Constructing Distributed Doubly Linked Lists without Distributed Locking

Kota Abe
Osaka City University, Osaka, Japan / NICT, Tokyo, Japan
Email: k-abe@media.osaka-cu.ac.jp

Mikio Yoshida†
BBR Inc., Osaka, Japan

***Abstract*—A distributed doubly linked list (or bidirectional ring) is a fundamental distributed data structure commonly used in structured peer-to-peer networks. This paper presents *DDLL*, a novel decentralized algorithm for constructing distributed doubly linked lists. In the absence of failure, DDLL maintains consistency with regard to lookups of nodes, even while multiple nodes are simultaneously being inserted or deleted. Unlike existing algorithms, DDLL adopts a novel strategy based on conflict detection and sequence numbers. A formal description and correctness proofs are given. Simulation results show that DDLL outperforms conventional algorithms in terms of both time and number of messages.**

I. Introduction

A distributed doubly linked list (or a bidirectional ring) is a distributed data structure where each node is connected by bidirectional links. Distributed doubly linked lists are commonly used as a fundamental data structure in structured peer-to-peer (P2P) networks, such as Chord [1], Chord$^{\#}$ [2], Skip graphs [3], SkipNet [4], and their variants.

In distributed doubly linked lists, each node has pointers, such as an IP address, to the next (successor) and previous (predecessor) nodes. The difficulty of designing algorithms for constructing distributed doubly linked list is that they must consistently update opposite-direction pointers on two physically distinct nodes even in environments where multiple nodes may simultaneously and independently try to insert or delete themselves.

Several algorithms for constructing distributed doubly linked lists are known, and most can be classified as *eventual consistency* approaches, such as in [1], [5], [6], or *locking* approaches, such as in [7], [8].

The eventual consistency approach is well known for its use in the Chord algorithm [1]. Using this approach, a linked list may temporarily be inconsistent after a node insertion or deletion and it is recovered by a periodically executed *stabilizing* procedure. While the benefit of this approach is that it is simple and straightforward to recover from failure, the drawback is that it does not guarantee consistency with regard to lookups; some nodes might be temporarily unreachable even when there is no failure. This issue affects the accuracy of Chord and other distributed hash tables (DHTs). A DHT stores data at a node which is responsible for the key, so it is desirable that lookups of the node responsible for a key return consistent results. However, in the eventual consistency approach, lookup results may differ depending on the querying node.

In the locking approach, distributed locking techniques are used for mutual exclusion of simultaneous node insertion or deletion. When a node is to be inserted or deleted, it first sends a lock request message to a *lock node* (typically a neighbor node) to acquire a lock, changes the links of the adjacent nodes, then finally sends an unlock request message to the lock node to release the lock. In this approach, pointers to the next and previous nodes are consistently updated and all inserted nodes can be looked up.

However, distributed locking has a drawback. In this approach, a short locking duration is desired because locking blocks insertion and deletion of other nodes. However, in a distributed environment, the lock duration may be long due to transmission delays. Furthermore, the duration may be quite long if the node that acquired the lock fails before sending an unlock request. In this case, the lock must be forcibly released, usually by a timeout. Implementations handling such cases are troublesome in general.

In this paper, we propose a novel decentralized algorithm, *DDLL* (Distributed Doubly Linked List) for constructing distributed doubly linked lists. DDLL is based on neither the eventual consistency approach nor the locking approach. Instead, DDLL adopts a simple strategy based on conflict detection and sequence numbers. In DDLL, next-node pointers are always correct.

This work was supported by JSPS KAKENHI (24500089).
†Deceased August 29, 2014.

Previous-node pointers are not always correct, but the duration of the incorrect state does not exceed the one-way message transmission time. The operations for node insertion and deletion are atomic and all inserted nodes can be looked up from any inserted node, even if multiple node insertion or deletion are interleaved.

This paper is organized as follows. In the next section, we review related work. Section III describes some preliminaries. Section IV describes the DDLL algorithm and Section V gives correctness proofs. Section VI provides discussion and evaluation. Lastly, Section VII summarizes this work.

II. Related Work

Algorithms for distributed doubly linked lists, or bidirectional rings, have been well studied in the field of structured P2P networks. As described above, most conventional algorithms can be classified as either eventual consistency approaches or locking approaches.

There is a lot of work on eventual consistency approaches, e.g., [1], [5], [6], [9] to name a few. As described earlier, such approaches do not guarantee consistency with regard to lookups.

Ghodsi [7] proposed *Atomic Ring Maintenance*. This algorithm adopts the locking approach and has recovery procedures to handle failures. It uses timeouts to forcibly release remaining locks. It also considers the case where timeout is premature. In such a case, a stabilizing procedure similar to the one used in Chord recovers incorrect links. This algorithm requires a FIFO property in the underlying transport, so message order must be preserved. Ghodsi has proved that lookup consistency cannot be guaranteed in an asynchronous network that partitions. Furthermore, it is also conjectured that this assertion can be extended to an asynchronous network without partitioning. From these impossibility results, when we discuss lookup consistency in this paper, no failure is assumed.

The protocol proposed by Li *et al.* [8] is another locking-based protocol for maintaining a bidirectional ring. They first present a protocol by which a node can be inserted between two arbitrary nodes (i.e., the node does not have a specific key) and later modify their protocol to be used for Chord, where each node has a key and must be inserted in a proper location. However, they do not address recovery from failures.

Risson *et al.* [10] proposed a fault-tolerant active ring protocol that guarantees consistency even in the presence of failure. In this protocol, each insertion and deletion is treated as a transaction and handled with the Paxos Commit algorithm. The drawbacks of this approach are its implementation difficulty and message complexity.

In the field of parallel programming, Sundell *et al.* [11] have proposed a lock-free deque based on doubly linked list. In this algorithm, pointers to the next and previous nodes are updated using the compare-and-swap (CAS) instruction, which atomically modifies content in a memory to a given new value if the content in a memory equals a given old value. Sundell's algorithm and DDLL share the idea that use of the CAS(-like) operation for updating the next-node pointer, but differ in the following points: (1) For updating the previous-node pointers, DDLL does not use CAS but employs sequence numbers, which eliminates the necessity of iterating CAS operations. (2) DDLL is for distributed linked lists (rather than in-memory deques) and (3) DDLL provides recovery procedure from failures.

III. Preliminaries

DDLL constructs a distributed doubly linked list, sorted by a node-specific *key*. Each node connects with other nodes (or the node itself) by a *right link* and a *left link*. We assume that rightward is the key-increasing direction. We also assume that the linked list is circular, so the right node of the node with the maximum key is the node with the minimum key, and vice versa. A doubly linked list can be considered as a combination of two singly linked lists with opposite directions. We call these singly linked lists *rightward linked lists* and *leftward linked lists*.

Keys are elements of a totally ordered set. We assume that each key is unique. A simple way to eliminate duplicate keys is to add some random bits on the right side of identical keys. The notation (a, b) denotes the interval from a to b in a circular key space, excluding a and b. The notations $[a, b)$ and $(a, b]$ are similar but respectively include a and b.

Each node executes the same algorithm at an arbitrary speed. We do not assume Byzantine failures. We assume that message delivery between nodes is reliable but asynchronous; messages are eventually received, but there is no bound on message delivery time. Unless otherwise noted, we assume that message order is not preserved (i.e., is non-FIFO).

We denote the key of node u by u as well. Each node maintains several variables, including **s** (node status), **l** (left link), **r** (right link), $\mathbf{l}_{\mathrm{seq}}$ (left sequence number), and $\mathbf{r}_{\mathrm{seq}}$ (right sequence number). These are explained in the sections below. To simplify the presentation, we do not distinguish between a locator (such as an IP address) and a key of a node. For example, l (resp., r) represents both the locator and the key of the left (resp., right)

node. We denote a variable x on node u by $u.x$.

IV. The DDLL Algorithm

A. Fundamental Idea

On either node insertion or deletion, the right link of the immediate left node and the left link of the immediate right node must be updated. We denote the message for updating a right link of a remote node as the *SetR* message and for updating a left link as the *SetL* message.

1) Updating a Right Link: We denote the node to be inserted or deleted by u, u's left node by p, and u's right node by q. We would like for node p to accept a SetR message from u only when no conflict occurs. We would like to reject a SetR message if another node has been inserted between p and q or if q has been deleted. To detect conflicts, we use the fact that if another node has been inserted between p and q, or if q has been deleted, then p's right link points to a node other than q. Thus, we include the *expected current right node of* p (denoted by $r_{\rm cur}$) in a SetR message. When p receives the message, p compares $r_{\rm cur}$ in the message with $p.r$. If they are equal, p accepts the request and changes $p.r$ accordingly, but otherwise it indicates that conflicts have occurred and p rejects the request. Using this method, a right link can be updated in a controlled manner.

The $r_{\rm cur}$ field checks whether the sender node u knows the latest situation of the recipient node p and effectively serializes simultaneous node insertion and deletion in a rightward linked list without the necessity of distributed locking.

2) Updating a Left Link: A left link should be updated consistently with the corresponding right link. We devised a novel method that allows a node to process a SetL message delivered in an arbitrary order and eliminates the necessity of distributed locking. When a SetR message is accepted, a corresponding SetL message is sent. Our idea is to assign a consistently increased sequence number to a SetL message. For any node x, whenever a new SetL message is sent to x, the sequence number is increased. The sequence number in a SetL message is used by the recipient node for deciding whether the received message is newer than the ones previously received. Each node maintains a *left sequence number* $l_{\rm seq}$ that is the maximum sequence number for SetL messages received thus far. When node x receives a SetL message, it compares the sequence number of the message with $x.l_{\rm seq}$, and if the former is larger than the latter, x changes its left link accordingly and updates $x.l_{\rm seq}$.

Each node also maintains a sequence number for its right node. This number is called the *right sequence number* $r_{\rm seq}$. As we prove later in Theorem 1, for any inserted node u, there exists exactly one inserted node v that satisfies $v.r = u$. We denote such a node by $\mathrm{left}(u)$. $\mathrm{left}(u)$ is u's correct left node. When $\mathrm{left}(u)$ is changed from node v to node w by node insertion or deletion, $v.r_{\rm seq}$ is incremented and transferred to w. When any two nodes v and u that satisfies $v.r = u$ are in steady state (i.e., no message is in transmission to them), $v = u.l \wedge v.r_{\rm seq} = u.l_{\rm seq}$ holds (see Section V).

TABLE I: Node status

Status	Description
out	the node is out of the linked list
ins	waiting for the SetRAck/SetRNak message on insertion
in	(at least) inserted into the rightward linked list
del	waiting for the SetRAck/SetRNak message on deletion

B. Detailed Algorithm

Pseudo code of the algorithm is shown in Fig. 1. We use a notation similar to the abstract protocol notation introduced in [12]. In this notation, the behavior of a node is defined by the **var** section, the **init** section, and a set of *actions*. Each action is of the form $\langle$*guard*$\rangle \rightarrow \langle$*statement*$\rangle$, delimited by **[]**. The statement of an action is executed only if the guard expression is true. The execution of an action is atomic; no action can be executed while another action is executing on the same node. If multiple actions can be executed on the same node, one of them is randomly selected and executed. An assignment statement (**:=**) can assign multiple values to multiple variables.

Each node u has a status $u.s$, which is initialized to *out*. The possible statuses are listed in Table I and explained below. We assume that a node drops all the messages it receives when its status is *out*.

C. Creation

To create a new linked list, the initial node u (the first node in the linked list) sends a message *Create()* to u. In action A_1, u initializes both the right and left links to itself, $u.s$ to *in*, and the right and left link sequence numbers to 0.

D. Insertion

Let us assume that node u is going to be inserted between node p and q, where p and q are respectively u's immediate left and right nodes and satisfies $p.r = q \wedge u \in (p, q)$. How u finds p and q is discussed later in Section IV-F. We assume that both the initial $p.r_{\rm seq}$ and $q.l_{\rm seq}$ are i (Fig. 2). To insert node u, u sends a message *Insert(p, q)* to u. Then, the following actions

```
1  process u
2  var s: {out, ins, in, del}
3    l, r: {pointer to a node or nil}
4    l_seq, r_seq: {integer or nil}
5  init s = out; l = r = nil; l_seq = 0; r_seq = nil
6  begin
7    {Create a linked list}
8    (A_1) receive Create() from app →
9      l, r, s, l_seq, r_seq := u, u, in, 0, 0
10   {Insert between p and q}
11 [] (A_2) receive Insert(p, q) from app →
12     if (s ≠ out ∨ u ∉ (p, q)) then error; fi
13     l, r, s := p, q, ins
14     send SetR(u, r, l_seq) to l
15   {Delete}
16 [] (A_3) receive Delete() from app →
17     if (s ≠ in) then error
18     else if (u = r) then {in case of the last node}
19       s := out
20     else s := del; send SetR(r, u, r_seq + 1) to l; fi
21 [] (A_4) receive SetR(r_new, r_cur, r_newseq) from v →
22     if (s = in ∧ r = r_cur) then
23       if (r_new = v) then {insertion case}
24         send SetL(r_new, r_seq + 1) to r
25       else {deletion case}
26         send SetL(u, r_newseq) to r_new; fi
27       send SetRAck(r_seq + 1) to v
28       r, r_seq := r_new, r_newseq
29     else send SetRNak() to v; fi
30 [] (A_5) receive SetRAck(r_newseq) from v →
31     if (s = ins) then
32       s, r_seq := in, r_newseq
33     elseif (s = del) then
34       s := out; fi
35 [] (A_6) receive SetRNak() from v →
36     if (s = ins) then
37       s := out; error {app retries insertion later}
38     elseif (s = del) then
39       s := in; error; fi {app retries deletion later}
40 [] (A_7) receive SetL(l_new, seq) from v →
41     if (l_seq < seq) then l, l_seq := l_new, seq; fi
42 end
```

Fig. 1: DDLL algorithm (without optimization)

are executed.

(A_2) u sets u's left link and right link to p and q, respectively. u also sets $u.s$ as *ins* to indicate u is inserting. u sends a SetR message to p, which contains u (as the new right node), q (as the expected current right node, or r_{cur}), and zero (as the new right sequence number, or r_{newseq}).

(A_4) On receiving the SetR message, p checks whether its status is *in* and r_{cur} equals $p.r$. If the former is false, either p has not received a SetRAck message after its insertion (as we describe next, SetRAck message is to inform that node insertion or deletion is succeeded), or p has started its deletion. If the latter is false, it indicates either that another node has inserted at the right side of p, or that q has been deleted. In either case, p rejects the request and sends a SetRNak message to u to notify that the insertion failed. Otherwise, p sends a SetL message to p's right node (q in this case) to update its left link to u. The SetL message contains u (as the new left node) and $p.r_{\mathrm{seq}}+1\,(=i+1)$ (as the sequence number of the SetL message). Next, p sends a SetRAck message to u to notify that the insertion was successful. Because $\mathrm{left}(q)$ is changed from p to u, the incremented right sequence number for q should be transferred from p to u. For this purpose, the SetRAck message contains $p.r_{\mathrm{seq}}+1\,(=i+1)$. Finally, p changes $p.r$ to u and $p.r_{\mathrm{seq}}$ to 0 (r_{newseq}). Because u's right link has already been set to q, the rightward linked list is never interrupted, even for a moment. Note that at this moment, $p.r_{\mathrm{seq}} = u.l_{\mathrm{seq}}$ holds.

(A_5) On receiving the SetRAck message, u confirms that u is successfully inserted. Node u updates $u.s$ to *in* to indicate that u is *inserted*, and sets $u.r_{\mathrm{seq}}$ to $i+1$.

(A_7) On receiving the SetL message, q compares the sequence number of the SetL message with $q.l_{\mathrm{seq}}$. If the former is larger (we assume this case), q updates $q.l$ to u and $q.l_{\mathrm{seq}}$ to $i+1$. Otherwise, q ignores the message.

In the scenario above, it is assumed that a SetRAck message is sent to u in A_4. If a SetRNak message is sent (i.e., in the case of insertion failure), then (A_6) $u.s$ is reverted to *out* and u retries the insertion procedure from locating its insertion position.

Note that a node u might receive a SetL message before receiving a SetRAck message. This happens, for example, when another node is inserted between p and u while the SetRAck message from p to u is still in transmission. This is normal and the algorithm can handle this situation. Actually we consider a node u becomes *inserted* at the moment when a SetRAck message is sent to u (see Section V).

Figure 3 depicts the situation where two nodes send a SetL message to the same node. There are 4 nodes A, B, C and D ($A < B < C < D$) and nodes A and D are initially inserted. $A.r_{\mathrm{seq}}$ and $D.l_{\mathrm{seq}}$ are i. Nodes B and C are then inserted in this order. When D receives the SetL message from C, its left link is updated to C and its left sequence number is updated to $i+2$. When D later receives the SetL message from B, D ignores it because its sequence number ($i+1$) is smaller than D's left sequence number ($i+2$). Thus, the receiving order of the SetL message does not affect the final results.

E. Deletion

Let us assume that node u, which is inserted between p and q, is going to be deleted. We also assume that both $p.r_{\mathrm{seq}}$ and $u.l_{\mathrm{seq}}$ are i_1 and that both $u.r_{\mathrm{seq}}$ and $q.l_{\mathrm{seq}}$ are i_2 (Fig. 4). To delete node u, u sends a message *Delete()* to u. Then, the following actions are executed.

(A_3) If $u.s$ is not *in*, deletion is rejected because it is uncertain whether u is inserted. If u is the last node (i.e.,

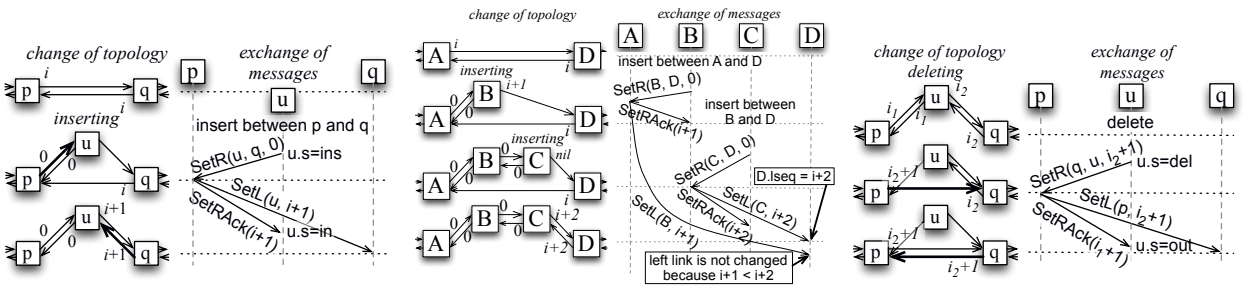

Fig. 2: Message sequence diagram of node insertion

Fig. 3: A sequence where a node receives 2 SetL messages in reverse order

Fig. 4: Message sequence diagram of node deletion

in the case of $u.r = u$), then u is immediately deleted by setting $u.s$ to *out*. Otherwise, u sets $u.s$ as *del* to indicate u is deleting and sends a SetR message to p, which contains q as the new right node, u as the expected current right node, and $u.r_{\mathrm{seq}} + 1$ as the new right sequence number. (The last node is specially handled because otherwise the last node cannot be deleted; it would receive a SetR message from itself while $s = del$ and reply with a SetRNak message.)

The following sequence is similar to the insertion case and thus we omit the details. Note that after the SetRAck and SetL messages are delivered, both $p.r_{\mathrm{seq}}$ and $q.l_{\mathrm{seq}}$ will be correctly updated to $i_2 + 1$.

F. Lookup and Traverse

As a fundamental data structure, distributed doubly linked lists should support traversing. In DDLL, every node that has been inserted during traversal can be looked up by traversing the linked list either rightward or leftward, as described below.

When traversing rightward, every node that has been inserted during traversal can be visited from any inserted node because right links are always correct in DDLL, as we prove later in Theorem 1. When traversing leftward, it should be considered that the left link of a node might not point to the latest left node; when node u visits an inserted node x to fetch the pointer of x's left node w (here we assume iterative routing), there might be some inserted node in interval (w, x). To traverse such a missed node, u should fetch $w.r$ as well as $w.l$ when u visits w. If $w.r$ is not equal to the previous node x, then the node pointed to by $w.r$ is a missed node and u should visit it. Such temporary rightward traversal continues until it encounters a node whose right link points to (or passes over) x. However, such cases should not be common, considering that a left link converges quickly.

It should also be considered that traversing may encounter a deleted node. For example, consider the case where node u visits an inserted node x to fetch a pointer of its right or left node y. When u visits y, y may already have been deleted. Such cases are unavoidable if iterative routing is used. In these cases, u simply restarts traversing from x.

When recursive routing is used and the underlying transport has the FIFO property, it is possible to traverse the linked list without encountering a deleted node, provided that the algorithm is properly modified. For rightward traversal, the SetL message for node deletion should be sent from the deleting node u, rather than u's left node, after u receives the SetRAck message. For leftward traversal, while it is also possible to modify the algorithm (it requires additional messages and we omit the detail due to lack of space), practically it is sufficient for a deleting node to have a grace period after receiving the final SetRAck message. During the grace period, a node forwards received messages to the left node. Because there is always a chance for a node to receive messages after the grace period expires, some retransmission mechanism should be used as well.

As described in Section IV-B, to insert a node u, u needs pointers to the immediate left and right nodes. These nodes can be looked up by traversing the linked list until reaching the node x that satisfies $u \in (x, x.r)$. The immediate left and right node of u are x and $x.r$, respectively. It is not necessary to visit x's right node.

G. Optimization

Let us consider the case where multiple nodes are simultaneously trying to be inserted between the same two nodes. Such massive insertion is seen in some P2P data structures such as skip graphs or Chord$^{\#}$, where not a physical node but each data item is inserted in the structure. In such cases, all but one of the inserting nodes will fail and retry insertion. We describe a method for improving performance in such cases.

Consider the case where node u tries to be inserted between p and q, and p replies with a SetRNak message

to u because of a current right node mismatch. Let us denote p's right node at this moment by x. If $u \in (p, x)$ holds, obviously u should retry insertion between p and x. Thus in this case, u obtains x through the SetRNak message from p and skips the procedure for locating u's next insertion position. If $u \notin (p, x)$ holds, u locates its insertion position by traversing rightward from x. (If p replies with a SetRNak message not because of a right node mismatch, x in the SetRNak message should be nil and u retries normally.) The effect of this optimization is evaluated in Section VI.

H. Handling Failures

In a real network, nodes may fail and messages may be lost. In this subsection, we loosen the initial assumption and allow node failures and message loss and describe how the algorithm should be extended for handling such failures.

We use timeouts for detecting failure. Because failed nodes are indistinguishable from slow nodes, we must allow that nodes may be erroneously suspected to have failed. We consider crash, omission and timing failure. As described in Section II, consistent lookup is impossible in the presence of failure. In DDLL, lookup consistency is preserved only if there is no network partition and no erroneously suspected node.

When a node p fails, the right node of p (denoted by u) conducts the recovery procedure. This is because we would like the sequence number of a SetL message sent to u to be increased monotonically. We discuss the sequence number at recovery in Section IV-H3.

1) Recovery Procedure: Here, we describe the recovery procedure at node u. When $u.s$ is *in*, u periodically executes the following procedure.

1) Find the live node v that is closest to u and on its left. This procedure is called ***findLiveLeft()*** and discussed in Section IV-H2.
2) Obtain $v.r$ and $v.r_{\mathrm{seq}}$ from v.
3) If $(v = u.l \wedge v.r = u \wedge v.r_{\mathrm{seq}} = u.l_{\mathrm{seq}})$, no recovery procedure is required.
4) Otherwise, u tries to connect u and v. Namely, u changes $u.l$ to v, $u.l_{\mathrm{seq}}$ to i, and sends SetR(u, $v.r$, i) to v, where i denotes the new sequence number assigned to the link between v and u. The value of i is discussed in Section IV-H3. Node v handles the SetR message in the same manner described in Fig. 1, with the exception that no SetL message is sent. The SetR message should be extended to indicate whether the message is for recovery or not.
5) If u receives a SetRAck message from v, the recovery procedure completes. If u receives a SetRNak message (which means another node is inserted at the right side of v after u retrieves $v.r$, or v has started its deletion), or does not receive a message within a timeout period, u retries the recovery procedure from the first step after waiting for a time.

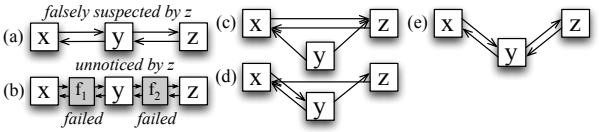

Fig. 5: Recovery from incorrect findLiveLeft() results

2) Finding the Closest Inserted Live Node on the Left: We describe how to implement findLiveLeft(), which finds the closest inserted live node on the left. We assume that each node u maintains a *neighbor node set* N, which contains a sufficient number of pointers to the left-side nodes, including $u.l$. We start from the closest live inserted node found in $u.N$ and traverse the linked list rightward, until encountering a node whose right link either points to a failed node, points to u, or passes over u.

1) v := the closest live node in $u.N$ where $v.s$ is *in*. If no such node exists, $v := u$.
2) if ($v.r$ fails $\vee$ $u \in (v, v.r]$) then return v.
3) $v := v.r$ and go to Step 2.

The procedure above does not always find the closest inserted live node. Consider the case where node z executes findLiveLeft(). Node z may falsely detect an inserted live node y as failed due to a temporary network failure (Fig. 5a). Also, z may not notice an inserted live node y that is sandwiched by two failed nodes and not yet in $z.N$ (Fig. 5b). In such cases, y is excluded from the linked list by the recovery procedure at z (Fig. 5c, which depicts the state after Fig. 5a).

This situation is eventually recovered (if false detection is ceased). When y executes the recovery procedure, y will find the right link of x, the closest inserted live node on the left side, does not point to y. Thus y changes $y.l$ to x and sends a SetR message to x to change $x.r$ to y (Fig. 5d). Then, when z executes the recovery procedure, z will find y by traversing rightward from x and correct the links (Fig. 5e).

3) Sequence Number of a Recovered Link: Here, we discuss the sequence number of a recovered link. We consider the following scenario: Nodes A, B and C ($A < B < C$) are inserted. Let $B.r_{\mathrm{seq}}$ and $C.l_{\mathrm{seq}}$ be i. Then, node X is inserted between B and C, and B fails. Note that $X.r_{\mathrm{seq}}$ is $i + 1$. C starts the recovery procedure before the SetL message from X arrives at C. findLiveLeft() at C fails to find X but finds A, so C sends a SetR message to A to update $A.r$ to C. If

C naively assigns $C.l_{\mathrm{seq}} + 1\ (= i + 1)$ as the sequence number to the link between A and C, both A and X would have the same right node (C) and the same right sequence number ($i + 1$). This implies that sequence numbers of SetL messages subsequently sent to C would no longer be monotonically increased. (Consider the case where another node is inserted between A and C or X and C.)

To make sure that a left link is consistent with the corresponding right link after recovery, we extend a sequence number to the form (g, s). The s part is updated in the same manner described in the previous sections. The g part is initially zero and increased only when a link is repaired. Namely, when node u repairs a link, $(u.l_{\mathrm{seq}}.g + 1, 0)$ is used both as r_{newseq} of the SetR message to v and as $u.l_{\mathrm{seq}}$. When comparing two sequence numbers (e.g. line 41 in Fig. 1), the g part is first compared and if they are same the s part is compared.

This scheme guarantees that once node u recovers a rightward link toward u (i.e., finds a live left node v and updates $v.r$ to point to u), then u only accepts a SetL message for node insertion or deletion on the recovered right link. For example, if C receives the SetL message whose sequence number equals $(0, i + 1)$ from B after incrementing $C.l_{\mathrm{seq}}$ to $(1, 0)$, C ignores the message. Note that X is excluded from the linked list, but it eventually returns to the linked list by executing the recovery procedure at X.

4) Failures on Insertion and Deletion: We first consider the case where node y is trying to insert between two adjacent nodes x and z. Because x might fail, if y receives no response message within a timeout period after sending a SetR message, y must retry the insertion procedure, from locating its insertion position. Here, we have to consider the case where x is in fact alive and messages are simply delayed or lost.

(Case 1) When y retries its insertion and searches for its insertion position, y may notice that y has already been inserted. This happens if the SetRAck message from x is delayed or lost. In this case, y changes $y.s$ to *in* and $y.r_{\mathrm{seq}}$ to zero. Because this $y.r_{\mathrm{seq}}$ (zero) is incorrect (it would have been set by the SetRAck message from x), successive SetL messages to z, caused by another node insertion between y and z for example, may be ignored. This situation is recovered by the recovery procedure of z. Note that continuity of the rightward linked list is still preserved even in this case.

(Case 2) y may receive a delayed SetRAck or SetRNak message from x after y sends another SetR message for a retry. To avoid confusion, y should ignore such a delayed response. This is easily achieved by assigning some ID to a SetR message and including the ID in the corresponding SetRAck and SetRNak message. When a node receives a SetRAck or SetRNak message, the node drops it if the ID does not match with the one that the node sent in the latest SetR message.

We next consider the case where node y, which is inserted between nodes x and z, is trying to delete itself and no response message is received from x within a timeout period. In this case, y acts as if the SetRAck message is received. Here, $x.r$ would point to an deleted node (y) if x does not receive the SetR message from y, but this situation would be recovered by the recovery procedure at z.

V. Correctness

Here, we prove the correctness of the insertion and deletion algorithm, in absence of failure. (To prove the correctness of the recovery procedure is one of our future work.)

Let $m^-(m, x)$ denote the number of incoming messages of type m to node x. We define the following predicates for node x.

$$\begin{aligned}\mathrm{inserted}(x) \equiv ((x.s = ins \wedge m^-(\mathrm{SetRAck}, x) > 0)\\ \vee (x.s = in) \vee (x.s = del \wedge m^-(\mathrm{SetRAck}, x) = 0))\end{aligned}$$

When $\mathrm{inserted}(x) = \mathrm{true}$, we say node x is *inserted*. An *inserted* node is a node that is (at least) inserted to a rightward linked list.

In DDLL, the right link of an inserted node always points to the closest inserted node on the right side. This is expressed in the following theorem.

Theorem 1. *Let V be the set of all nodes (both inserted and uninserted). Then, the following proposition holds.*

$$\forall x \in V, \mathrm{inserted}(x) \rightarrow P(x) \wedge Q(x)$$

where $P(x) = (\nexists y \in V, \mathrm{inserted}(y) \wedge (y \in (x, x.r))$ *and* $Q(x) = \mathrm{inserted}(x.r)$.

Let us begin by proving the following lemmas.

Lemma 1. *At the moment when a node y is successfully inserted at the immediate right of node x, $y \in (x, z)$ holds, where z denotes the right node of x just before y is inserted.*

Proof: We observe the following: (1) An uninserted node becomes inserted only by receiving a SetRAck message. (2) A SetRAck message is sent only in action A_4, in response to a SetR message. (3) An uninserted node sends a SetR message only in action A_2. (4) To make x send a SetRAck message to y, y must send a SetR message to x whose r_{cur} equals z.

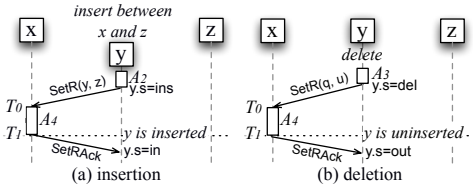

Fig. 6: Figures for the proof of Lemmas 2 and 3. Parameters regarding sequence numbers are omitted.

These observations imply that $p = x$ and $q = r_{\rm cur} = z$ hold in action A_2. Considering this and the condition at line 12, $y \in (x, z)$ holds. ■

Lemma 2. *If a node $y \in V$ tries to insert itself and sends a SetR message to an inserted node $x \in V$, and if Theorem 1 holds at T_0, where T_0 denotes the moment when x receives the SetR message, and if x sends a SetRAck message to y, then $P(x) \wedge Q(x) \wedge P(y) \wedge Q(y)$ holds at T_1, where T_1 denotes the moment when x finishes action A_4.*

Proof: Let us assume that y is trying to insert between x and some node z. According to action A_2, $y.r$ equals z and $r_{\rm cur}$ of the SetR message equals z. Because we assume that y receives a SetRAck message from x, x's right link points to z at T_0. Given that Theorem 1 holds at T_0, no inserted node exists in (x, z) and z is *inserted* at T_0 (Fig. 6a).

Because an action is atomically executed, A_4 should be the only action executed at x between T_0 and T_1. At T_1, because $(m^-(\mathrm{SetRAck}, y) > 0) \wedge (y.s = \mathrm{ins}) \wedge (x.r = y)$ holds, y is *inserted*, and $Q(x)$ holds. Also at T_1, $Q(y)$ holds because $y.r = z$ and z is still *inserted*.

Considering Lemma 1, it is also understood that $P(x)$ holds at T_1 because no inserted node exists in $(x, x.r\,(= y))$ and $P(y)$ holds because no inserted node exists in $(y, y.r\,(= z))$. ■

Lemma 3. *If a node $y \in V$ tries to delete itself and sends a SetR message to an inserted node $x \in V$ and if Theorem 1 holds at T_0, where T_0 denotes the moment when x receives the SetR message, and if x sends a SetRAck message to y, then $P(x) \wedge Q(x)$ holds at T_1, where T_1 denotes the moment when x finishes action A_4.*

Proof: Let us assume that $y.r = z$. According to action A_3, $r_{\rm cur}$ of the SetR message equals y. Because we assume that y receives a SetRAck message from x, x's right link points to z at T_0. Given that Theorem 1 holds at T_0, no inserted node exists both in (x, y) and in (y, z), and z is *inserted* at T_0. (Fig. 6b).

We can say that node z is *inserted* at T_1 by observing the following: (1) To make z *uninserted*, z must execute A_3 and send a SetR message to an inserted node whose right link points to z and receives a SetRAck message. (2) y is the only node whose right link points to z at T_0, because Theorem 1 holds at T_0. (3) Because $y.s = \mathrm{del}$ after y sends a SetR message to x, if y receives a SetR message from z, y sends a SetRNak (rather than a SetRAck) message to z.

At T_1, because $m^-(\mathrm{SetRAck}, y) > 0$ and $y.s = \mathrm{del}$ hold, y is *uninserted* and thus no inserted node exists in (x, z). Also because $x.r = z$ and z is *inserted*, $P(x) \wedge Q(x)$ holds at T_1. ■

Proof of Theorem 1: We observe the following: (1) This theorem is true in the state where only the initial node is inserted. (2) The correctness of the theorem is affected only by node insertion and deletion. (3) Cases where a SetRNak message is sent do not affect the correctness of the theorem. (4) An uninserted node becomes inserted and an inserted node becomes uninserted only by receiving a SetRAck message (except the initial and the last node). (5) A SetRAck message is sent only in response to a SetR message.

These observations indicate that Lemmas 2 and 3 are sufficient to prove this theorem. ■

The next theorem states the correctness of the algorithm with regard to left links. In this theorem, we assume that each action is executed instantaneously (i.e., when a node receives a SetR message, a SetRAck or a SetRNak message is instantly sent).

Theorem 2. *For any inserted node u, if there is no SetL messages to u in transmission, $u.l$ points to* $\mathrm{left}(u)$.

Proof: We observe the following: (1) When a node u becomes *inserted* $u.l_{\rm seq} = v.r_{\rm seq}\ (= 0)$ holds, where $v = \mathrm{left}(u)$ (see Fig. 2). (2) When a node x receives a SetL message, $x.l$ is updated iff the parameter *seq* is larger than $x.l_{\rm seq}$, which is the maximum *seq* that x has received thus far (A_7). (3) Whenever $\mathrm{left}(u)$ is changed, a SetL message is sent to u (A_4).

Therefore, it suffices to show that when $\mathrm{left}(u)$ changes from p to q, the *seq* of the corresponding SetL message sent to u is larger than $p.r_{\rm seq}$. This is easily shown by the following observation: In the case of either insertion or deletion, when $\mathrm{left}(u)$ is changed from p to q, $q.r_{\rm seq}$ is updated to $p.r_{\rm seq} + 1$ and is used as *seq* of the SetL message sent to u (see Figs. 2 and 4). ■

It is also clear that (1) for adjacent nodes v and u ($v.r = u$), $u.l_{\rm seq}$ equals $v.r_{\rm seq}$ if there is no SetL message to u in transmission, and that (2) the duration for which the left link of a node points to an incorrect

node is bound by the max one-way message transmission time.

VI. Discussion and Evaluation

A. Deadlock and Livelock

DDLL is *deadlock-free* because it does not use explicit locking. However, it is not free of *livelocks*. For example, if all nodes try to delete at the same time each receives a SetRNak message, so no node can be deleted. This situation can be easily avoided by using randomness for the retry period.

B. Implementing DHT using DDLL

DDLL can be used as a foundation of ring-based DHTs. When implementing a DHT using DDLL, each node u should be responsible for the range $[u, u.r)$. The reason is that, when u receives a put or get request for some data d, u can check whether u is responsible for d because u always has a correct right link. u accepts the request only if d's key is in $[u, u.r)$ and otherwise rejects it or forwards it to the proper node. In this way, it can be ensured that a request is always directed to the responsible node.

C. Simulation

To evaluate the performance of DDLL, we have implemented a discrete event simulator that simulates the DDLL, Atomic Ring Maintenance (abbreviated as *Atomic*) [7], Li's algorithm for Chord rings [8] and Chord [1]. In the simulation, the following assumptions are made: nodes do not fail, message transmission is reliable, and message order is preserved. (The last condition is required by the Atomic algorithm.) Messages received are processed instantly (zero-time).

We first conducted the following simulation for each algorithm: insert an initial node p and then, insert n nodes simultaneously and measure the time and number of messages required until all links are converged. The keys and insertion order of n nodes are randomly selected. Each node looks up its insertion position by traversing the linked list rightward, starting from p. The number of messages includes messages used for locating insertion position. We vary n from 0 to 100. Each experiment is iterated 50 times to obtain average values. All messages are transmitted in abstract constant time $T = 1$.

We use recursive routing in the traversal; a lookup message is forwarded to a neighbor node until reaching the proper node and the result is sent directly to the originating node. We omit the finger table in Li's algorithm and Chord to evaluate all algorithms under the same conditions. Algorithm-specific details are as follows:

(DDLL) DDLL(Opt) and DDLL(NoOpt) are implemented, which respectively represent DDLL with and without the optimization described in Section IV-G.

(Li's algorithm) In the original algorithm, a *busy* node cannot forward a message rightward because the right link of a *busy* node may be not initialized. To make such forwarding possible, we made a small enhancement, which does not affect the correctness of the algorithm; when a node u is trying to insert between p and q, u sets $u.r$ to q prior to sending a *join* request to q.

(Chord) To reduce the time for completion of insertion, we use 10 as the stabilization period, which is excessive short considering $T = 1$.

In all algorithms except DDLL(Opt), when a node receives a SetRNak or a *retry* message, it retries insertion after waiting some period. From the results of preliminary experiments, we use a random value chosen from $[0, T]$ as a waiting period to reduce the average insertion time. Also, to speed up locating a insertion position for the next retry, we traverse the linked list rightward from the previous predecessor candidate node, except in Atomic (in Atomic, leftward from the previous successor candidate node).

The results are shown in Figs. 7 and 8. DDLL(Opt) outperforms other algorithms in terms of both time and number of messages. Except in Chord, the time required for insertion increases logarithmically. This can be explained as follows: when m nodes simultaneously compete for insertion between the same nodes, one node is successfully inserted and the other $m - 1$ nodes retry. Next time $(m - 1)/2$ nodes compete and thus the number of retries is roughly the logarithm of m. In DDLL(Opt), the average number of insertion attempts also increases logarithmically. At $n = 10$ and $n = 100$, the numbers are around 3.39 and 7.46, respectively.

We next evaluated the performance of DDLL and locking-based algorithms in the environment where nodes have non-uniform network latencies. We classified nodes in two groups N_L and N_H, which respectively represent low-latency and high-latency nodes. We assumed that nodes are connected in a simple star topology where the latency from the center to a N_L-nodes is $0.5T$ and to a N_H-nodes is $2T$. Thus, latencies among N_L-nodes are T, between N_L and N_H-nodes are $2.5T$, and among N_H nodes are $4T$. We fixed the number of nodes to 50 and vary the ratio of high-latency nodes. Other conditions are the same to the first experiment. We measured the insertion time of each node and plotted the 50th and 90th percentiles.

The results are shown in Fig. 9. In general,

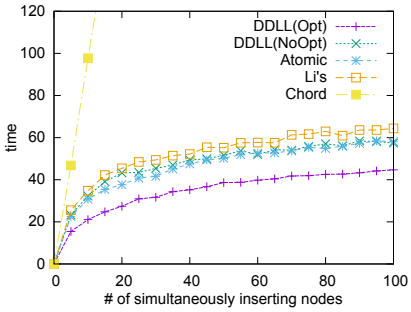

Fig. 7: Node insertion time vs. number of simultaneously inserted nodes. Time in Chord is approximately $9.9n$.

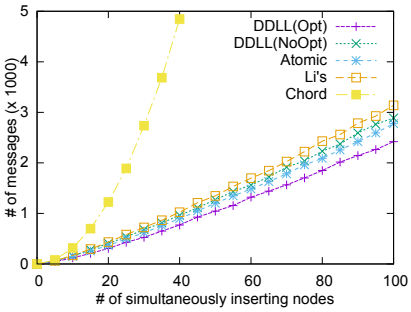

Fig. 8: Total number of messages vs. number of simultaneously inserted nodes. The number of messages in Chord is approximately $2.95n^2$.

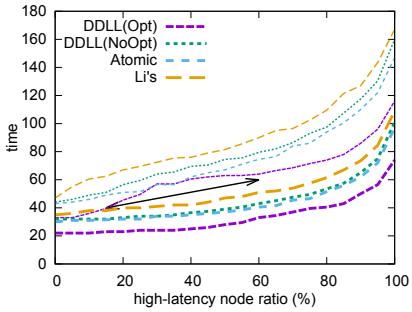

Fig. 9: Node insertion time vs. high-latency node ratio. Thick and thin lines represent 50th and 90th percentiles, respectively.

DDLL(Opt) outperforms in all ratios. We observe a slight increase in the 90th percentiles line of DDLL(Opt) around the ratio between 20% and 60% (depicted by an arrow in the figure). This increase is caused by the reinsertion of high-latency nodes. In DDLL, to insert a node u successfully between nodes p and p's right node q, p's right node should not be changed in the duration from the moment when u obtains the pointer of q from p, to the moment when p receives a SetR message from u. Because the duration of a high-latency node is longer than that of a low-latency node, insertion of a high-latency node tends to be interrupted by low-latency nodes.

D. Implementation Example

We have implemented the DDLL algorithm in the open source P2P platform PIAX [13] and it has been used as a foundation of our fault-tolerant skip graph and Chord# implementations.

VII. Conclusion

We have proposed a novel decentralized algorithm DDLL for constructing a distributed doubly linked list. To the best of our knowledge, it is the first algorithm that is based on conflict detection and sequence numbers. In DDLL, next-node pointers are always correct even while multiple nodes are simultaneously being inserted or deleted and thus, any inserted node can be looked up from any inserted node. The algorithm is simple, thanks to not using distributed locking. It provides recovery procedure from failures. It is efficient, in terms of both number of messages and time required for node insertion. It does not require the FIFO property for the underlying transport so it can be easily implemented using UDP, which is NAT-traversal friendly. We therefore conclude that DDLL is a good foundation for implementing structured P2P systems based on doubly linked lists or bidirectional rings.

Future work includes to prove the correctness of recovery procedure and evaluate the algorithm under various failure situations.

References

[1] 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. on Net.*, vol. 11, no. 1, pp. 17–32, 2003.

[2] T. Schütt, F. Schintke, and A. Reinefeld, "Range queries on structured overlay networks," *Computer Commun.*, vol. 31, no. 2, pp. 280–291, 2008.

[3] J. Aspnes and G. Shah, "Skip graphs," *ACM Trans. on Algorithms*, vol. 3, no. 4, pp. 1–25, 2007.

[4] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: a scalable overlay network with practical locality properties," in *Proc. of 4th conf. on USENIX Sympo. on Internet Technologies and Systems*, 2003, pp. 113–126.

[5] T. Clouser, M. Nesterenko, and C. Scheideler, "Tiara: A self-stabilizing deterministic skip list and skip graph," *Theor. Comput. Sci.*, vol. 428, pp. 18–35, 2012.

[6] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig, "Skip+: A self-stabilizing skip graph," *J. ACM*, vol. 61, no. 6, pp. 36:1–36:26, 2014.

[7] A. Ghodsi, "Distributed k-ary System: Algorithms for distributed hash tables," PhD Dissertation, KTH—Royal Institute of Technology, 2006.

[8] X. Li, J. Misra, and C. G. Plaxton, "Concurrent maintenance of rings." *Distributed Comp.*, vol. 19, no. 2, pp. 126–148, 2006.

[9] A. Shaker and D. Reeves, "Self-stabilizing structured ring topology p2p systems," in *Proc. of 5th IEEE Intl. Conf. on P2P Computing*, 2005, pp. 39–46.

[10] J. Risson, K. Robinson, and T. Moors, "Fault tolerant active rings for structured peer-to-peer overlays," in *Proc. of 30th Ann. IEEE Conf. on Local Computer Networks*, 2005, pp. 18–25.

[11] H. Sundell and P. Tsigas, "Lock-free and practical doubly linked list-based deques using single-word compare-and-swap," in *Principles of Distributed Systems*, ser. LNCS. Springer Berlin Heidelberg, 2005, vol. 3544, pp. 240–255.

[12] Mohamed G. Gouda, *Elements of Network Protocol Design*. John Wiley and Sons, 1998.

[13] Y. Teranishi, "PIAX: Toward a Framework for Sensor Overlay Network," in *Proc. of CCNC'09*, 2009, pp. 1–5.