the level 0 ring (m_2) and those for constructing routing tables (m_3). As many nodes are inserted in a short time in this experiment, m_1 is $O(n)$ for Chord#, $O(\log n)$ for Skip Graph, and a *nearly logarithmic number* against n for Suzaku (n denotes the number of nodes). m_2 is 3 in DDLL. m_3 is 2 for Chord# (for copying a finger table from the predecessor node) and $O(\log n)$ for both Suzaku and Skip Graph because it is in proportion to the height of the routing tables. In Suzaku, updating a single level of finger tables requires 7 messages at the most (2 for getEnt, 2 for replies, 3 for removeRev). Skip Graph requires 6 messages (2 on average for reaching the MV-matching node, 1 for the reply, and 3 for inserting to the linked list, assuming DDLL is used).

In summary, node insertion in Suzaku requires a *nearly logarithmic number* of messages even in the worst case.

4.9 Node Deletion Traffic

Node deletion requires deletion from the level 0 ring and sending deletion notification messages to each node in the reverse pointer set. The former requires 3 messages in DDLL. The latter depends on the size of reverse pointer set. The size is usually bounded by the sum of the height of FFT and BFT. It is temporarily increased by $O(\log n)$ when the successor node is deleted (see Sect. 3.6), but as time elapses, the size is restored. Therefore, we can say that Suzaku requires $O(\log n)$ messages for node deletion.

Table 1 Comparison matrix.

	Skip Graph	Chord#	Suzaku
Avg. # of lookup hops (after massive insertion)	$O(\log n)$	$O(n)$	larger than $O(\log n)$ but *nearly logarithmically* increased and smaller than Skip Graph when $n \leq$ around 100k
Avg. # of lookup hops (converged)	$O(\log n)$	$\leq \lceil \log_2 n \rceil$	$\leq \lceil \log_2 n \rceil - 1$
Max. # of lookup hops (converged)	severalfold of $\log_2 n$	$\lceil \log_2 n \rceil$	$\lceil \log_2 n \rceil - 1$
Lookup close nodes in the counterclockwise direction	fast	slow	fast
Routing to deleted nodes	no	yes	no
Implementation complexity	complex	simple	simple

4.10 Permanent Traffic

Both Chord# and Suzaku update their finger tables periodically (every T_{ft}). Chord# requires 2 messages (getEnt and its reply) for single entry updates but Suzaku may require one additional message (removeRev) when an entry is modified.

However, while Chord# requires short T_{ft} to reduce the number of hops in the case of massive node insertion, it is not necessary for Suzaku (see Sect. 4.3). Therefore, Suzaku requires less frequent finger table updates than Chord#. As for Skip Graph, while it does not require periodical routing table updates, it still requires the sending of periodical keep-alive messages to neighbor nodes of each level to maintain connectivity with other nodes. Quantitative comparison of permanent traffic is the focus of future studies.

The literature of Chord#[6] proposes an idea of piggybacking FTE information when sending search results to improve routing table quality. This technique can also be applied to Suzaku.

4.11 Implementation Complexity

In comparison with Chord#, Suzaku includes the algorithms of initial finger table updates, passive updates, and reverse pointers. As these algorithms are not complex, the implementation complexity of Suzaku is not much different from Chord#. Suzaku is less complex than Skip Graph because it does not need to manage the consistency among multiple distributed doubly linked lists.

4.12 Comparison

Table 1 shows the comparison matrix among Chord#, Skip Graph, and Suzaku.

5. Conclusion

We presented Suzaku, a novel KOPSON that combines the strengths of Chord$^{\#}$ (fast lookup when routing tables are converged and low implementation complexity) and those of Skip Graphs (no performance degradation in churn and fast lookup of close nodes in both directions). Suzaku is highly scalable; as the number of nodes increases, both the number of lookup hops and the number of messages required for node insertion/deletion increase *nearly logarithmically* at most even in the worst case (where all nodes are inserted in a short time) for practical number of nodes, and it requires less frequent finger table updates. Although not described in this paper, like Chord$^{\#}$ and Skip Graph, Suzaku efficiently supports range queries. Given these properties and features, Suzaku is useful for various applications.

We have implemented Suzaku in our P2P network platform PIAX [16] and replaced Skip Graph with Suzaku as its main overlay network.

Our future work includes conducting more detailed evaluations and computing the theoretical bound of the maximum number of lookup hops.

Acknowledgments

This work was in part supported by JSPS KAKENHI Grant Number JP16K00135.

References

[1] A. Yahyavi and B. Kemme, “Peer-to-peer architectures for massively multiplayer online games: A survey,” ACM Comput. Surv., vol.46, no.9, pp.1–51, 2013.
[2] R. Banno, S. Takeuchi, M. Takemoto, T. Kawano, T. Kambayashi, and M. Matsuo, “Designing overlay networks for handling exhaust data in a distributed topic-based pub/sub architecture,” J. Information Processing, vol.23, no.2, pp.105–116, 2015.
[3] X. Shao, M. Jibiki, Y. Teranishi, and N. Nishinaga, “Effective load balancing mechanism for heterogeneous range queriable cloud storage,” Proc. CloudCom 2015, pp.1–8, 2015.
[4] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup protocol for internet applications,” IEEE/ACM Trans. Netw., vol.11, no.1, pp.17–32, 2003.
[5] A.R. Bharambe, M. Agrawal, and S. Seshan, “Mercury: Supporting scalable multi-attribute range queries,” Proc. SIGCOMM 2004, pp.353–366, ACM, 2004.
[6] T. Schütt, F. Schintke, and A. Reinefeld, “Range queries on structured overlay networks,” Comput. Commun., vol.31, no.2, pp.280–291, 2008.
[7] J. Aspnes and G. Shah, “Skip graphs,” ACM Trans. Algorithms, vol.3, no.4, pp.1–25, 2007.
[8] T. Kawaguchi, R. Banno, M. Hojo, and K. Shudo, “Self-refining skip graph: A structured overlay approaching to ideal skip graph,” Proc. COMPSAC 2016, pp.377–378, 2016.
[9] M. Hojo, R. Banno, and K. Shudo, “FRT-skip graph: A skip graph-style structured overlay based on flexible routing tables,” Proc. IEEE Intl. Symposium on Computers and Communication 2016, pp.657–662, IEEE, 2016.
[10] M.T. Goodrich, M.J. Nelson, and J.Z. Sun, “The rainbow skip graph: A fault-tolerant constant-degree distributed data structure,” Proc. 17th annual ACM-SIAM symposium on Discrete algorithm, pp.384–393, 2006.
[11] A. González-Beltrán, P. Milligan, and P. Sage, “Range queries over skip tree graphs,” Comput. Commun., vol.31, no.2, pp.358–374, 2008.
[12] A. Singh and S. Batra, “P-skip graph: An efficient data structure for peer-to-peer network,” in Intelligent Distributed Computing, pp.43–54, Springer, 2015.
[13] K. Abe and M. Yoshida, “Constructing distributed doubly linked lists without distributed locking,” Proc. IEEE Intl. Conf. on P2P Computing 2015, pp.1–10, 2015.
[14] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, “Recursively cautious congestion control,” Proc. 11th USENIX Conference on Networked Systems Design and Implementation, NSDI’14, pp.373–385, Berkeley, CA, USA, USENIX Association, 2014.
[15] N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh, and V. Jacobson, “BBR: Congestion-based congestion control,” Commun. ACM, vol.60, no.2, pp.58–66, 2017.
[16] Y. Teranishi, “PIAX: Toward a framework for sensor overlay network,” Proc. CCNC’09, pp.1–5, 2009.

Kota Abe received his M.E. and Ph.D. degrees from Osaka University, Japan, in 1994 and 2000, respectively. He was an engineer at Nippon Telegraph and Telephone Corporation (NTT), Japan, from 1994 to 1996. He was a research associate at Media Center, Osaka City University, Japan, from 1996 to 2000 and a lecturer from 2000 to 2003. He was a lecturer of Graduate School for Creative Cities, Osaka City University from 2003 to 2005, an associate professor from 2005 to 2012, and a professor since 2012. Since 2018, he is a professor of Graduate School of Engineering, Osaka City University. He received IPSJ Best Paper Award in 2013. His research interests include distributed systems and system software. He is a member of the IPSJ, IEICE and IEEE.

Yuuichi Teranishi received his M.E. and Ph.D. degrees from Osaka University, Japan, in 1995 and 2004, respectively. From 1995 to 2004, he was engaged Nippon Telegraph and Telephone Corporation (NTT). From 2005 to 2007, he was a Lecturer of Cybermedia Center, Osaka University. From 2007 to 2011, he was an associate professor of Graduate School of Information Science and Technology, Osaka University. Since August 2011, He has been a research manager and project manager of National Institute of Information and Communications Technology (NICT). He received IPSJ Best Paper Award in 2011. His research interests include technologies for distributed network systems and applications. He is a member of the IPSJ, IEICE and IEEE.