PAPER

Broadcast with Tree Selection from Multiple Spanning Trees on an Overlay Network

Takeshi KANEKO[†*], *Nonmember and* **Kazuyuki SHUDO**[†a)], *Member*

SUMMARY On an overlay network where a number of nodes work autonomously in a decentralized way, the efficiency of broadcasts has a significant impact on the performance of distributed systems built on the network. While a broadcast method using a spanning tree produces a small number of messages, the routing path lengths are prone to be relatively large. Moreover, when multiple nodes can be source nodes, inefficient broadcasts often occur because the efficient tree topology differs for each node. To address this problem, we propose a novel protocol in which a source node selects an efficient tree from multiple spanning trees when broadcasting. Our method shortens routing paths while maintaining a small number of messages. We examined path lengths and the number of messages for broadcasts on various topologies. As a result, especially for a random graph, our proposed method shortened path lengths by approximately 28% compared with a method using a spanning tree, with almost the same number of messages.

key words: *broadcast, overlay network, path length, Plumtree, spanning tree*

1. Introduction

An overlay network is an application-level logical network built on an existing network such as the Internet. It is applied to various distributed systems: e.g. distributed key/value stores [1], video streaming [2], and online games [3]. In recent years, application to the fields of IoT and blockchain is also expected [4].

On an overlay network where a number of nodes work autonomously in a decentralized way, the large-scale distributed system requests frequent information exchange with low latency to achieve high scalability and high reliability. Therefore, the efficiency of broadcasts has a significant impact on the performance of the distributed system.

A variety of methods to realize efficient broadcasts have been proposed [5]. In particular, while tree-based methods using a spanning tree built on the network produce a small number of messages, the routing path lengths are prone to be larger than those of flooding and gossip-based methods. Moreover, when multiple nodes share a single spanning tree and issue broadcasts on it, inefficient broadcasts often occur because the appropriate topology of the tree differs for each node.

To address this problem, we propose a novel protocol in which a source node selects an efficient tree from multiple spanning trees when broadcasting. This method reduces the frequency of inefficient broadcasts for multiple source nodes, even if most participating nodes can be source nodes. It thereby shortens routing path lengths while maintaining the advantage of tree-based methods, which is producing a small number of messages. We adopt Plumtree [6] as the spanning tree construction protocol though the proposed tree selection does not depend on a specific tree construction protocol.

This paper is an extended version of our previous work [7]. This paper presents experiments conducted using `Torus Grid Graph`; the results are given in Fig. 9, Fig. 10 and Fig. 11 in Sect. 6.2. Pseudocode for data propagation, tree repair and optimization, and membership management are also added as Algorithm 4, Algorithm 5 and Algorithm 6 in Sect. 4.3. An analysis of the proposed method shown in Sect. 5 is also part of the difference.

The remainder of this paper is organized as follows. Section 2 presents the related work on broadcasts using spanning trees. Section 3 defines notations used in this paper. Section 4 presents the proposed method in detail. Section 6 presents the evaluation experiments for the proposed method and the results. Finally, Sect. 7 presents the conclusion of this study.

2. Related Work

In a gossip-based broadcast [8], each node randomly selects f nodes from the network for a system parameter f called fanout, and sends messages to them. While this provides high scalability and high reliability, it has the disadvantage of producing a large number of messages. In a tree-based broadcast, a spanning tree is built on the network and each node sends messages on the tree topology. While this maintains a small number of messages, it provides low fault tolerance and low reliability.

2.1 Plumtree

Leitao et al. proposed a broadcast method called Plumtree [6], which combines a gossip protocol and a tree-based protocol. This achieves both a small number of messages and high reliability. In addition, the routing path lengths of each broadcast are relatively small for multiple source nodes due to the optimization process of the tree topology. However, as with other tree-based broadcasts, the lengths are still prone to

Manuscript received January 24, 2022.
Manuscript revised April 20, 2022.
Manuscript publicized August 16, 2022.
[†]The authors are with the School of Computing, Tokyo Institute of Technology, Tokyo, 152-8552 Japan.
[*]Presently, with Yahoo Japan Corporation.
a) E-mail: shudo@media.kyoto-u.ac.jp
DOI: 10.1587/transcom.2022EBP3007

be larger than those of flooding because broadcast messages are sent only on a single spanning tree topology.

2.2 Methods Using Multiple Spanning Trees

SplitStream [9] and Chunkyspread [10] are multicast methods that construct multiple spanning trees on a network. Because the purpose is streaming, the content is divided into some strips and they are sent with all trees simultaneously; the load is thereby balanced for each tree. On the other hand, the proposed method selects an appropriate tree instead of multiple trees and shortens the routing paths. SplitStream also differs from the proposed method in that it constructs spanning trees on a structured overlay Pastry [11], rather than an unstructured overlay. On a structured overlay, a tree is constructed by unicast routing supported by the overlay. Participating nodes route to a specific ID and the resulted paths constitute a tree [12]. In contrast, an unstructured overlay does not support unicast routing. A tree is constructed by flooding. A source node floods a tree construction message to all the nodes. A receiver establishes a tree branch to the sender when it receives the message for the first time. Thereafter the receiver ignores the message from other senders [6].

Thicket [13] is a multicast method that constructs and manages multiple spanning trees on an unstructured overlay. As with other existing methods, the purpose of using multiple spanning trees is load balancing. On the other hand, the proposed method selects one tree from multiple spanning trees when broadcasting to shorten the routing paths. It is possible to partially apply the proposed method to Thicket because the idea for the tree selection is simple.

2.3 Mobile ad hoc Network

There have been techniques constructing a tree for multicast and broadcast in mobile ad hoc networks (MANETs) [14], [15]. Utilizing multiple trees simultaneously also have been considered for communication quality such as bandwidth [16], [17].

Costs to consider for MANETs and overlay networks are different. In MANET, a node can send a message to all the near nodes at once. But in an overlay network, a node has to send a message to each neighbor node individually. Because of it, the number of messages is an important measure of cost in an overlay network. Our goal is to achieve better performance while keeping the cost on par with a single tree.

2.4 IP Multicast

IP multicast protocols construct a distribution tree of routers for each multicast group, represented by a IP multicast address. All existing IP multicast routing protocols construct a single distribution tree for a multicast group while the proposed method utilizes multiple trees for a single receiver group. There is little room for utilizing multiple trees because a network structure of routers is fixed and then the network has almost no degrees of freedom in a tree topology. Moreover, IP multicast has ever hardly been deployed and then overlay network based techniques such as the proposed method have been exploited.

3. Notation

This section defines notations used in this paper.

An undirected graph G is represented by a pair (V, E), where V denotes a set of vertices and E denotes a set of edges. We define the following for $G = (V, E)$:

- $\mathrm{adj}_G : V \ni v \mapsto$ (the set of all neighbors of v) $\in 2^V$
- $\deg_G : V \ni v \mapsto |\mathrm{adj}_G(v)| \in \mathbb{N}$
- $\mathrm{dist}_G : V \times V \ni (u, v) \mapsto$ (the distance between u and v) $\in \mathbb{N} \cup \{\infty\}$
  - Especially, $^\forall v \in V, \mathrm{dist}_G(v, v) = 0$.

An undirected rooted tree T is represented by a triple (V, F, r); where V denotes a set of vertices, F denotes a set of edges, and r denotes the root node. It can also be regarded simply as an undirected graph $T = (V, F)$. We define the following for $T = (V, F, r)$:

- $\mathrm{parent}_T : V - \{ r \} \ni v \mapsto$ (the parent of v) $\in V$
- $\mathrm{children}_T : V \ni v \mapsto$ (the set of all children of v) $\in 2^V$
- $\mathrm{subtree}_T : V \ni v \mapsto$ (the rooted subtree induced by v)
- $\mathrm{height}_T := \max \{ \mathrm{dist}_T(r, v) \mid v \in V \} \in \mathbb{N}$

4. Proposed Method

In the proposed method, multiple spanning trees are constructed on an overlay network (Sect. 4.1), and a source node selects an appropriate tree from them (Sect. 4.2) and issues a broadcast (Sect. 4.3).

Each spanning tree is embedded in an unstructured overlay network managed by a peer sampling service [18], [19]. A spanning tree is maintained along the Plumtree protocol [6]. In Plumtree, a node has two types of neighbors in *eagerPushPeers* and *lazyPushPeers* sets. *EagerPushPeers* include neighbors along tree edges and *lazyPushPeers* include other neighbors. A broadcast is basically performed over a tree, that is, forwarding to *eagerPushPeers*. A node sometimes forwards a notification of arrival of a message to *lazyPushPeers*. The notified node starts repairing a tree if it has not received the message along the tree. Tree construction is as follows. A node has all its neighbors in its *eagerPushPeers* at the start of the construction. The initial tree is lengthy and there are multiple paths to a single node. If a node receives the same message from multiple neighbors, the node replies a PRUNE message to the sender and both the sending and receiving nodes move each other from *eagerPushPeers* to *lazyPushPeers*.

Since messages for each broadcast are sent on a single tree, the number of messages is approximately $\Theta(n)$ for a stable overlay network where n denotes the number of participating nodes.

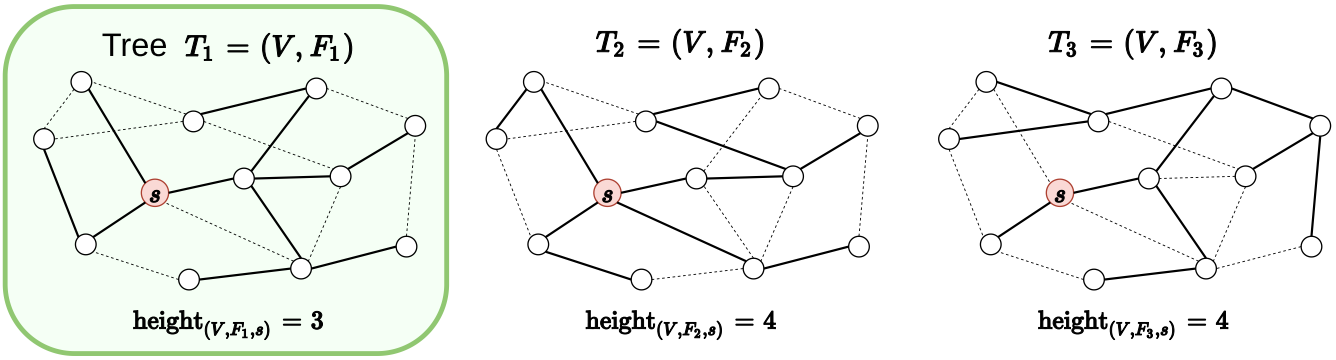
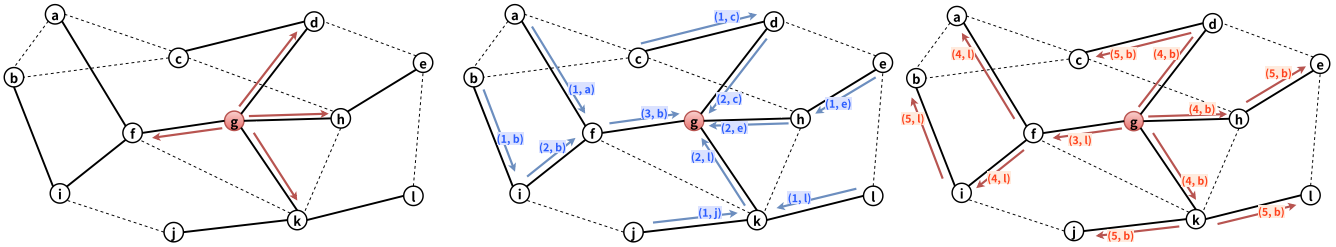


Fig. 1 An example of tree selection from multiple spanning trees for a source node s.



(a) Construction of spanning trees from the root node g (`ConstructTree` in Algorithm. 2).

(b) Propagation of distance information from all the leaf nodes into the root node (`UpdateDistTowardRoot` in Algorithm. 2).

(c) Propagation of distance information from the root node to all the leaf nodes (`UpdateDistTowardLeaves` in Algorithm. 2).

Fig. 2 Tree construction.

Moreover, our aim is to minimize the maximum path length from the source node to each node for each broadcast by the following process:

1. Suppose that a finite number of spanning trees $T_1 = (V, F_1), T_2 = (V, F_2), \ldots, T_k = (V, F_k)$ are managed on a graph $G = (V, E)$[†] of the network topology.
2. A source node $s \in V$ selects the tree T_λ ($\lambda \in \{1, 2, \ldots, k\}$) that minimizes

$$\text{height}_{(V,F_\lambda,s)} = \max \left\{ \text{dist}_{T_\lambda}(s, v) \,\middle|\, v \in V \right\}, \quad (1)$$

which is the maximum path length of each simple path whose one end is the node s on T_λ.

Figure 1 shows an example of tree selection. In this example, the network manages three spanning trees T_1, T_2, and T_3, the source node is a node s, and then $\text{height}_{(V,F_1,s)} = 3$, $\text{height}_{(V,F_2,s)} = 4$, and $\text{height}_{(V,F_3,s)} = 4$. Since T_1 minimizes (1), the source node s selects T_1 from them.

Note that, however, since each node does not have global information on the network, it selects a tree based on a value estimated only from information it knows rather than the exact value of (1).

[†]Strictly speaking, an overlay network with a peer sampling service forms a topology of a directed graph. However, for simplicity, the network topology is regarded as an undirected graph in this paper because each managed spanning tree behaves like an undirected graph.

Algorithm 1: Fields and Initialization in a Node

$v_{current}$

```
1  data structure Tree
2      field eagerPushPeers: Set⟨Node⟩
3      field lazyPushPeers: Set⟨Node⟩
4      field distFor: Map⟨Node, Int⟩
5      field parent: Node
6  upon receiving ⟨INIT⟩ then
7      peers ← getPeers()
8      trees ← new Map⟨Int, Tree⟩
9      receivedMsgs ← new Map⟨Int, Message⟩
10     missing ← ∅
```

4.1 Tree Construction

The proposed method constructs multiple spanning trees on the network at the beginning of the protocol. The number of trees, a positive integer k, is a system parameter determined before running the protocol. We randomly select k start nodes from the network[††] and construct a spanning tree for each start node based on the delivery tree by flooding. The constructed k trees are not uniform. Each k tree is effi-

[††]For example, we can sample some start nodes by doing a random walk from a node and selecting nodes on every certain number of hops. However, because sampling with a simple random walk is biased by the degree of each node, if it is preferred to sample nodes with a uniform distribution, e.g., we should use a random walk using Metropolis-Hastings algorithm [20].

Algorithm 2: Tree Construction in a Node $v_{current}$

```
1  upon receiving ⟨ConstructTree, treeId⟩ from v_sender then
2      if treeId ∉ trees.keys() then
3          t ← new Tree
4          trees[treeId] ← t
5          t.eagerPushPeers ← peers
6          t.parent ← v_sender
7          for v_adj ∈ t.eagerPushPeers do
8              if v_adj = v_sender then
9                  continue
10             send ⟨ConstructTree, treeId⟩ to v_adj
11     else
12         send ⟨UpdateDistTowardRoot, treeId, ∞⟩ to v_sender
13 upon receiving
       ⟨UpdateDistTowardRoot, treeId, distForSender⟩ from v_sender
       then
14     t ← trees[treeId]
15     if distForSender < ∞ then
16         t.distFor[v_sender] ← distForSender
17     else
18         t.eagerPushPeers ← t.eagerPushPeers − {v_sender}
19         t.lazyPushPeers ← t.lazyPushPeers ∪ {v_sender}
20     if t.eagerPushPeers − { t.parent } = t.distFor.keys() then
21         if t.parent = v_current then
               // when v_current is the root node of t
22             send ⟨UpdateDistTowardLeaves, treeId, 0⟩ to
                 v_current
23         else
24             maxDist ←
                 getMaxDistExceptOne(t.distFor, t.parent)
25             send
                 ⟨UpdateDistTowardRoot, treeId, maxDist + 1⟩
                 to t.parent
26 upon receiving
       ⟨UpdateDistTowardLeaves, treeId, distForSender⟩ from
       v_sender then
27     t ← trees[treeId]
28     if v_sender ≠ v_current then
29         t.distFor[t.sender] ← distForSender
30     for v_adj ∈ t.eagerPushPeers do
31         if v_adj = v_sender then
32             continue
33         maxDist ← getMaxDistExceptOne(t.distFor, v_adj)
34         send
             ⟨UpdateDistTowardLeaves, treeId, maxDist + 1⟩
             to v_adj
35 function getMaxDistExceptOne(distFor, v_excepted)
36     maxDist ← 0
37     for (v_adj, d) ∈ distFor do
38         if v_adj = v_excepted then
39             continue
40         maxDist ← max{maxDist, d}
41     return maxDist
```

cient for the corresponding k start node because a spanning tree construction protocol including Plumtree constructs an efficient tree for the start node.

Algorithm 1 and Algorithm 2 are pseudocodes on initialization and spanning tree construction, respectively. For simplicity, the latter pseudocode assumes that no messages are lost and no nodes join or leave during the construction of a spanning tree. In practice, it is necessary to address such situations by using timeout timers.

Figure 2 illustrates the tree construction process described in Algorithm 2. Figure 2(a) shows construction of spanning trees corresponding to `ConstructTree`, Fig. 2(b) shows propagation of distance information into the root node corresponding to `UpdateDistTowardRoot`, and Fig. 2(c) shows propagation of distance information to all the leaf nodes corresponding to `UpdateDistTowardLeaves`.

Each start node v_{start} constructs a new spanning tree by issuing an event $\langle$`ConstructTree`$, treeId_{new}\rangle$ (line 1 in Algorithm 2) to itself v_{start}. Herein, we generate $treeId_{new}$, the ID corresponding to the tree to be constructed, e.g., by a pseudorandom number to avoid duplicate IDs.

When each node $v_{current}$ receives an event $\langle$`ConstructTree`$, treeId\rangle$, if it does not have the corresponding tree data, then it generates a new tree t and initializes $t.eagerPushPeers$ with its neighbor nodes and sends the same event $\langle$`ConstructTree`$, treeId\rangle$ to each of the nodes. If it has the corresponding tree data, then it sends an event $\langle$UpdateDistTowardRoot$, treeId, \infty\rangle$ (line 13 in Algorithm 2) to the sender to indicate that $v_{current}$ has already received the event for the $treeId$. The sender that receives it excludes $v_{current}$ from $t.eagerPushPeers$. Through these processes, if each node is considered to have an edge with each node in $t.eagerPushPeers$, a spanning tree forms on the network where the start node v_{start} is the root node.

A node that sent an event $\langle$`ConstructTree`$, treeId\rangle$ to its neighbor nodes, after receiving an event $\langle$UpdateDistTowardRoot$, treeId, distForSender\rangle$ from each of them, sends an event $\langle$UpdateDistTowardRoot$, treeId, maxDist + 1\rangle$ to the parent node $t.parent$ where $maxDist$ is

$$maxDist = \max\left(\begin{Bmatrix}\{\,0\,\}\cup \\ \{\, t.distFor[v] \mid \\ \quad v \in t.eagerPushPeers - \{\, t.parent\,\}\end{Bmatrix}\right). \tag{2}$$

In general, the following holds for any node $v \in V$ of a rooted tree $T = (V, F, v_{start})$:

$$\mathrm{height}_{\mathrm{subtree}_T(v)} = \max\left(\begin{Bmatrix}\{\,0\,\}\cup \\ \{\, \mathrm{height}_{\mathrm{subtree}_T(u)} + 1 \mid u \in \mathrm{children}_T(v)\,\}\end{Bmatrix}\right). \tag{3}$$

Thus, this process means computing the height of the tree by dynamic programming through the propagation of "distance" information into the root node v_{start}. Therefore, Each $t.distFor[v_{adj}]$ represents $(\mathrm{height}_{\mathrm{subtree}_T(v_{adj})} + 1)$.

After completing the propagation of "distance" information into the root node, "distance" information propagates into leaf nodes such that each node sends an event $\langle$UpdateDistTowardLeaves$, treeId, distForSender\rangle$

Algorithm 3: A Broadcast with Tree Selection in a Node $v_{current}$

```
1  upon receiving ⟨BROADCAST, msg⟩ from v_sender then
2      msgId ← hash(msg ‖ v_current)
3      treeId ← selectTree()
4      send ⟨GOSSIP, msg, msgId, treeId, 0, 0⟩ to v_current
5  function selectTree()
6      minTreeId ← ∞
7      minDist ← ∞
8      for (treeId, t) ∈ trees do
9          distToEnd ← max({ 0 } ∪ t.distFor.values())
10         if distToEnd < minDist then
11             minDist ← distToEnd
12             minTreeId ← treeId
13     return minTreeId
```

(line 26 in Algorithm 2) to each of the children. When completing the propagation into leaf nodes, each $t.distFor(v_{adj})$ represents $(\text{height}_{\text{subtree}_{(V,F,v_{current})}(v_{adj})} + 1)$.

Through the above processes, a spanning tree is constructed on the overlay network. In a one-and-a-half round-trip broadcast by the start node v_{start}, every $t.distFor$ is computed by dynamic programming with piggyback of the "distance" information.

4.2 Tree Selection

A source node $v_{current}$ broadcasts by issuing an event $\langle \text{BROADCAST}, msg \rangle$ (line 1 in Algorithm 3) for a message body msg to itself $v_{current}$. Here, it selects a tree from *trees* and makes each node send messages on the tree.

Algorithm 3 is a pseudocode on the tree selection. A function `selectTree` returns the ID of the tree t in *trees* such that (4) is minimum.

$$\max\left(\{ 0 \} \cup \left\{ distFor[v_{adj}] \;\middle|\; v_{adj} \in t.eagerPushPeers \right\}\right) \tag{4}$$

If t represents a rooted tree $(V, F, v_{current})$, $distFor[v_{adj}]$ represents $(\text{height}_{\text{subtree}_{(V,F,v_{current})}(v_{adj})} + 1)$. Thus, (4) represents

$$\max\left(\{ 0 \} \cup \left\{ \begin{array}{l} \text{height}_{\text{subtree}_{(V,F,v_{current})}(v)} + 1 \mid \\ \quad v \in \text{adj}_{(V,F,v_{current})}(v_{current}) \end{array} \right\} \right) \tag{5}$$

$=\text{height}_{(V,F,v_{current})}$.

This is equivalent to (1) where $v_{current} = s$. Therefore, this process intuitively means tree selection minimizing the maximum path length for the broadcast.

In practice, tree selection does not always minimize the maximum path length because the selection is based on local information and the network topology is dynamic. How the values $t.distFor$ are updated is important for improving the accuracy of the tree selection.

4.3 Message Propagation and Value Update

Message propagation and topology management during a broadcast mostly follow the Plumtree protocol for a tree determined by the tree selection algorithm. Algorithm 4 is a pseudocode on the message propagation, and Algorithm 5 is a pseudocode on the topology management and optimization. A node starts broadcasting by sending a `Gossip` message to itself.

Each tree t has two sets of neighbor nodes $t.eagerPushPeers$ and $t.lazyPushPeers$ (lines 2 and 3 in Algorithm 1) because a tree is maintained in the Plumtree protocol. The two sets are subject to eager push and lazy push for, respectively. Eager push is a protocol where a node sends the message body msg directly to the neighbor nodes. Lazy push is a protocol where a node sends the message ID $msgId$ first, and sends msg only when it receives a request from a receiving node of $msgId$. The former is used for fast message propagation on the tree, while the latter is used to guarantee the reliability of broadcasts and recover the tree topology.

To support dynamics of the network for tree selection, each node updates the values of $t.distFor$ (line 4 in Algorithm 1) through the message propagation with piggybacked "distance" information. Since the data size of the piggyback is very small, the additional cost of the communication is insignificant.

The more frequent broadcasts are, the faster the values of *distFor* follow the exact values for the dynamic network, and the more accurate tree selection becomes. On the other hand, the more the network changes, the more there is a delay in following the values, and the less accurate tree selection becomes. If the frequency of broadcast is too low and the *distFor* values likely become too stale, it is worth considering to update *distFor* values by sending a message to all the k trees periodically.

Finally, Algorithm 6 is a pseudocode on joining and leaving of nodes. `NeighborUp` (line 7) and `NeighborDown` (line 1) are events issued by a peer sampling service and update information for neighbor nodes. Moreover, the protocol also locally computes the values of $t.distFor$.

5. Analysis

This section provides an analysis of the proposed method in cost, scalability and robustness.

5.1 Cost

The memory space required on a node is increased by k in case the number of tree is k, that is a system parameter. The size of data structure *Node* (in Algorithm 1) would be tens of bytes and the size of data structure *Tree* would be hundreds of bytes. The space they consume is increased by the proposed method by k times. In the following experiments (Sect. 6), k is 10 and then the space overhead would be several kilobytes.

Algorithm 4: Data Propagation in a Node $v_{current}$

```
1  upon receiving
     ⟨Gossip, msg, msgId, treeId, round, distForSender⟩ from
     v_sender then
2      t ← trees[treeId]
3      if msgId ∉ receivedMsgs.keys() then
4          trigger a event with msg
5          receivedMsgs[msgId] ← msg
6          if ∃(id, _, _, _) ∈ missing s.t. id = msgId then
7              cancelTimer(msgId)
8          if v_sender ≠ v_current then
9              t.eagerPushPeers ←
                 t.eagerPushPeers ∪ {v_sender}
10             t.lazyPushPeers ← t.lazyPushPeers − {v_sender}
11             t.distFor[v_sender] ← distForSender
12         execEagerPush(msg, msgId, treeId, round, v_sender)
13         execLazyPush(msg, msgId, treeId, round, v_sender)
14         optimizeTree(msgId, round, v_sender)
15     else
16         t.eagerPushPeers ← t.eagerPushPeers − {v_sender}
17         t.lazyPushPeers ← t.lazyPushPeers ∪ {v_sender}
18         delete t.distFor[v_sender]
19         send ⟨Prune, treeId⟩ to v_sender
20 upon receiving ⟨Prune, treeId⟩ from v_sender then
21     t ← trees[treeId]
22     t.eagerPushPeers ← t.eagerPushPeers − {v_sender}
23     t.lazyPushPeers ← t.lazyPushPeers ∪ {v_sender}
24     delete t.distFor[v_sender]
25 function execEagerPush(msg, msgId, treeId, round, v_sender)
26     t ← trees[treeId]
27     for v_adj ∈ t.eagerPushPeers do
28         if v_adj = v_sender then
29             continue
30         maxDist ← getMaxDistExceptOne(t.distFor, v_adj)
31         send
             ⟨Gossip, msg, msgId, treeId, round + 1, maxDist + 1⟩
             to v_adj
32 function execLazyPush(msg, msgId, treeId, round, v_sender)
33     t ← trees[treeId]
34     for v_adj ∈ t.lazyPushPeers do
35         if v_adj = v_sender then
36             continue
37         maxDist ← getMaxDistExceptOne(t.distFor, v_adj)
38         lazyQueue ← lazyQueue ∪
             { (⟨IHave, msgId, treeId, round+1, maxDist+1⟩, v_adj) }
39     dispatch()
40 function dispatch()
41     announcements ← policy lazyQueue
42     for (data, v_adj) ∈ announcements do
43         send data to v_adj
44     lazyQueue ← lazyQueue − announcements
```

Algorithm 5: Tree Repair and Optimization in a Node $v_{current}$

```
1  upon receiving ⟨IHave, msgId, treeId, round, distForSender⟩
     from v_sender then
2      if msgId ∉ receivedMsgs.keys() then
3          if ¬isActiveTimer(msgId) then
4              setupTimer(msgId, timeout)
5          missing ← missing ∪
             { (msgId, treeId, v_sender, round, distForSender) }
6  upon receiving ⟨Timeup, msgId⟩ then
       // when a timer for msgId expires
7      (_, treeId, v_missing, round, _) ←
         removeFirstAnnouncement(missing, msgId)
8      t ← trees[treeId]
9      maxDist ← getMaxDistExceptOne(t.distFor, v_missing)
10     send ⟨Graft, msgId, treeId, round, maxDist + 1⟩ to
         v_missing
11 upon receiving ⟨Graft, msgId, treeId, round, distForSender⟩
     from v_sender then
12     t ← trees[treeId]
13     t.eagerPushPeers ← t.eagerPushPeers ∪ {v_sender}
14     t.lazyPushPeers ← t.lazyPushPeers − {v_sender}
15     t.distFor[v_sender] ← distForSender
16     if msgId ∈ receivedMsgs.keys() then
17         msg ← receivedMsgs[msgId]
18         maxDist ←
             getMaxDistExceptOne(t.distFor, v_sender)
19         send
             ⟨Gossip, msg, msgId, treeId, round, maxDist + 1⟩ to
             v_sender
20 function optimizeTree(msgId, round, v_sender)
21     if ∃(id, treeId, v_missing, r, distForMissing) ∈
         missing s.t. id = msgId then
22         if round − r ≥ threshold then
23             t ← trees[treeId]
24             t.eagerPushPeers ←
                 t.eagerPushPeers ∪ {v_missing}
25             t.lazyPushPeers ← t.lazyPushPeers − {v_missing}
26             t.distFor[v_missing] ← distForMissing
27             t.eagerPushPeers ←
                 t.eagerPushPeers − {v_sender}
28             t.lazyPushPeers ← t.lazyPushPeers ∪ {v_sender}
29             delete t.distFor[v_sender]
30             maxDist ←
                 getMaxDistExceptOne(t.distFor, v_missing)
31             send ⟨Graft, null, treeId, round, maxDist + 1⟩
                 to v_missing
32             send ⟨Prune, treeId⟩ to v_sender
```

The number of messages for tree construction (Sect. 4.1) is also increased by k. If the base protocol requires additional messages for topology maintenance, the number of such messages is also increased by k. Note that Plumtree does not require such additional messages.

Nevertheless, the number of messages for broadcasting (Sect. 4.3) does not increase because a broadcast is performed on a single tree after tree selection (Sect. 4.2) and all the additional information for the proposed method such as distance is piggybacked on the broadcasted messages.

Increases in memory space for tree management and in the number of messages for tree construction and maintenance are linear to the number of trees, that is a small constant, for example, 10.

Algorithm 6: Membership Management in a Node $v_{current}$

```
1  upon receiving ⟨NeighborDown, v_down⟩ then
2      for (_, t) ∈ trees do
3          t.eagerPushPeers ← t.eagerPushPeers − {v_down}
4          t.lazyPushPeers ← t.lazyPushPeers − {v_down}
5          delete t.distFor[v_down]
6      missing ← { (_, _, v_sender, _) ∈ missing | v_sender ≠ v_down }
7  upon receiving ⟨NeighborUp, v_up⟩ then
8      distSet ← ∅
9      for (treeId, t) ∈ trees do
10         maxDist ← getMaxDistExceptOne(t.distFor, v_up)
11         distSet ← distSet ∪ {(treeId, maxDist + 1)}
12     send ⟨DistUpdate, distSet⟩ to v_up
13 upon receiving ⟨DistUpdate, distSet⟩ from v_sender then
14     reverseDistSet ← ∅
15     for (treeId, distForSender) ∈ distSet do
16         if treeId ∉ trees.keys() then
17             trees[treeId] ← new Tree
18             trees[treeId].lazyPushPeers ← peers
19         added ← [v_sender ∉ t.eagerPushPeers]
20         t ← trees[treeId]
21         t.eagerPushPeers ← t.eagerPushPeers ∪ {v_sender}
22         t.lazyPushPeers ← t.lazyPushPeers − {v_sender}
23         t.distFor[v_sender] ← distForSender
24         if added then
25             maxDist ←
                 getMaxDistExceptOne(t.distFor, v_sender)
26             reverseDistSet ←
                 reverseDistSet ∪ {(treeId, maxDist + 1)}
27     if reverseDistSet ≠ ∅ then
28         send ⟨DistUpdate, reverseDistSet⟩ to v_sender
```

5.2 Scalability

Scalability along the number of node is completely determined by the single tree construction method on which we base. In our case, it is Plumtree. The proposed method is scalable to the same degree as Plumtree.

5.3 Robustness

Robustness is also determined by the single tree construction method. In our case, it is Plumtree. The effect of node failures to Plumtree was evaluated in Sect. 4.3 of the Plumtree paper [6]. In the proposed method, last delivery hop (LDH) and reliability are just the same as Plumtree because the proposed method utilizes a single tree for broadcast. Relative message redundancy (RMR) can be k-fold but repair of a tree requires only a few messages in Plumtree protocol. Healing time is the same as Plumtree. Healing, i.e. tree repair, for each k tree can occur at various times.

If a repair happens along the Plumtree protocol, distance information ($distFor$ in Algorithm 1) could be incorrect for a while. It lowers efficiency of the proposed method until the distance information is updated but broadcasting keeps working.

5.4 Impact of the Number of Trees

The proposed method constructs k trees (Sect. 4.1). The k affects the effect of the proposed method. Larger k is more effective, but the effect of an increase by one of k diminishes as k increases. In other words, smaller k yields large part of the effect. An analysis follows.

There are $|V|$ possible trees where $|V|$ is the number of nodes because all the node can be the root of a possible tree. Note that the topologies of two possible trees are the same except their root nodes. The proposed method constructs k trees randomly selected from $|V|$ possible trees. A broadcast enjoys the best tree from k trees by the tree selection (Sect. 4.2). We can rank all the possible trees from the first rank to the last, $|V|$-th rank though the order is partial order. The expected value of the rank of the best tree in k trees is $|V|/k + 1$. In case the number of nodes $|V|$ is 10000, we expect the 5000-th ranked tree where $k = 1$, the 3333-rd ranked tree where $k = 2$, and the 909-th ranked tree where $k = 10$. The expected rank number decreases monotonically as k increases. The 10 as k is the parameter we adopt in our experiments shown in Sect. 6.

We expect the 833-rd ranked tree where $k = 11$ by increasing k by one from 10. How much better the 833-rd is compared to the 909-th depends on the distribution of tree heights of the possible $|V|$ trees. It is still an open problem to analyze such distribution in Plumtree, but only 10 trees have already yielded large part of the effects. For example, in Fig. 3 showing maximum path lengths on a random graph, a single tree ($k = 1$) showed about 15, only 10 trees ($k = 10$) showed about 11. Even 10000 trees ($k = 10000$) will show a number no less than 6, that is the number `flooding` showed and the lower limit.

6. Evaluation

We conducted experiments by simulating broadcasts on an overlay network, and measured the path lengths and the number of messages. There is no need for advanced features simulating physical networks that existing network simulators provide. And then we developed a simple simulator by ourselves. Thereby, we observed the effect of the proposed method on them.

6.1 Experiment Settings

The simulating procedure for a combination of a method and an overlay network is as follows:

1. We generate an overlay network $G = (V, E)$, on which spanning trees are built.
2. We construct spanning trees if a method requires them. For example, multiple trees are built for the proposed method, a single tree is built for Plumtree and no tree is built for flooding. Number of trees for the proposed method is 10 as stated below. The start node for tree

construction $v_{start} \in V$ is randomly determined.

3. Let the following procedure be 1 cycle, and we execute 1000 cycles.
   a. We determine a source node $v_{start} \in V$ randomly.
   b. We issue a broadcast on the source node v_{start}.
   c. We measure the path lengths and the number of messages.
   d. In Plumtree and the proposed method, we update tree topologies by following each method. Note that the Plumtree protocol changes the tree topology as time goes by.

We use the following graphs as the overlay network, on which spanning trees are built. The number of nodes for each graph is $|V| = 10000$.

`Random Graph`
: A random graph generated by Erdös-Rényi model [21]. In this experiment, we generate a graph of $\Gamma_{10000,50000}$ where $|V| = 10000$ and $|E| = 50000$. $\Gamma_{n,N}$ is a graph generation model whose N edges are randomly selected from $\binom{n}{2}$ node pairs for the fixed number of nodes n.

`BA Model Graph`
: A graph generated by Barabási-Albert model [22]. The model generates a scale-free graph. In this experiment, $|V| = 10000$, and the number of edges added at each step of the generation is 5. Thus, $|E| \approx 50000$.

`Torus Grid Graph`
: A $m \times n$ torus grid graph is a graph that adds edges between the leftmost nodes and the rightmost nodes and between the topmost nodes and bottommost nodes to a $m \times n$ grid graph. It is formally a isomorphic graph to the graph $G' = (V', E')$ where $V' = \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ and $E' = \{\{(i, j), (i, j + 1)\} \mid i \in \mathbb{Z}/m\mathbb{Z} \wedge j \in \mathbb{Z}/n\mathbb{Z}\} \cup \{\{(i, j), (i + 1, j)\} \mid i \in \mathbb{Z}/m\mathbb{Z} \wedge j \in \mathbb{Z}/n\mathbb{Z}\}$. In this experiment, we use a 100×100 torus grid graph.

All nodes keep alive and the network topology is constant during a simulation. Changes of participating nodes and network are dealt with by Plumtree protocol [6] that the proposed method utilizes.

We use the following broadcast methods for comparison.

`Plumtree`
: A broadcast method following Plumtree protocol, which manages a spanning tree on the network.

`proposed`
: The proposed method, which manages multiple spanning trees and where each source node selects a spanning tree. The number of trees is 10.

`ideal proposed`
: A node can select the best tree minimizing the maximum path length though the distance information has not been updated timely. Plumtree updates the tree topology along broadcasts and then the proposed method updates the distance information. But it can happen for the distance information updates not to follow the topology updates timely. The number of trees is 10.

`multiple Plumtrees`
: A method that manages 10 Plumtrees and broadcasts to all of them. The path length is the minimum path length among all the trees. The number of messages is approximately 10 times larger than `Plumtree`.

`flooding`
: A broadcast method by flooding.

Since we assume that there are multiple source nodes for broadcasts, we set the value *threshold* used in the topology optimization of Plumtree protocol to 7. We set the communication time between each node to be constant and the value *timeout* in Algorithm 5 to five times the communication time. Note that concrete time length does not matter in these experiments.

6.2 Experiment Result

Figure 3 and Fig. 4 show the transitions of the maximum path lengths and the average path lengths of broadcasts for `Random Graph`, respectively. The translucent lines represent the measured values, and the opaque lines represent the moving average value for 50 cycles. The same representation is used in the following figures showing the transition of path lengths. We use moving averages to facilitate comparison of the relative values for each broadcast method. As a result of Fig. 3, the proposed method `proposed` reduces the maximum path length by approximately 28% compared with `Plumtree`. However, it is not as short as that of `ideal proposed`, which suggests that the update of values *distFor* does not sufficiently follow the topology changes by Plumtree protocol. In addition, the maximum path length of `ideal proposed` is almost the same as that of `multiple Plumtrees`. This suggests that the proposed method can potentially achieve the same length as the maximum path length when using multiple Plumtrees despite the number of messages on using a single Plumtree if the values *distFor* follows the topology changes. Furthermore, as a result of Fig. 4, the magnitude relationship of the average path length of each broadcast method tends to be almost the same as that of the maximum path length.

Figure 5 shows the relation between the maximum path length and the number of messages for `Random Graph`. We plotted the results for 901–1000 simulation cycles. As a result, the numbers of messages of `Plumtree`, `proposed`, and `ideal proposed` are approximately 10000 ($\approx |V|$), which are much smaller than those of `flooding` and `multiple Plumtrees`, and especially, are approximately 11% of that of `flooding`. This is because, in the former, most messages are sent on a single spanning tree and there are few duplicate messages, while in the latter, many duplicate messages occur due to the property of the methods. Therefore, the experimental results for `Random Graph` show that the proposed method shortens path lengths compared to `Plumtree` while maintaining the small number of messages.

Figure 6 and Fig. 7 show the transitions of the maxi-

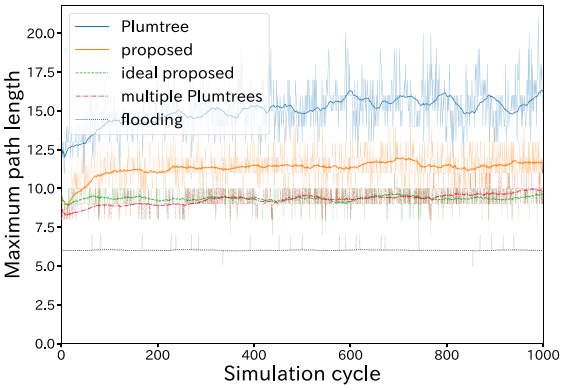


Fig. 3 Maximum path length for `Random Graph`.

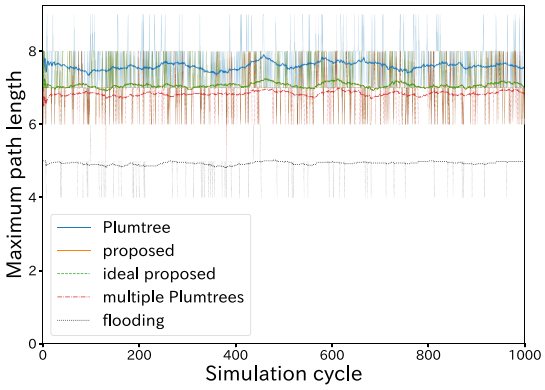


Fig. 6 Maximum path length for `BA Model Graph`.

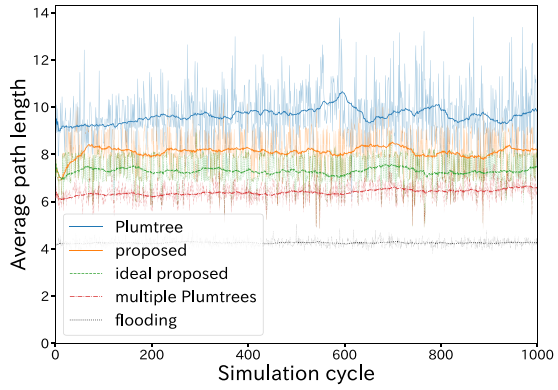


Fig. 4 Average path length for `Random Graph`.

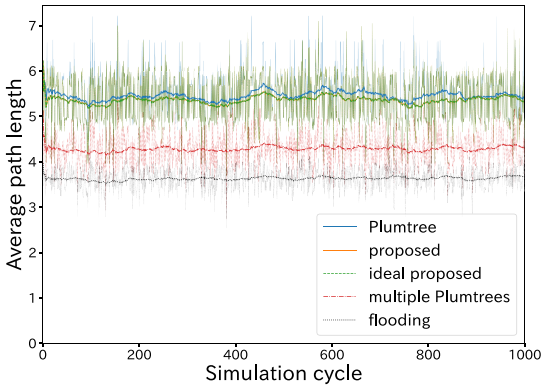


Fig. 7 Average path length for `BA Model Graph`.

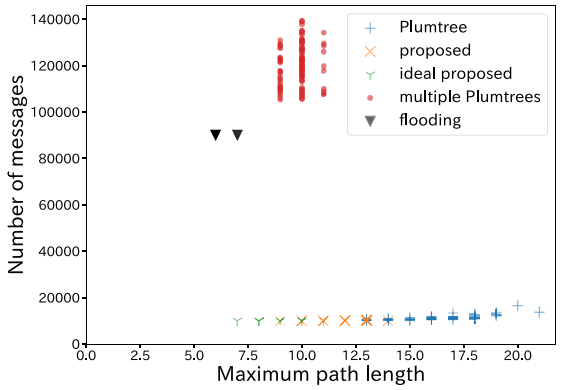


Fig. 5 Relation between the maximum path length and the number of messages for `Random Graph`.

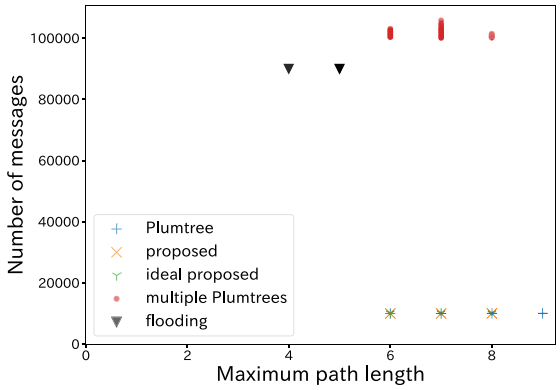


Fig. 8 Relation between the maximum path length and the number of messages for `BA Model Graph`.

mum path lengths and the average path lengths of broadcasts for `BA Model Graph`, respectively. As a result of Fig. 6, the proposed method `proposed` reduces the maximum path length by approximately 7% compared to `Plumtree`. In addition, it achieves almost the same maximum path length as that of `ideal proposed`, which suggests that the update of values *distFor* significantly follows the topology changes by Plumtree protocol for `BA Model Graph`.

Figure 8 shows the relation between the maximum path length and the number of messages for `BA Model Graph`. We plotted the results for 901–1000 simulation cycles. It presents a similar result to Fig. 5 on the number of messages.

Figure 9 and Fig. 10 show the transitions of the maximum path lengths and the average path lengths of broadcasts for `Torus Grid Graph`, respectively. As a result of Fig. 9, the proposed method `proposed` reduces the maximum path length by approximately 11% compared to `Plumtree`. Although the results are similar to those for `Random Graph`, the reduction ratio is inferior.

Figure 11 shows the relation between the maximum

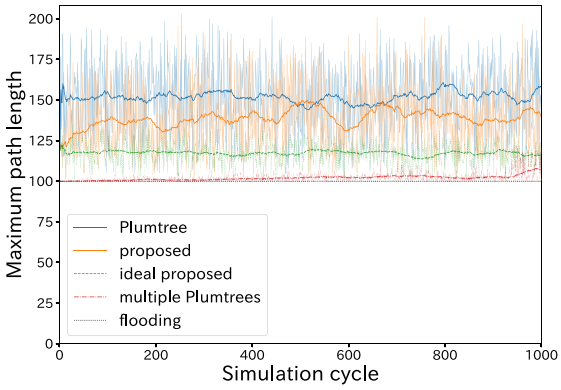


Fig. 9 Maximum path length for `Torus Grid Graph`.

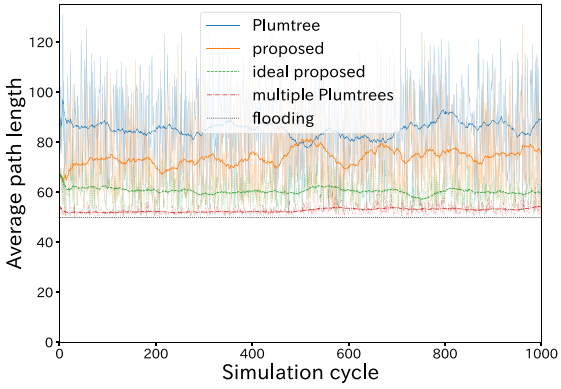


Fig. 10 Average path length for `Torus Grid Graph`.

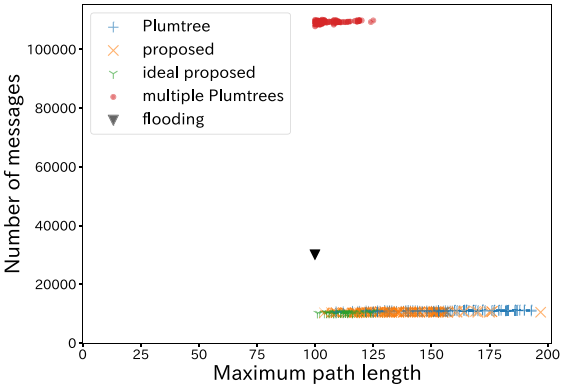


Fig. 11 Relation between the maximum path length and the number of messages for `Torus Grid Graph`.

path length and the number of messages for `Torus Grid Graph`. We plotted the results for 901–1000 simulation cycles. It presents a similar result to Fig. 5 on the number of messages.

Table 1 summarizes the reduction ratio of maximum path length by the proposed method compared to Plumtree. How much the proposed method can contribute in the real world application depends heavily on the overlay network. The method contributes much if which spanning tree is efficient depends heavily on each source node. In other words,

Table 1 Reduction ratio of maximum path length by proposed method compared to Plumtree.

Overlay network	Reduction ratio
`Random Graph`	28%
`BA Model Graph`	7%
`Torus Grid Graph`	11%

if every source node has its own efficient tree, the proposed method contributes much. `BA Model Graph` is known as a scale-free network, that provides a small diameter. Such a network has hub nodes, that have a large number of neighbors. The hub nodes should be part of a large number of trees and they will reduce a variety of multiple spanning trees. And then the proposed method contributes little on `BA Model Graph`. Nevertheless, the proposed method could contribute 7% even on a natively effective scale-free network. It is natural for us to expect 7% or more of the contribution in a real world application.

7. Conclusion

In this paper, we proposed a novel broadcast method. It constructs multiple spanning trees on the overlay network, and a source node selects an appropriate tree from them when broadcasting. This reduces the frequency of inefficient broadcasts for multiple source nodes. It thereby achieves shortening routing path lengths while maintaining a small number of messages.

The evaluation experiments show that the effect of the proposed method on the path lengths is dependent on the topology of the overlay network. For a random graph, the reduction ratio was approximately 28% compared to Plumtree. Moreover, the number of messages was almost the same as the number of nodes. This shows that the proposed method shortens path lengths while maintaining the small number of messages.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number JP21H04872.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," Proc. Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP'07, New York, NY, USA, pp.205–220, ACM, 2007.

[2] N. Ramzan, H. Park, and E. Izquierdo, "Video streaming over P2P networks: Challenges and opportunities," Signal Processing: Image Communication, vol.27, no.5, pp.401–411, May 2012.

[3] A. Yahyavi and B. Kemme, "Peer-to-peer architectures for massively multiplayer online games: A survey," ACM Comput. Surv., vol.46, no.1, pp.9:1–9:51, July 2013.

[4] A. Dorri, S.S. Kanhere, and R. Jurdak, "Towards an optimized blockchain for IoT," Proc. Second IEEE/ACM International Conference on Internet-of-Things Design and Implementation, IoTDI'17, New York, NY, USA, pp.173–178, ACM, 2017.

[5] P. Ruiz and P. Bouvry, "Survey on broadcast algorithms for mobile ad hoc networks," ACM Computing Surveys (CSUR), vol.48, no.1, pp.8:1–8:35, July 2015.
[6] J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), pp.301–310, Oct. 2007.
[7] T. Kaneko and K. Shudo, "Broadcast with tree selection on an overlay network," Proc. ICOIN 2022, Jan. 2022.
[8] A. Kermarrec, L. Massoulie, and A.J. Ganesh, "Probabilistic reliable dissemination in large-scale systems," IEEE Trans. Parallel Distrib. Syst., vol.14, no.3, pp.248–258, March 2003.
[9] M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-bandwidth multicast in cooperative environments," ACM SIGOPS Operating Systems Review, vol.37, no.5, pp.298–313, Oct. 2003.
[10] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast," Proc. 2006 IEEE International Conference on Network Protocols, pp.2–11, Nov. 2006.
[11] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," Middleware 2001, R. Guerraoui, ed., Lecture Notes in Computer Science, vol.2218, pp.329–350, Springer Berlin Heidelberg, 2001.
[12] M. Castro, P. Druschel, A. Kermarrec, and A.I.T. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," IEEE J. Sel. Areas Commun., vol.20, no.8, pp.1489–1499, Oct. 2002.
[13] M. Ferreira, J. Leitão, and L. Rodrigues, "Thicket: A protocol for building and maintaining multiple trees in a P2P overlay," 2010 29th IEEE Symposium on Reliable Distributed Systems, pp.293–302, Oct. 2010.
[14] L. Junhai, X. Liu, and Y. Danxia, "Research on multicast routing protocols for mobile ad-hoc networks," Computer Networks, vol.52, no.5, pp.988–997, April 2008.
[15] X. Li, T. Liu, Y. Liu, and Y. Tang, "Optimized multicast routing algorithm based on tree structure in manets," China Commun., vol.11, no.2, pp.90–99, Feb. 2014.
[16] Y.H. Chen and E.H.K. Wu, "Bandwidth-satisfied multicast by multiple trees and network coding in lossy manets," IEEE Syst. J., vol.11, no.2, pp.1116–1127, June 2017.
[17] H. Wu and X. Jia, "QoS multicast routing by using multiple paths/trees in wireless ad hoc networks," Ad Hoc Networks, vol.5, no.5, pp.600–612, July 2007.
[18] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," ACM Trans. Comput. Syst., vol.25, no.3, pp.8–es, Aug. 2007.
[19] J. Leitao, J. Pereira, and L. Rodrigues, "HyParView: A membership protocol for reliable gossip-based broadcast," 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pp.419–429, June 2007.
[20] M. Al Hasan, "Methods and applications of network sampling," Optimization Challenges in Complex, Networked and Risky Systems, INFORMS TutORials in Operations Research, ch. 5, pp.115–139, INFORMS, Oct. 2016.
[21] P. Erdös and A. Rényi, "On random graphs I," Publicationes Mathematicae, vol.6, no.26, pp.290–297, 1959.
[22] A.L. Barabási and R. Albert, "Emergence of scaling in random networks," Science, vol.286, no.5439, pp.509–512, Oct. 1999.

Takeshi Kaneko received the B.S. and M.S. degrees in information science from Tokyo Institute of Technology, Tokyo, Japan, respectively in 2019 and 2021. He is currently working with Yahoo Japan Corporation. His research interests include distributed systems.

Kazuyuki Shudo received the B.E. degree in 1996, the M.E. degree in 1998, and the Ph.D. degree in 2001 all in computer science from Waseda University. He worked as a Research Associate at the same university from 1998 to 2001. He later served as a Research Scientist at National Institute of Advanced Industrial Science and Technology. In 2006, he joined Utagoe Inc. as a Director, Chief Technology Officer. From December 2008, he served as an Associate Professor at Tokyo Institute of Technology. Since April 2022, he currently serves as a Professor at Kyoto University. His research interests include distributed computing, programming language systems and information security. Dr. Shudo has received the best paper award at SACSIS 2006, Information Processing Society Japan (IPSJ) best paper award in 2006, the Super Creator certification by Japanese Ministry of Economy Trade and Industry (METI) and Information Technology Promotion Agency (IPA) in 2007, IPSJ Yamashita SIG Research Award in 2008, Funai Prize for Science in 2010, The Young Scientists' Prize, The Commendation for Science and Technology by the Minister of Education, Culture, Sports, and Technology in 2012, and IPSJ Nagao Special Researcher Award in 2013. He is a member of IEEE, IEEE Computer Society, IEEE Communications Society and ACM.