Atomic Ring Maintenance for Distributed Hash Tables

Ali Ghodsi[1] and Seif Haridi[2]

[1] Swedish Institute of Computer Science (SICS)
Box 1263, 16429 Kista
Sweden
`ali(at)sics.se`

[2] KTH – Royal Institute of Technology
Electrum 229, 16440 Kista
Sweden
`haridi(at)kth.se`

SICS technical report T2007:05

Abstract. This paper provides algorithms to maintain a ring-structure for structured peer-to-peer systems. The algorithms guarantee *consistent* lookup results in the presence of joins and leaves, regardless of at which node the lookup is initiated. Every join and leave event appears as if it happened atomically, thus guaranteeing that lookup results will be the same as if no joins or leaves took place. The ring maintenance algorithms guarantee that no routing failures occur as nodes are joining and leaving. We also show that lookup consistency is impossible to provide given $\diamond\mathcal{P}$ failure detector, and show how the algorithms can be extended to handle failures. The correctness of all the provided algorithms is proven. Previous approaches to this problem either assume a fault-free environment, or have no proof of correctness.

1 Introduction

Distributed Hash Tables (DHTs), which form a large subset of structured peer-to-peer systems, have emerged as distributed data structures suitable for large scale dynamic settings. Thus far, most of the DHTs provide best-effort guarantees. The majority of DHTs are *ring-based*, i.e. assign they assign identifiers to each node and let them form a distributed ring sorted by the identifiers [1–11].

DHTs commonly partition an *identifier space* into n sets, and bijectively map each set to one of the n participating nodes. Each node is then said to be *responsible* for every identifier which is in the set mapped to it. DHTs commonly provide a *lookup operation*, which enables any node to find the node currently *responsible* for any given identifier. A distributed hashtable can then be built by hashing keys onto identifiers, and storing an item at the node currently responsible for that identifier.

Unfortunately, the joining and leaving of nodes affects the *consistency* of lookup results. For each configuration of the system, lookups for an identifier i

initated at different nodes may return different results. This happens, in particular, while nodes are joining and leaving. Consequently, put and get operations on the same key might go to different nodes, in the same configuration. If items are replicated, it becomes difficult to design quorum algorithms which ensure that any two quorums always intersect, as inconsistent lookups can temporarily give rise to "more replicas" than seemingly available.

There are also problems specifically related to nodes leaving the system. In most DHTs, the number of nodes simulatenously leaving has to be lower than some threshold which depends on the rate of topology maintenance, otherwise the ring will break down indefinitely [12]. Furthermore, leaves can lead to *routing failures*, as some pointers become dangling when nodes leave.

The contributions of this paper are threefold. First, we provide algorithms to maintain a ring structure which guarantees *consistent* lookup results in the presence of joins and leaves, regardless of where the lookup is initiated. Every join and leave event appears as if it happend atomically, thus guaranteeing that lookup results will be the same as if no joins or leaves took place. Second, it is guaranteed that no routing failures can occur as nodes are joining and leaving. Third, we show how ring maintenance can be augmented to handle arbitrary additional routing pointers. Thus, lookup consistency can be extended to handle pointers placed according to any of the previously known topologies, such as Plaxton [13], Skip graphs [14], or De-bruijn [15]. As a side effect of our algorithms, there will be no bound on the number of nodes that may simultaneously join or leave the system. We show that lookup consistency is impossible to provide given $\diamond\mathcal{P}$ failure detector, and show how the algorithms can be extended to handle failures. The correctness of all the provided algorithms is proven. Our algorithms are based on simple ideas, and have been implemented in the DKS middleware [16].

2 Atomic Ring Maintenance

Our aim is to provide algorithms that ensure lookup consistency, do not restrict the number of leaves, and guarantee no routing failures. We incrementally attack the problem, starting on a high level, giving the intuition behind our approach before delving in to details.

A simple approach to atomic ring maintenance would be to let every node i *host* a lock L_i, which can only be acquired by at most one node. each node. Each joining and leaving node would then be required to acquire three locks: its predecessor's, its own, and its successor's. After acquiring the locks, the pointers of the respective nodes could be updated to complete a join or a leave operation. Since a join or leave of a node q only requires changes to the pointers of node q, q's predecessor, and q's successor, attempting to lock those three nodes against concurrent modifications would solve concurrency related problems. There is, however, a slightly simpler approach, which will resemble *the dining pilosophers' problem*.

In our join and leave algorithms, the joining or leaving node n will first acquire its own lock L_n, and thereafter its successor's (denoted *n.succ*) lock ($L_{n.succ}$). Only once it has acquired both locks, it can update the relevant pointers. Thereafter it will release both locks. This reduces the number of locks to two, with one of them being a local lock, which can be acquired without the overhead of network communication.

The above scheme will ensure mutually exclusive access to the relevant pointers for the joining or leaving node.

Theorem 1 (Non-interference). *Assume a system, of at least two nodes, with correct pointers. If a node j successfully acquires the locks L_j and $L_{j.succ}$, then j's successor q (j.succ) and predecessor p (j.pred if j is leaving, and j.succ.pred if j is joining) cannot leave the system until the locks are released. Furthermore, no other join or leave operation will affect the pointers p.succ, j.pred, j.succ, and q.pred as long as j holds the locks.*

Proof. We refer to j's successor (*j.succ*) as q. We refer to j's predecessor as p, which is *q.pred* if j is about to join, and *j.pred* if j is about to leave. Assume on the contrary that j's predecessor p is leaving. That would imply that p has acquired the locks L_p and $L_{p.succ}$, where *p.succ* is either j or q depending on whether j is leaving or joining. Either way, it contradicts the fact that node j holds L_j and L_q. Similarly, assume that q is leaving the system, then q must have acquired the locks L_q and $L_{q.succ}$, contradicting that j holds the lock L_q. For the remaining part of the proof, there are two ways in which the pointers *p.succ* and *q.pred* can be altered. Either a node with j as successor tries to join, or a node with q as successor tries to join. Both cases are impossible as the locks L_j and L_q are held by the node j, and hence cannot be acquired by any other node. Node j's *succ* and *pred* pointers can be altered if j gets a new predecessor or a new successor. Both cases are impossible as a new predecessor would have to acquire L_j, and a new successor would have to acquire $L_{j.succ}$, both of which are already held by j. □

If a node is joining, the above theorem would even hold if the system size was 1. That would imply that the joining node j has acquired its own lock, L_j, as well as the lock of the remaining node q in the system. The theorem would be trivially true for that case as there are no other nodes that can interfere with the join operation, and q would not be able to leave as L_q would be held by node j while it is joining.

If the system size is 2 and j is leaving, j's successor and predecessor are the same node. The theorem will still hold, as j will acquire its own lock, as well as its successors, and then complete its leave operation without any interference from any other node.

The similarity to the dining philosophers' problem is obvious. The forks represent the locks, and a joining or leaving node represents a philosopher wanting to eat. We therefore re-use some of the existing solutions to this problem.

One known solution to the dining philosopher's problem is to introduce asymmetry. We propose such a solution to avoid cyclic wait (deadlock), which we call *asymmetric locking*. Let z be the node with the highest identifier. A node k can locally determine if it has the highest identifier if $k > k.succ$. If node z attempts to leave the system, it should first attempt to acquire its successor's lock $L_{z.succ}$, and thereafter its own lock L_z. In any other case, where some node j wants to join or leave, it will first acquire its own lock L_j, and then thereafter acquire its successor's lock $L_{j.succ}$.

So far we have assumed that the pointers in the system are correct and that a node indeed manages to acquire its own lock and current successor's. This need not be the case. If a node ever tries to acquire a lock that is not free, the node will wait until it becomes free and then acquires it. The node which is waiting for a lock L_i will be notified by node i when the lock is free. This requires that node i queues requests to the lock it hosts in a *lock queue*, and notifies and removes one node in the queue each time L_i is released. Two additional operations are needed to ensure that nodes can properly acquire their successor's lock.

A leaving node's lock queue should be transferred to its successor. We first describe a naïve algorithm to achieve this, and later refine it. When a leaving node i has acquired all the relevant locks, it transfers its lock queue to its successor j, which will enqueue the lock queue of i onto its current lock queue. Hence, the elements in the lock queue of j maintain the same position in the queue after i leaves, while an element at position k in the lock queue of i gets position $k + l$, where l is the number of elements in j's lock queue before the merger of the lock queues. Hence, if some node i is waiting for its successor's lock $L_{i.succ}$ to become free, it will be notified even if its successors leave the system.

A joining node might need to take over parts of its successor's lock queue. When a joining node i has acquired all the relevant locks, its successor $i.succ$ transfers its lock queue to i. Node i will then remove from its lock queue every node that has $i.succ$ as its successor. Similarly, node $i.succ$ will remove from its lock queue every node that has i as its successor. More precisely, only nodes in the range $(i.succ, i]$ from $i.succ$'s lock queue are stored in i's lock queue, while only nodes in the range $(i, i.succ]$ from $i.succ$'s lock queue are stored in $i.succ$'s lock queue. Hence, if a node p is waiting for its successor's lock $L_{p.succ}$ and meanwhile gets a new successor q, it will be notified by the new successor q when the lock becomes free.

The above explained scheme will ensure that there will never be a cyclic wait.

Theorem 2. *The join and leave algorithms with asymmetric locking will never deadlock.*

We want to ensure that our solution to satisfy the liveness property that it is starvation free. A liveness property that is desirable for our algorithm is that it is free from *starvation*.

There exist many solutions to the dining philosophers' problem that are starvation free. However, our problem is slightly different from the problem of the dining philosophers, as nodes are joining and leaving. Hence, the number of locks and philosophers is constantly changing. The joining and leaving of nodes can make nodes starve, as we show next.

The problem with the current algorithm is that when nodes leave, their lock queue is merged with their successor's lock queue. If some node is leaving, and its successor's lock queue is non-empty, the nodes in the lock queue of the leaving node will have a worse position after the lock queue of the leaving node is merged with the successor's lock queue. It is therefore conceivable that under conditions of continuous leaves and joins, some node j attempts to acquire a lock and ends up in a lock queue, which gets merged over and over with the successor's lock queue, resulting in node j never acquiring the desired lock.

We will therefore slightly modify our algorithm to ensure starvation freedom. We modify asymmetric locking to ensure that whenever a node attempts to acquire its own lock to leave, no other requests can be enqueued in its lock queue. This is realized by a *forwarding mechanism* as follows. As soon as a leaving node i attempts to acquire its own lock L_i, it will ensure that all further requests to its lock L_i are forwarded to its successor $i.succ$. This forwarding of requests makes sense as a leaving node i's request to acquire L_i indicates that i is about to leave, and requests enqueued after such a request should anyway be handled by i's successor after i has left the system. The full algorithmic specification of the algorithm with asymmetric locking with forwarding mechanism can be found in the accompanying technical report [17].

We have now arrived at the full algorithm for asymmetric locking, as can be seen by Algorithms 1 and 2. Algorithm 1 mainly uses RPC notation, while the parts related to the forwarding mechanism (Algorithm 2) use event notation. The reason for the use of event notation is that it simplifies describing the forwarding mechanism.

The algorithm uses the variable *LockQueue*, which represents a FIFO queue. The *Enqueue*(m) procedure enqueues a request by node m in the lock queue. The *Dequeue*$()$ procedure simply removes the first element from the lock queue.

We now prove that asymmetric locking with the forwarding mechanism is starvation free. For that we need to introduce some simple notation.

Recall that a leaving node has to acquire its own lock and its successor's lock. Similarly, a joining node has to acquire its own lock and its successor's lock. Therefore, a lock queue can contain four types of requests: a request by a leaving node i to acquire its own lock L_i, a request by a leaving node i to acquire the lock of its successor, a request by a joining node i to acquire its own lock L_i, and a request by a joining node i to acquire the lock of its successor.

The lock queue and the four types of requests appearing in it are modeled as follows. The lock queue of a node i is represented by a sequence subscripted by the node identifier. The sequence $\langle\rangle_i$ represents an empty lock queue at node i, which indicates that lock L_i is free. The elements of the sequence are one of the symbols $\{\mathtt{j}, \mathtt{js}, \mathtt{l}, \mathtt{ls}\}$. The left-most element in the sequence is the first element in the lock queue, which represents the request currently holding the lock. The right-most element is the last element in the lock queue.

The symbols have the following meaning:

– The symbol `j` indicates a request by a joining node to acquire its own lock.

Algorithm 1 Asymmetric locking with forwarding

```
 1: procedure $n$.JOIN($succ$)                         ▷ Join the ring with $succ$ as successor
 2:     $Leaving$ :=false                              ▷ Initialize variable
 3:     $LockQueue$.ENQUEUE($n$)                       ▷ Enqueue request to local lock
 4:     $slock$ :=GETSUCCLOCK()
 5:     $pred := succ.pred$
 6:     $pred.succ := n$
 7:     $succ.pred := n$
 8:     $LockQueue := succ.LockQueue$                  ▷ Copy successor's queue
 9:     $LockQueue$.FILTER($(pred, n]$)                ▷ Keep requests in the range
10:     $succ.LockQueue$.FILTER($(n, pred]$)           ▷ Keep requests in the range
11:     $LockQueue$.DEQUEUE()                          ▷ Remove local request
12:     RELEASELOCK($slock$)
13: end procedure

14: procedure $n$.LEAVE()                              ▷ Leave the ring
15:     if $n > succ$ then                             ▷ Asymmetric Locking
16:         $slock$ :=GETSUCCLOCK()
17:         $Leaving$ := true                          ▷ Enable forwarding
18:         $LockQueue$.ENQUEUE($n$)                   ▷ Enqueue request to local lock
19:     else
20:         $Leaving$ := true                          ▷ Enable forwarding
21:         $LockQueue$.ENQUEUE($n$)                   ▷ Enqueue request to local lock
22:         $slock$ :=GETSUCCLOCK()
23:     end if
24:     $pred.succ := succ$
25:     $succ.pred := pred$
26:     $LockQueue$.DEQUEUE()                          ▷ Remove local requst
27:     RELEASELOCK($slock$)
28: end procedure

29: procedure $n$.GETSUCCLOCK()
30:     sendto $succ$.ACQLOCK($n$)
31:     receive LOCKGRANTED() from $m$
32:     return $m$                                     ▷ Return identity of lock host
33: end procedure

34: procedure $n$.RELEASELOCK($dest$)
35:     sendto $dest$.FREELOCK()
36: end procedure
```

Algorithm 2 Asymmetric locking with forwarding continued

```
 1: event $n$.AcqLock($src$) from $m$
 2:     if $leaving$ = true then
 3:         sendto $succ$.AcqLock($src$)
 4:     else
 5:         $LockQueue$.Enqueue($src$)          ▷ Enqueue $src$'s request last
 6:     end if
 7: end event

 8: event when New top element $m$ in $LockQueue$ at $n$
 9:     sendto $m$.LockGranted()
10: end event

11: event $n$.FreeLock() from $m$
12:     $LockQueue$.Dequeue()                   ▷ Remove top element
13: end event
```

- The symbol `js` indicates a request by a joining node to acquire its successor's lock.
- The symbol `l` indicates a request by a leaving node to acquire its own lock.
- The symbol `ls` indicates a request by a leaving node to acquire its successor's lock.

For example, the sequence $\langle \mathtt{js}, \mathtt{js}, \mathtt{ls}, \mathtt{l} \rangle_5$ represents the lock queue at node 5. The first two items (`js`'s) in the lock queue represent requests by some joining nodes to acquire their successor's lock L_5. The third item in the lock queue (`ls`) is a request by the predecessor of 5, which wants to acquire L_5 in order to leave. The last item in the lock queue (`l`) is a request by node 5 to acquire L_5 to leave the system.

With the four symbols we can represent the lock queue at any given node at any time. We shall prove that any element in the lock queue will eventually reach the front of the lock queue, and hence every request to acquire a lock will eventually be granted.

Lemma 1. *If the symbol* `l` *occurs in a sequence, it must be the last element.*

Proof. Assume the symbol `l` occurs in the sequence of node i. The symbol `l` indicates that node i is attempting to leave, and has thus requested to acquire its own lock L_i. As shown by the AcqLock event in Algorithm 1 line 3, any further requests to the lock queue of node i will be redirected to the successor of node i, hence no other requests can be enqueued after enqueueing `l` in the sequence representing the lock queue of node i. Furthermore, there can only be one `l` in any sequence, as a node cannot request to leave while it already has a pending leave request. Therefore `l` must be the last element of the sequence.

□

Theorem 3. *Asymmetric locking with forwarding (Algorithm 1 and 2) is starvation free.*

Proof. Notice that a joining node can always trivially acquire its own lock, since its lock queue is empty. So if the symbol `j` occurs in the sequence of node i, it must be the only symbol in the sequence, since node i is not yet part of the system and i is yet unknown to other nodes. Furthermore, notice that any symbol in a sequence can only improve (move toward the top element) or maintain its position in the queue. It remains to show that any symbol in a sequence will always improve its position in the queue.

We will show that any symbol occurring in any lock queue will eventually reach the top position in the queue. Assume some symbol $s \in \{\mathtt{js}, \mathtt{l}, \mathtt{ls}\}$ occurs in the sequence of some node n. If s is the top element of the sequence we are done; the s request currently holds the lock. Assume s is not the top element of the sequence. According to Lemma 1 the `l` symbol can only be in the last position of a sequence and hence the symbol `l` cannot occur on the left side of symbol s. Hence, only symbols `js` and `ls` can occur on the left of symbol s in any sequence, which implies that the symbol occurring in the top position is either `js`, or `ls`. We inspect three cases separately.

Case 1; Assume n is the node with the highest identifier ($n = z$). Regardless of whether the top element is `js`, or `ls`, it represents a request by some node m to acquire the second and final lock. Hence, m has acquired both required locks and will soon release both of them by calling $Dequeue()$.

Case 2; Assume n is the successor of the node with the highest identifier ($n = z.succ$). If the top element is `js`, the node making the request has acquired both its locks and will eventually be dequeued from the sequence. If the top element is a `ls`, it represents a request made by node z. That implies that z has acquired its first lock, and z will request L_z, which by case 1 will eventually be granted, after which z has both required locks implying that `ls` will eventually be dequeued from the sequence.

Case 3; Assume n is any other node other than z and $z.succ$. This case is the same as case 1.

All three cases show that the top element will repeatedly be dequeued, until the top element becomes s, which completes the proof that any request to a lock will eventually be granted.

□

Drawbacks with Asymmetric Locking There are some performance drawbacks with the proposed asymmetric locking scheme. If neighboring nodes on the ring all try to leave at the same time, it might in the worst case happen that they can only make progress sequentially, one-by-one. Assume a system consisting of 10 nodes with the identifiers 5, 6, $\cdots$, 14. As indicated by Figure 1, nodes 5, 6, 7, 8, 9, might all attempt to leave the the same time. Each of the nodes i successfully acquires its own lock L_i. Thereafter, nodes 5 through 8 attempt to take the lock hosted by their successor, but as the lock is currently held by the hosting node, their request is forwarded until it ends up in the lock queue of node 10. Only node 9 will succeed in acquiring L_{10}, and then successfully leave. Thereafter, node 8, which is now placed on node 10's lock queue, can acquire L_{10} and then leave. This continues sequentially in this manner, until finally node 5 acquires L_{10} and leaves the system. The above situation can be generalized to n neighboring nodes leaving, in which it will take time linearly proportional to n before all of them are done leaving. In addition, if any node wants to join, and its successor is one of the leaving nodes, the joining node has to wait as well.

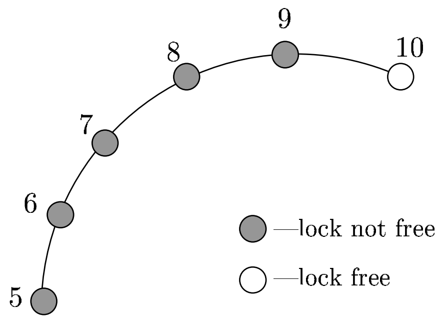

Fig. 1. Consecutive leaves leading to sequential progress. Nodes 5 through 9 are attempting to leave, each has acquired its own lock, and is waiting for its successor's lock. Only node 9 can make progress by acquiring L_{10}, thereafter node 8 makes progress, etcetera.

To circumvent the above situation, we provide another solution which is inspired by the third Coffman condition: preemption of nodes that hold a lock. Since the join/leave algorithms only modify pointers after they have acquired two locks, a node which manages to get one lock, but fails to get a second lock, could release the first lock and retry.

Our *randomized locking* algorithm works as follows. Every joining/leaving node j first attempts to acquire its own lock L_j, and thereafter its successor's lock, $L_{j.succ}$. If a node cannot acquire some lock because the lock is not free, the node releases all the locks it holds and retries to acquire the locks again after waiting a random time.

Aside from the performance reasons previously mentioned, this solution is simpler as it is stateless, and hence simplifies fault-tolerance. For example, if some node fails, all the nodes in its lock queue will be waiting indefinitely for it.

We first state a simple fact, and then show that the algorithm is starvation free.

Theorem 4. *The randomized locking algorithm is free from deadlocks.*

Proof. The third Coffman condition, preemption of locks, is never satisfied. Therefore, the necessary conditions for a deadlock are never satisfied. □

Hence, the randomized locking algorithm ensures that every held lock is eventually released, either because a node has acquired both necessary locks and will release both locks after updating the relevant pointers, or because the node holding the lock was not able to acquire both necessary locks and will therefore release any acquired locks to try again later.

Next, we show that the algorithm is free from starvation assuming some finite bound on the total number of joins and leaves. The assumption is justified because there can only be a finite number of nodes that can contend for the lock at any given point in time.

Theorem 5. *The randomized locking algorithm is free from starvation.*

Proof. Assume the maximum number of nodes that can contend for a lock at any given instant is k. Theorem 4 showed that the lock is always freed, in which case all the nodes race to acquire it. One of them will always succeed. We assume that all nodes contending for a lock have equal probability of succeeding. This is motivated by the random wait in the algorithm.

The probability that a fixed node j is never able to fetch its first lock and starve is:

$$Pr[\text{j } starves] = \lim_{n\to\infty}\left(1-\frac{1}{k_1}\right)\left(1-\frac{1}{k_2}\right)\cdots\left(1-\frac{1}{k_n}\right)$$

Where k_i is the number of contending nodes time i, hence $k_i \leq k$. Therefore,

$$Pr[\text{j } starves] \leq \lim_{n\to\infty}\left(1-\frac{1}{k}\right)^n = 0$$

The above argument shows that every node will eventually get its first lock. The argument can be extended to the second lock as well, as a node that acquired its first lock will contend for the second lock. Even if it is not able to get the second lock, it will be able to eventually acquire its first lock, and again contend for its second lock. Hence, a node keeps contending for its second lock, and will eventually acquire it by the same argument as above.

□

3 Lookup Consistency

The previous section primarily dealt with concurrency control. It showed how concurrent join and leaves could be coordinated to avoid two neighboring nodes in the ring joining and/or leaving at the same time. So far, we have not dealt with the traversal of these pointers, i.e. lookups. While joins and leaves are happening, we would like to make lookups to find the successor of certain identifier.

Correct lookups in the presence of dynamism is not only important for applications using the overlay, it is crucial to make joins work properly. In the algorithms described in the previous section we assumed that a joining node knows its successor. For this assumption to be valid, a joining node needs to acquire a reference to its successor, which it does by making a lookup.

Correctness of lookups will depend on the lookup algorithm, as well as the join and the leave algorithms. So far, we have only explained how a node successfully acquires locks to avoid conflicting updates to pointers. We have to, however, ensure that potential lookups are correct when the *succ* and *pred* pointers are being updated during join and leave operations. Next, we show how a joining or leaving node should update the relevant pointers when it has acquired the necessary locks. Since we assume the relevant locks are acquired, the *succ* and *pred* pointers can be updated without the interference of any other joins or leaves (see the Non-interference Theorem (1)).

3.1 Lookup Consistency in the Presence of Joins

In Section 2 we showed how a node acquires the relevant locks. In this section we describe how a joining node, which has acquired both relevant locks, updates its own, as well as its successor's and predecessor's *succ* and *pred* pointers. We refer to the joining node as q, its predecessor as p, and its successor as r.

Algorithm 3 assumes that some joining node has acquired both relevant locks, and therefore has a correct *succ* pointer. We also assume that its *pred* pointer is set to `nil`. The time-space diagram shown by Figure 2 depicts the same algorithm fully. Time-space diagrams normally only show one out of many possible executions. However, Algorithm 3 has no alternative executions, or interleavings and therefore the time-space diagram contains all information about the algorithm.

As seen by Figure 2, the joining node q sends an UPDATEPRED message to its successor r. The successor r, upon receipt of UPDATEPRED, sets a special boolean variable called JOINFORWARD to `true`, updates its *pred* pointer to point to the joining node q, and sends a JOINPOINT message to the joining node. The receipt of the UPDATEPRED message constitutes a *join point*, which represents that responsibility of the identifiers in the range $(p, q]$ are instantaneously transferred from r to q. The rest of the algorithm is straight forward, as the joining node updates both its pointers, sends an UPDATESUCC message to its predecessor p, which then sends a STOPFORWARDING message to its successor r and updates its successor pointer to point to the newly joined node. Node r sets its special `JoinForward` variable to `false` upon receipt of STOPFORWARDING, and terminates the algorithm by sending FINISH to the joining node. The joining node knows the pointers have been updated correctly when it receives FINISH, and can safely release any held locks.

Any node in the system might do a lookup while nodes are joining. During a join, however, node p's successor pointer might point to either node r or node q. We would like it to point to r before the join point, and to q after the join point. The former case is ensured automatically assuming p's successor pointer was correctly pointing to r before the join operation. The latter case, however, is not necessarily satisfied. We however circumvent the problem by letting r forward requests coming from p ($r.oldpred$) to node q while r's variable $JoinForward$

Algorithm 3 Pointer updates during joins

```
 1: event n.UpdateJoin() from n                ▷ Assuming succ is correct
 2:     sendto succ.UpdatePred()
 3: end event

 4: event n.UpdatePred() from m
 5:     JoinForward := true                    ▷ Forwarding Enabled
 6:     sendto m.JoinPoint(pred)               ▷ Join Point
 7:     oldpred := pred
 8:     pred := m
 9: end event

10: event n.JoinPoint(p) from m
11:     pred := p
12:     succ := m
13:     sendto pred.UpdateSucc()
14: end event

15: event n.UpdateSucc() from m
16:     sendto succ.StopForwarding()
17:     succ := m
18: end event

19: event n.StopForwarding() from m
20:     JoinForward := false                   ▷ Forwarding Disabled
21:     sendto pred.Finish()
22: end event
```

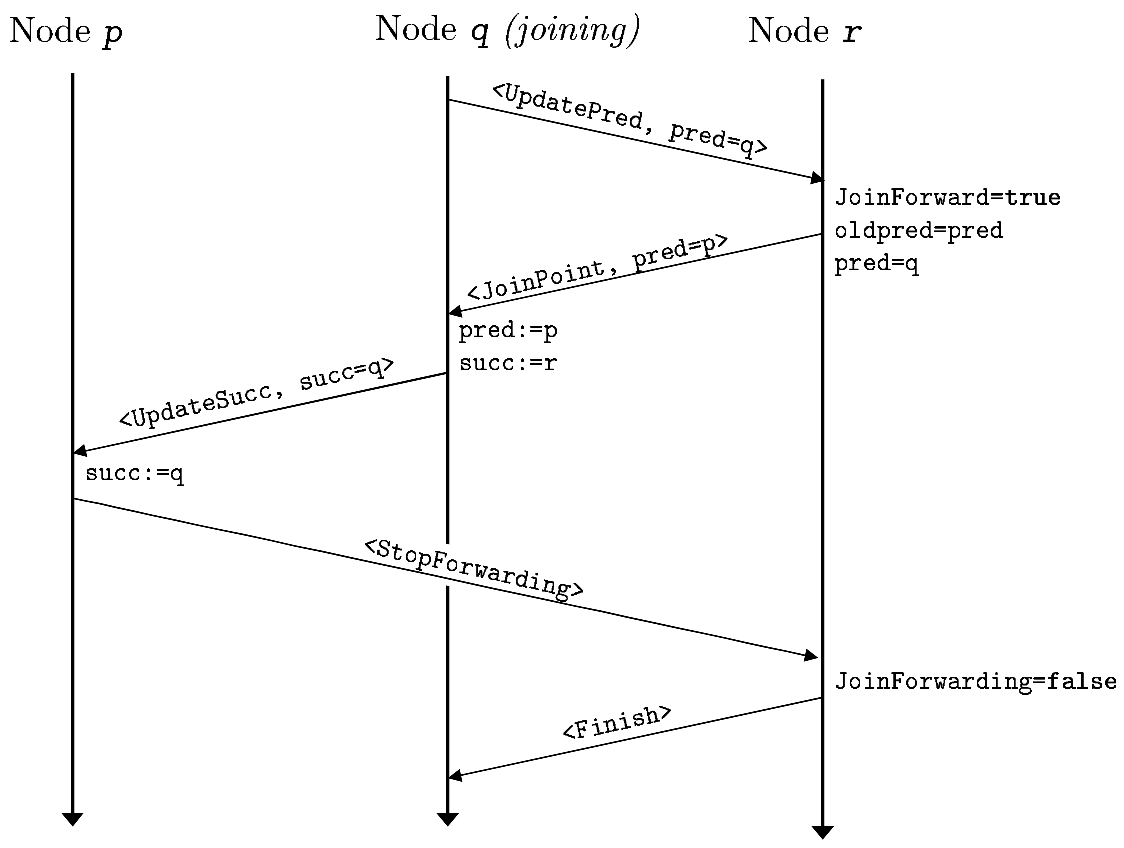

Fig. 2. Time-space diagram showing how a joining node should update the relevant *succ* and *pred* pointers. Node q should have acquired the relevant locks before initiating the algorithm, and it should release the locks when the algorithm finishes.

is true. The FIFO requirement for channels ensures that messages from p pass through node q after the join point.

3.2 Lookup Consistency in the Presence of Leaves

In this section we describe how a leaving node, which has acquired both relevant locks, updates its successor's and predecessor's *pred* and *succ* pointers, respectively. We refer to the leaving node as q, its predecessor as p, and its successor as r.

Algorithm 4 assumes that some leaving node has acquired both relevant locks. The time-space diagram shown by Figure 3 depicts the same algorithm fully.

As seen by Figure 3, the leaving node q starts by setting its boolean *Leave-Forward* variable to `true` and sends a LeavePoint message to its successor r.

This constitutes a *leave point*, which represents that responsibility of the identifiers in the range $(p, q]$ are instantaneously transferred from q to r. The rest of the algorithm is straightforward, as node r updates its predecessor pointer to point to p and informs p to update its successor pointer to point to r. Thereafter, node p sends a StopForwarding message to q. Node q sets its special *LeaveForward* variable to `false` upon receipt of StopForwarding.

The leaving node knows the pointers have been updated correctly when it receives StopForwarding, and can safely release any held locks and leave the system.

Algorithm 4 Pointer updates during leaves

```
 1: event $n$.UpdateLeave() from $n$
 2:     $LeaveForward :=$ true                     ▷ Forwarding Enabled
 3:     sendto $succ$.LeavePoint($pred$)
 4: end event

 5: event $n$.LeavePoint($p$) from $m$
 6:     $pred := p$
 7:     sendto $pred$.UpdateSucc()
 8: end event

 9: event $n$.UpdateSucc() from $m$
10:     sendto $succ$.StopForwarding()
11:     $succ := m$
12: end event

13: event $n$.StopForwarding() from $m$
14:     $LeaveForward :=$false                     ▷ Forwarding Disabled
15: end event
```

As with the join case, any node in the system might do a lookup while nodes are leaving. During a leave, however, node p's successor pointer might point to either node r or node q. We would like it to point to q before the leave point, and to r after the leave point. The former case is ensured automatically assuming p's successor pointer was correctly pointing to r before the leave operation. The latter case, however, is not necessarily satisfied. We however circumvent the problem by letting q forward requests coming from p to node r while q's variable *LeaveForward* is true. The FIFO requirement for channels ensures that messages from p pass through node r after the leave point.

3.3 Data Management in Distributed Hash Tables

So far, we have only mentioned that identifier responsibility moves from one node to another as nodes join and leave. As we previously mentioned, the concept of identifier responsibility can be used to build a distributed hash table (DHT)

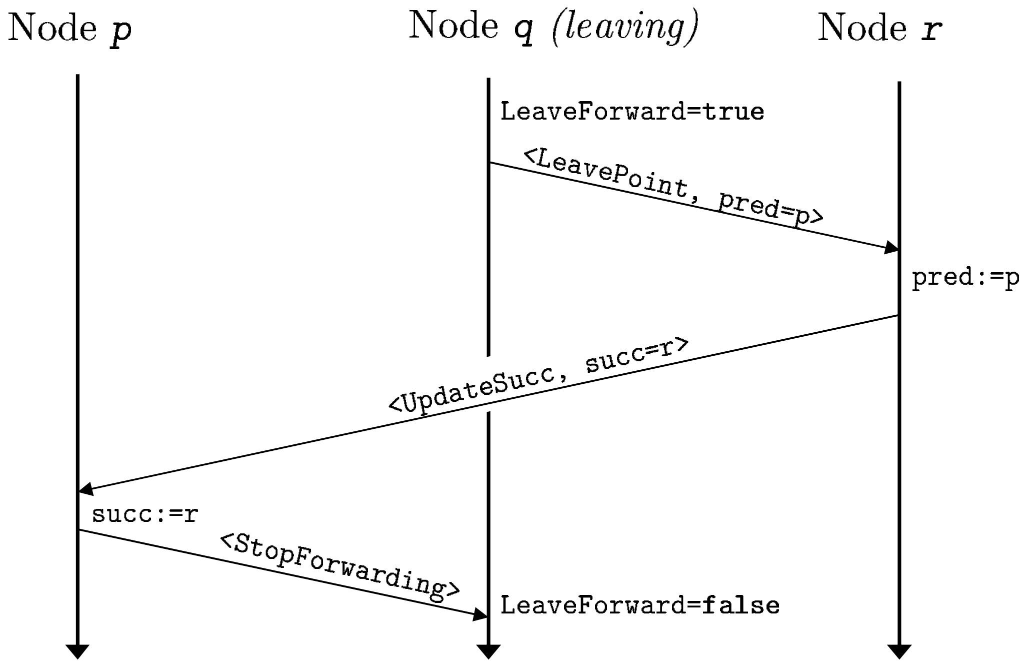

Fig. 3. Time-space diagram showing how a leaving node should update the *succ* and *pred* pointers. Node q should have acquired the relevant locks before initiating the algorithm, and it should release the locks when the algorithm finishes.

abstraction. In such a case, a node might be locally storing data items, whose keys are in the range of the node's identifier responsibility. As identifier responsibility changes, so do the items that a node should be storing.

We first present naïve solution. As a node's responsibility is changed by the sending of a JoinPoint or LeavePoint, items in the changed ranged can be piggy-backed with the message, ensuring that data items are always present at the right place.

As the size of the data items grow, it might be infeasible to piggy-back all necessary items in one message. Nevertheless, what is important is that data responsibility is always consistently defined, which we will show is the case with our algorithms. Another protocol could be used, which lazily, or eagerly fetches items according to the data responsibility. For example, as data responsibility shifts with the sending of a LeavePoint message, the successor of the leaving node could buffer all requests to the identifiers in the changed range, while the leaving node transfers the items over to its successor. Whenever the successor of the leaving node has received all items of the leaving node, it can begin to process the buffered queries. A similar scheme can be used for joins.

3.4 Lookups With Joins and Leaves

The previous sections paved the way for the lookup algorithm, which we now fully define.

Algorithm 5 shows a *transitive lookup*, which goes from node to node until it arrives at the successor of the identifier, in which case it returns directly to the source of the request. The algorithm is initiated by sending a LOOKUP(id, src) message to any node, where *id* is the identifier whose successor is to be found, and *src* is the source node to receive the response.

The algorithm first checks if the $JoinForward$ variable is true, in which case it ensures that messages from its predecessor's predecessor (the *oldpred* variable) are redirected to its predecessor. A similar check is made if the variable $LeaveForward$ is true, in which case the node knows it is leaving, and hence forwards the message to its successor. Note that $JoinForward$ and $LeaveForward$ cannot both be true, as that would indicate that the current node is leaving while its predecessor is joining, which contradicts the locking mechanism described in Section 2.

If both $JoinForward$ and $LeaveForward$ are false, the algorithm first checks to see if *pred* is `nil`. This can happen if a joining node initiates a lookup before reaching its join point, in which case it forwards the query to its successor. Otherwise, if the destination identifier is in its own responsibility, it responds with an answer. In any other case, it forwards the message along the ring to its successor.

Algorithm 5 Lookup algorithm

```
 1: event $n$.LOOKUP($id$, $src$) from $m$
 2:     if $JoinForward =$ true and $m = oldpred$ then
 3:         sendto $pred$.LOOKUP($id$, $src$)                ▷ Redirect Message
 4:     else if $LeaveForward =$ true then
 5:         sendto $succ$.LOOKUP($id$, $src$)                ▷ Redirect Message
 6:     else if $pred \neq$ nil and $id \in (pred, n]$ then
 7:         sendto $src$.LOOKUPDONE($n$)
 8:     else
 9:         sendto $succ$.LOOKUP($id$, $src$)
10:     end if
11: end event
```

Proving Correctness of Lookup Consistency Our consistency requirement will be that at any given time, every identifier will be under the responsibility of exactly one node.

More formally, we say that the *configuration* of the system at any given discretized time, is the nodes in the system and their *succ*, *pred* pointers as well as their variables $JoinForward$, $LeaveForward$, and *oldpred*.

We now construct a function, which given a configuration, mimics the lookup operation of the system. For any given configuration of the system δ, we define a function called $lookup_\delta$ that takes two identifiers k and i, where k is some arbitrary destination identifier and i is the identifier of a node in δ, and returns the identifier of some node in δ. We do not provide the function, but it looks almost identical to Algorithm 5, except that the message passing is replaced with recursive calls.

Our consistency requirement can therefore be defined as:

$$\text{if } lookup_\delta(k,i) = p \text{ and } lookup_\delta(k,j) = q, \text{ then } p = q$$

The above requirement ensures that if the system state is frozen at any given instant, lookups for any identifier will return the same responsible node regardless of the node at which the lookup is initiated.

Theorem 6. *The lookup algorithm satisfies the consistency requirement.*

Proof. We first proceed by induction on joins. The hypothesis is that the consistency requirement is true for a configuration.

First, notice that the first node ever is handled as a special case, where the joining node j sets $j.succ = j$ and $j.pred = j$, making it responsible for all lookups. Hence, the hypothesis is trivially true for the base case.

Assume the hypothesis is true for some configuration δ. Then we show that it will be true for all configurations which result from the steps of the join algorithm. Assume node q is joining, with predecessor p and successor r. Before q joins, $r.pred$ is pointing to p, making $lookup_\delta(k,r) = r$ for all keys k in $(p,r]$, and by the hypothesis $lookup_\delta(k,i) = r$ for all nodes i in δ.

In the first step of q's join, $q.succ$ is set to r and $q.pred$ is set to `nil`. This implies that lookups are unaffected, as any lookup from q will be forwarded to r, and lookups do not terminate at q since $q.pred$ is set to `nil`.

The second step is the join point when r receives UPDATEPRED, sets $r.pred$ to point to q, and enables join forwarding. From thereon, lookups for identifiers $(p,q]$ will return q regardless of where they are initiated. If initiated by r, they are forwarded to q since join forwarding is on. If initiated by q, they will be forwarded to r which redirects it to q, which by the FIFO assumption has set $q.pred$ to p, and hence will return itself as responsible. If they are initiated anywhere else, they will by the induction hypothesis end up at node r, which forwards them to node q, which returns itself as responsible. The next step, the receipt of UPDATESUCC by p, does not affect the results of lookups, but merely incorporates q into the chain of successors. It remains to show that the step where r turns of join forwarding does not affect lookups. By the FIFO assumption, the receipt of STOPFORWARDING ensures that $q.succ = r$, $q.pred = p$, $p.succ = q$, $r.pred = q$, i.e. q is properly incorporated into the ring, therefore forwarding is no longer necessary.

The existence of configurations where the hypothesis is true due to join has been shown. We now show change our hypothesis to be that the consistency requirement is true for δ or δ contains no nodes. Assume the hypothesis is true for δ, we then show that if q (with predecessor p and successor r) leaves, it hypothesis will be true for all intermediary configurations. If q is the last node, then the hypothesis is trivially true. Otherwise, by the hypothesis, all lookups for $(p, q]$ terminate at q with q as responsible. In the first step, leave forwarding is enabled by q. Hence, any lookups terminating in δ at node q, will be forwarded to node r which will, by the FIFO assumption, have $r.pred = p$. Therefore, any queries previously returning q as responsible will return r as responsible. Second step makes $r.pred = p$, ensuring lookups to identifiers in $(p, q]$ reaching r are terminated with r as responsible. Note that the second step causally succeeds the first step, ensuring that requests to q are forwarded to r. The third step ensures that $p.succ = r$, $r.pred = p$, and leave forwarding is enabled, hence there are no pointers to q in the configuration. Finally, q safely disables leave forwarding, as no more lookups could arrive to q as of the third step.

This completes the proof that the consistency requirement is always satisfied. □

4 Optimized Atomic Ring Maintenance

In this section we combine the randomized locking algorithm, and the lookup consistency algorithm, with all required special cases for system sizes less than three and describe the algorithms.

It is possible to combine the asymmetric or randomized locking scheme with the pointer update algorithm (Algorithms 3 and 4) to arrive at a full algorithm. The algorithm can, however, be optimized to consume less messages. This can be realized by a close look at the asymmetric locking algorithm (Algorithm 1). A joining or leaving node has to acquire its successor's lock, which requires two messages. Only thereafter it can update the successor's *pred* pointer, a step which also requires two messages. This section optimizes these two steps such that a successful request to acquire the successor's lock will have the side effect that the successor correctly updates its *pred* pointer.

General Algorithm Description The lock at each node is represented by the variable *lock*, which takes two possible values {`free`,`taken`}, initially set to `free`. Similarly, each node uses two boolean variables called $JoinForward$ and $LeaveForward$, which are initially set to `false`.

Each node also keeps a variable called *status*, which is only used to facilitate the understanding of the algorithm. The *status* variable changes values according to the state machine shown in Figure 4. The state called `inside` indicates that the node is not leaving nor joining, nor is its predecessor leaving. The rest of the states are explained, below, in the informal descriptions of the algorithms.

4.1 The Join Algorithm

We now informally describe the join algorithm, which is given by Algorithms 6 and 7. Throughout the example, we will assume that a node q is joining between a node r and its predecessor p.

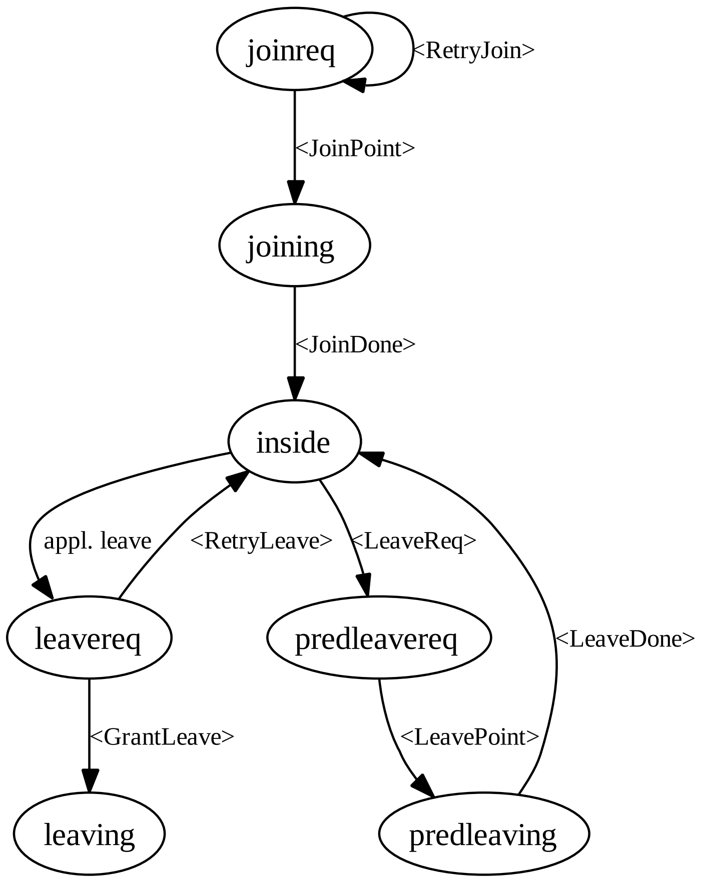

Fig. 4. State transition diagram showing how a nodes status can change for the optimized randomized algorithm. Events indicate received messages, while the states indicate the status of the node.

Initially, a joining node starts with *lock* set to `taken` and *status* set to `joinreq`, indicating that it has acquired the local lock and it is waiting to join. An exception is made if the node is the only node in the system, in which case it initializes its pointers, sets its lock to `free`, and sets *status* to the state `inside`. The next step for the joining node with id q is to send a JoinReq message to the current successor of identifier q. This is trivially done by following the successor pointers until a node r is found where q is an identifier which is under the responsibility of r ($q \in (r.pred, r]$). We are currently not really concerned with the efficiency or the algorithmic details of finding q's successor, but we shall return to this issue later in Chapter **??**.

The successor r of a joining node q will either grant q's request or asks q to retry joining later. The latter case occurs when r's lock is taken, in which case r sends q a RetryJoin message, which results in q waiting a random amount of time before retrying. This scheme can be optimized by letting the successor preempt the retry when its lock becomes free.

If node r grants q's join request, r will immediately set its boolean variable $JoinForward$ to `true` and change the state of its lock to `taken`, indicating that it is locked because its predecessor is joining. It will also save its *pred* pointer in a temporary *oldpred* variable, and change its *pred* pointer to point to the joining node q. Thereafter r will send q a JoinPoint message, which constitutes the *join point*, where the identifiers in the range $(r.oldpred, q]$ are instantaneously transferred to the new node q. Node q updates its successor and predecessor variable whenever it receives the JoinPoint from its successor, and updates its status variable from `joinreq` to `joining`, indicating that the join point has occurred. Hence, both the nodes involved in the move of the join point can determine from their variables if their join point has occurred.

Finally, after receiving the JoinPoint message, the new node q will ask the predecessor to update its *succ* pointer. This is achieved by sending a NewSucc message to the predecessor, which responds by updating its *succ* variable to q and sends a NewSuccAck to its old successor r ($p.succ$), which will free its lock and set its status to `inside`. Thereafter, r sends a JoinDone message to the new node, which finally frees its lock.

As previously described, a node with $JoinForward$ = `true` will redirect messages received from *oldpred* to the new node (*pred*) to ensure that lookups relevant to the new node always end up at the new node after the join point. Hence, lookup consistency is always guaranteed (see lookup consistency in Section 3.4).

A successful execution of a join operation is shown by the time-space diagram shown in Figure 5.

4.2 The Leave Algorithm

We now informally describe the leave algorithm, which is given by Algorithms 8 and 9. Throughout the example, we will assume that a node q is leaving with predecessor p and successor r.

The leaving node q can only initiate a leave request when its lock is free. If it is not, it will wait and retry later. When its lock is free, it initiates the leave operation. If the node is the last node in the system, it will detect that, since its its *pred* and *succ* pointers will be pointing at itself, in which case it can leave unnoticed. If it is not the last node, it starts by sending a LeaveReq to its successor r.

The successor, node r, will only accept a leave request if its lock is free. If it is not, it will send a RetryLeave message, which results in q freeing its lock and waiting a random amount of time before retrying again. If r accepts the request, it sets its lock to `taken` and it changes its status from `inside` to `predleavereq` and sends a GrantLeave message to the leaving node q.

Upon receiving the GrantLeave message, the leaving node sets its variable $LeaveForward$ to `true`, changes its status to `leaving`, and transfers responsibility of all identifiers in $(q.pred, q]$ to its successor r. We will call this the *leave*

Algorithm 6 Optimized atomic join algorithm

```
1:  event $n$.JOIN($e$) from $app$
2:      if $e =$ nil then
3:          $lock :=$ free
4:          $pred := n$
5:          $succ := n$
6:      else
7:          $lock :=$ taken
8:          $pred :=$ nil
9:          $succ :=$ nil
10:         $status :=$ joinreq
11:         sendto $e$.JOINREQ($n$)
12:     end if
13: end event

14: event $n$.JOINREQ($d$) from $m$
15:     if $JoinForward$ and $m = oldpred$ then
16:         sendto $pred$.JOINREQ($d$)                              ▷ Join Forwarding
17:     else if $LeaveForward$ then
18:         sendto $succ$.JOINREQ($d$)                              ▷ Leave Forwarding
19:     else if $pred \neq$ nil and $pred \neq n$ and $d \in (n, pred]$ then
20:         sendto $succ$.JOINREQ($d$)
21:     else
22:         if $lock \neq$ free or $pred =$ nil then
23:             sendto $m$.RETRYJOIN()
24:         else
25:             $JoinForward :=$ true
26:             $lock :=$ taken
27:             sendto $d$.JOINPOINT($pred$)
28:             $oldpred := pred$
29:             $pred := d$
30:         end if
31:     end if
32: end event
```

Algorithm 7 Optimized atomic join algorithm continued

```
1: event $n$.JoinPoint($p$) from $m$
2:     $status$ := joining
3:     $pred := p$
4:     $succ := m$
5:     sendto $pred$.NewSucc()
6: end event

7: event $n$.NewSucc() from $m$
8:     sendto $succ$.NewSuccAck($m$)
9:     $succ := m$
10: end event

11: event $n$.NewSuccAck($q$) from $m$
12:     $lock$ := free
13:     $JoinForward$ := false
14:     sendto $q$.JoinDone()
15: end event

16: event $n$.JoinDone() from $m$
17:     $lock$ := free
18:     $status$ := inside
19: end event
```

point. This is done by sending a LeavePoint message to the successor r, which reacts by changing its status from `predleavereq` to `predleaving` and setting its *pred* pointer to the leaving node's predecessor, p.

After the leave point, r asks its new predecessor to update its *succ* pointer to point to r by sending a UpdateSucc message to p. Node p, reacts by sending UpdateSuccAck to its current successor q, and thereafter updating its *succ* pointer to point to r. The leaving node q knows by the receipt of UpdateSuccAck that its predecessor its no longer going to forward any queries to it, and can therefore send a LeaveDone message to its successor r and leave the system.

Finally, node r receives LeaveDone, frees its lock, and changes its status to `inside`, to allow new join or leaves, either from itself, its predecessor, or from new nodes.

As with joins, misdirected messages are redirected. In particular, any messages received will be redirected to the successor of the leaving node to ensure lookup consistency (see lookup consistency in Section 3.4).

A successful execution of a leave operation is shown by the time-space diagram shown in Figure 6.

Algorithm 8 Optimized atomic leave algorithm

```
 1: event $n$.LEAVE() from $app$
 2:     if $lock \neq$ free then                    ▷ Application should try again later
 3:     else if $succ = pred$ and $succ = n$ then
                                                    ▷ Last node, can quit
 4:     else
 5:         $status :=$ leavereq
 6:         $lock :=$ true
 7:         sendto $succ$.LEAVEREQ()
 8:     end if
 9: end event

10: event $n$.LEAVEREQ() from $m$
11:     if $lock =$ free then
12:         $lock :=$ taken
13:         sendto $m$.GRANTLEAVE()
14:         $state :=$predleavereq
15:     else if $lock \neq$ free then
16:         sendto $m$.RETRYLEAVE()
17:     end if
18: end event

19: event $n$.RETRYLEAVE() from $m$
20:     $status :=$ inside
21:     $lock :=$ free                              ▷ Retry leaving later
22: end event

23: event $n$.GRANTLEAVE() from $m$
24:     $LeaveForward :=$ true
25:     $status :=$ leaving
26:     sendto $m$.LEAVEPOINT($pred$)
27: end event
```

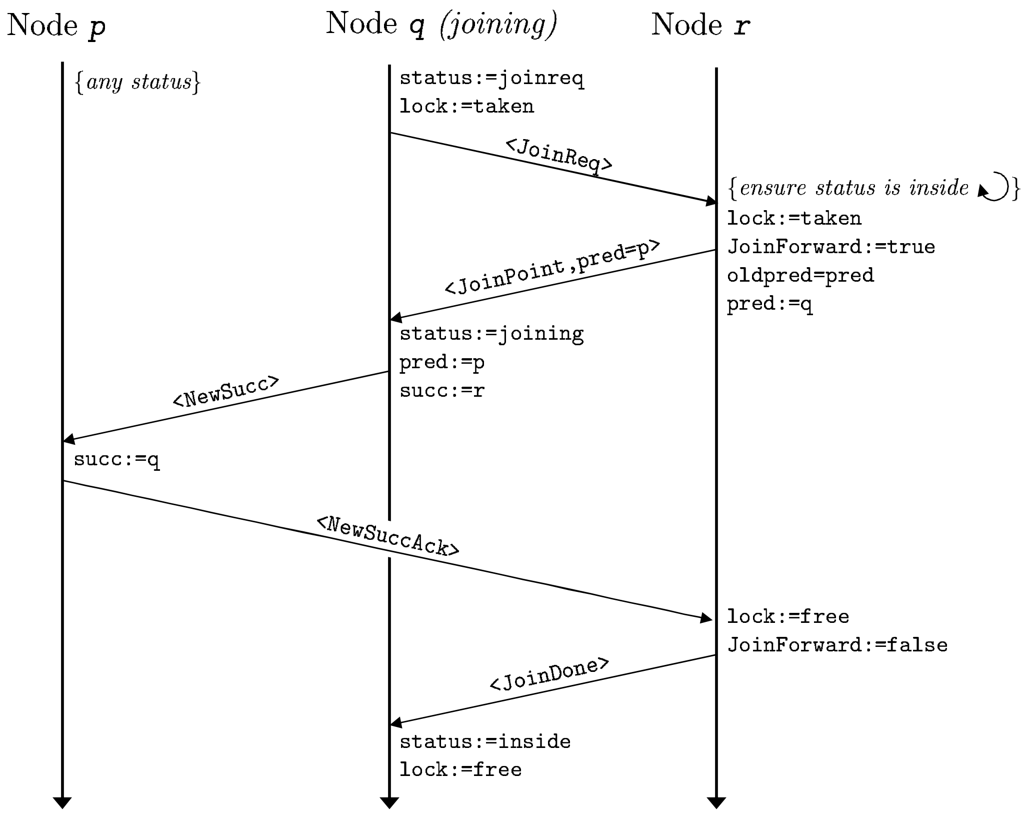

Fig. 5. Time-space diagram of the successful join of a node.

5 Dealing With Failures

Our purpose is to build a system which functions in an asynchronous network, such as the Internet. It is therefore natural to aim at providing lookup consistency in the presence of crash failures and network partitions.

Unfortunately, we will show that it is impossible to implement a system which provides lookup consistency in an asynchronous network with network partitions. The result is related to what is known as *Brewer's Conjecture* [18], which states that it is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

Algorithm 9 Optimized atomic leave algorithm continued

```
1: event $n$.LeavePoint($q$) from $m$
2:     $status$ :=predleaving
3:     $pred := q$
4:     sendto $pred$.UpdateSucc()
5: end event

6: event $n$.UpdateSucc() from $m$
7:     sendto $succ$.UpdateSuccAck()
8:     $succ := m$
9: end event

10: event $n$.UpdateSuccAck() from $m$
11:     sendto $succ$.LeaveDone()                    ▷ Leave the system
12: end event

13: event $n$.LeaveDone() from $m$
14:     $lock$ := free
15:     $status$ := inside
16: end event
```

The conjecture has been formalized and proven by Gilbert and Lynch [19]. We will take consistency to be lookup consistency as we defined in Section 3.4. We next describe the term availability and partition-tolerance.

By availability, it is meant that every request received by a non-failed node must eventually result in a response. This requirement is quite weak, as it does not require a response within any time bounds, but rather requires that a response comes back at some point in time. Hence, it is a natural termination requirement for any distributed service.

Partition tolerance[3], means that the nodes in the system can become partitioned into different components, in which nodes in different components cannot communicate.

We now give the impossibility result, which even allows for inconsistent lookups while the network is partitioned. The proof makes certain assumptions about lookups, because it is trivial to create a system which guarantees lookup consistency by always returning 0 as the result of any lookup. More precisely, we assume that a lookup returns the identity of one of the nodes that is in the same partition as the initiator of the lookup, and that the identity of all nodes is unique. The Chord lookup function, which returns the successor of the identifier satisfies this requirement, given that the responsible node is in the same network component as the lookup initiator.

[3] Gilbert and Lynch model a partition as a network which is allowed to lose arbitrarily many messages sent from one node to another. Hence, a network partition means that messages from the nodes in one component to another are dropped.

Theorem 7. *It is impossible in the asynchronous network model to provide a ring-based structured overlay network that guarantees the following properties:*

- *Lookup consistency in every network component*
- *Availability*
- *Partition tolerance*

Proof. The proof proceeds by contradiction. Assume there exists a system which guarantees availability, partition tolerance, and provides lookup consistency in every network component.

Assume a configuration C of a correct ring consisting of the nodes 0, 1, 2, 3, 4, 5. Assume the network partitions the nodes into the following two components $A = \{2, 3, 4\}$ and $B = \{0, 1, 5\}$.

The system still needs to provide availability. Hence, a lookup for identifier x in component A needs to return an identifier $i \in A$. Therefore, some operations O_A will take place on the nodes in A which adapt the pointers in A, such that the lookup returns an identifier $i \in A$. Similarly, a lookup for identifier x in component B needs to return an identifier $i \in B$. Therefore, some operations O_B will take place on the nodes in B which adapt the pointers in B, such that the lookup returns an identifier $i \in B$. We refer to the resulting configuration after all operations O_A and O_B as D. We now construct an execution starting in C, in which no partition takes place, where all the operations O_A take place first, and thereafter all the operations O_B take place. The asynchrony in the network permits delaying all messages between the two components long after the operations O_A and O_B are finished, making it appear as if there is a network partition. This execution is indistinguishable from the one in which the network partitioned. Hence, the system will end up in configuration D. Configuration D gives inconsistent lookups, as lookups for x initiated by a node in A will result in a different answer than lookups for x initiated by a node in B. More precisely, $\mathit{lookup}_D(x, i) \neq \mathit{lookup}_D(x, j)$ for $i \in A$ and $j \in B$. Since there only exists one network component, this contradicts the existence of a system which gives the assumed guarantees. □

Note that the impossibility result shows that lookup consistency is not possible in an asynchronous network which partitions. But perhaps lookup consistency is possible in an asynchronous network with failures, but without partitions. We do not know the answer to this. But the following observation makes us pessimistic.

We use failure detectors to detect and recover from failures. Nevertheless, any algorithm which attempts to detect failures in an asynchronous network risks inaccurately suspecting the failure of a correct, albeit slow, node [20]. The reason for this is that if this was not the case, the failure detector could be used to solve the *consensus problem* in an asynchronous network with failures, which is known to be impossible to solve [21]. Hence, our system may very well behave as follows. Assume a correct ring consisting of the nodes 0, 1, 2, 3, 4, and 5. At some point, node 2 inaccurately suspects that its predecessor 1 has failed, and node 4 inaccurately suspects that its successor 5 has failed. Similarly, node 1 inaccurately suspects its successor 2 has failed, and node 5 inaccurately suspects its predecessor 4 has failed. The system has partitioned into two parts, one containing $\{0, 1, 5\}$, and one containing $\{2, 3, 4\}$. Hence, our system will end up in the counter example used by the proof of the impossibility result. Note, that the mimicking of a network partition when using inaccurate failure detectors can occur in other topologies than the ring topology. Assume that such mimicking can be long lasting, and assume that availability requires a response before the ostensible partition recovers. Then a direct consequence of the theorem is that it is impossible to build a system which always guarantees lookup consistency and availability with inaccurate failure detectors.

As consequence of this, our goal will be to provide *eventual* lookup consistency in the presence of failures. Thus, we cannot guarantee lookup consistency when failures are detected, but as the network eventually becomes quiescent, we provide lookup consistency.

5.1 Periodic Stabilization and Successor-lists

In this subsection we show how the atomic ring maintenance for joins and leaves is modified to handle failures. This work relies on much work previously done by the authors of Chord. For a thorough reference, please refer to the Chord technical report [22].

The goal of periodic stabilization is to ensure that the pointers always eventually form a correct ring. However, the algorithms we have described make use of locking to guarantee lookup consistency. The atomic ring maintenance algorithms can therefore block if a node fails. Hence, we propose small modifications to the algorithms. Our goal will be to ensure that every lock in the system is eventually released. Periodic stabilization will take care of the rest, by ensuring that a correct ring is eventually formed. Hence, the system will eventually form a correct ring and all locks will eventually be released.

Next, we shortly describe the Chord protocols for periodic stabilization and the maintenance of successor-lists, and thereafter show our modifications.

Periodic stabilization, as we described in Section **??**, has two purposes: incorporate new nodes into the ring and remove failed nodes from the ring. It, however, does that by relying on successor-lists, as we described in Section **??**. But the successor-lists themselves may be incorrect due to joins and leaves, making the actions of periodic stabilization erroneous. For example, if a node p detects that its successor has crashed, it replaces it with the first alive entry q in its successor-list. Since the successor-list might be out of date, some node other than q might be the true successor of q. Hence, stabilization is done periodically to ensure that the ring is *eventually* correct.

The periodic stabilization protocol achieves its goals by striving to ensure that $p.succ.pred = p$ for any node p. This is done by two mechanisms: the *FixSucc* mechanism and the *FixPred* mechanism.

Informally, the *FixSucc* mechanism periodically moves the successor pointer of a node to the closest alive node in clockwise direction. This is partly achieved by the conditional of the STABILIZE procedure in Algorithm **??**, which updates the *succ* pointer at p if it finds that the successor's *pred* pointer points to a closer node than p's current *succ* pointer.

Informally, the *FixPred* mechanism periodically moves the predecessor pointer of a node to the closest alive node in anti-clockwise direction. This is partly achieved in the conditional of the NOTIFY procedure in Algorithm **??**, which updates the *pred* pointer at q if it finds that a node whose *succ* pointer is pointing at q is closer than q's current *pred* pointer.

Algorithm 10 Periodic stabilization with failures

```
 1: procedure $n$.CHECKPREDECESSOR($p$)              ▷ Locally called periodically
 2:     if $IsAlive(pred)$ = false then
 3:         $pred$ := nil
 4:     end if
 5: end procedure

 6: procedure $n$.STABILIZE()                         ▷ Locally called periodically
 7:     try
 8:         $p := succ$.GetPredecessor()
 9:         if $p \neq$ nil and $p \in (n, succ]$ then
10:             $succ := p$
11:         end if
12:         $slist := succ$.GetSuccList()
13:         $succlist := succ + slist$                 ▷ Prepend $succ$ to $slist$
14:         $succlist := trunc(succlist, k)$           ▷ Right-truncate to fixed size $k$
15:         $succ$.Notify($n$)
16:     end try catch(RemoteException)
17:         $succ := getFirstAliveNode(succlist)$      ▷ Get closest alive node
18:     end catch
19: end procedure

20: procedure $n$.GETPREDECESSOR()
21:     return $pred$
22: end procedure

23: procedure $n$.GETSUCCLIST()
24:     return $succlist$
25: end procedure

26: procedure $n$.NOTIFY($p$)
27:     if $pred$ = nil or $p \in (pred, n]$ then
28:         $pred := p$
29:     end if
30: end procedure
```

Algorithm **??** does not suffice to achieve a correct ring in presence of leaves and failures, because the listed algorithms only ensure that a node points to the closest node, not to the closest *alive* node as required. For example, assume a system with correct pointers where node 10's successor is node 20, whose successor is node 30. If node 20 leaves the system or fails, the STABILIZE procedure at node 10 will fail to contact its successor to change its *succ* pointer to 30. This is therefore remedied as follows. If a node detects that its successor is no longer present, it replaces it with the first alive entry, f, in its successor-list. Even if f is not the correct successor of that node, the *FixSucc* mechanism will update *succ* such that it eventually points to the closest successor.

The above amendment will not ensure that the *pred* pointer always points to the closest alive predecessor. For example, assume a system with correct pointers where node 10's successor is node 20, whose successor is node 30. Assume node 20 leaves the system or fails, and the *FixSucc* mechanism correctly updates 10's *succ* pointer to 30. Next time node 10 invokes the NOTIFY procedure at node 30, the conditional will fail and node 30's *pred* pointer will continue to point at node 20. This is remedied by setting the *pred* pointer to `nil` if it is detected that the predecessor is no longer present. The conditional in the NOTIFY procedure is changed such that the *pred* pointer is always updated if it has value `nil`.

The successor-list at each node is maintained periodically as well. Every node periodically makes sure that its successor-list gets updated by copying its successor's successor-list, prepending the successor in the beginning of the successor-list, and truncating the list to a fixed size.

The above described *FixSucc* and *FixPred* mechanisms, as well as the maintenance of the successor-lists, are listed by Algorithm 10.

In the Chord technical report [22], it is shown that periodic stabilization ensures that any interleaved sequence of joins and leaves will eventually result in a ring where $p.succ.pred = p$. For self-sufficiency, we include some of those theorems.

Theorem 8 (from [22]). *If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the succ pointers will form a cycle on all the nodes in the network.*

The above theorem can be extended to *pred* pointers as well.

Corollary 1. *If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the pred pointers will form a cycle on all the nodes in the network.*

Proof. By Theorem 8 the succ pointers will form a cycle on all the nodes in the network. The NOTIFY procedure just maintains the invariant that if a node p correctly points at its successor q, then q's *pred* pointer will point back at p. Hence, the *pred* pointers will also form a cycle on all nodes in the network. □

The size of the successor-list is usually set to be $\log_2(n)$, where n is the number of nodes in the system. Since, n is not globally known, it is either estimated or sometimes set to be the maximum number of nodes that could exist at any given time ($n = 2^{32}$ for every IP address). The reason for this is that it is proven that even if nodes would fail with probability 0.5, every node would still have some alive node in its successor-list. This result is proven, to varying degree of rigor, elsewhere [22, 2]. Hence, with an adequate size of successor-lists, the system remains connected in the presence of failures.

Theorem 9 (from [22]). *If we use a successor-list of length $r = O(\log N)$ in a network where every successor-list is correct, and then every node fails with probability $1/2$, then with high probability a lookup returns the closest living successor to the query key.*

We note that it is theoretically possible to construct a *loopy ring*, where $u.succ.pred = u$ for every node u, but where there exists a node v with an identifier between u and $u.succ$ (see Chapter **??**). Periodic stabilization cannot rectify such a ring. But since its not known how such a loopy ring can occur, we ignore it in the rest of this chapter.

5.2 Modified Periodic Stabilization

Previous section showed that the periodic stabilization algorithm, with the *FixSucc* and *FixPred* mechanisms, handles both joins and failures. But the atomic ring maintenance already takes care of joins and leaves. Therefore, a viable question is whether a simpler algorithm than periodic stabilization, which only deals with failures, can be used in conjunction with atomic ring maintenance. Nonetheless, any algorithm which attempts to detect failures in an asynchronous network risks inaccurately suspecting the failure of a correct, albeit slow, node. Hence, in addition to atomic ring maintenance, the system needs to detect and recover from failures, as well as incorporate nodes which have been inaccurately classified as failed. Thus, we will use both the *FixSucc* and *FixPred* mechanisms of periodic stabilization.

The atomic ring maintenance algorithms will block if a node fails before the algorithm has terminated. The reason for this is that locks acquired by failed nodes will never be released. We propose a simple solution, which ensures that all locks eventually get released. Our first assumption is that periodic stabilization is run whenever a node's lock is free. Similarly, a precondition for the n.NOTIFY procedure is that node n's lock is free, otherwise it will not modify its *pred* pointer.

Before we describe how to deal with failures, we describe the philosophy behind it. Rather than checking whether a predecessor or a successor has failed, we use timers which when expired lead to the locks being released. In other words, locks are only leased for a certain amount of time. The reason why we use leased locks is that it guarantees that the locks are eventually released. There are several pitfalls in relying on detecting the failure of a successor or predecessor, rather than using timeouts as we propose. One reason is that a predecessor or successor might be alive, even though it never sends the final message that releases the lock. The reason for this could be a bug in the program. Moreover, it is not difficult for an adversary to make a client which acquires a lock, which it never releases.

Since we are using timeouts, it could always be that a timeout is premature, which results in several different join and leave operations getting intertwined. For example, some node might preemptively release a lock it is hosting because of a timeout. Thereafter, its lock might be acquired by some other node. By that time, the node which in the first case acquired the lock might send, unaware of the preemptive release, some message according to the algorithm, which affects the latter operation. Therefore every node should always have as a precondition that the received message is in accordance with its lock. For example, a NewSuccAck message should always be ignored if the lock is free.

Furthermore, each joining and leaving node always attaches a random number to their leave or join operation. We refer to this as the *operation number*. This number is piggy-backed in all messages that have to do with the join or leave operation. Whenever the lock hosted by a node is acquired, the hosting node stores the operation number in a *opnum* variable. Whenever a node receives a message while its lock is not free, it ensures that *opnum* is equal to the operation number in the message, otherwise the message is ignored.

The join algorithm is modified, such that the successor of a joining node also piggy-backs its successor-list with the JoinPoint message, such that the joining node can initiate its own successor-list.

Our goal is to ensure that a node whose lock is acquired, ensures that its lock is eventually released. This is achieved by every node i starting a timer as soon as the lock it is hosting, L_i, is acquired. The timer is turned off as soon as L_i becomes free. If the timer expires, the node simply changes the state of its lock to free, and sets $JoinForward$ and $LeaveForward$ to `false`. If a joining node's timer expires and $succ =$ `nil`, then it restarts the join procedure until it gets its successor pointer. If a leaving node's timer expires, it simply leaves the system unnoticed.

We believe that the above algorithm will ensure eventual lookup consistency, which we motivate informally in the following. If no timeouts occur, the system will be the one described without periodic stabilization, and hence will provide lookup consistency. Hence, we turn to the case were timeouts occur. Because of timeouts, every lock is eventually released and the $JoinForward$ and $LeaveForward$ variables are set to `false`. This has two consequences. First, the node will start periodic stabilization. Second, it will ignore any remnant messages from any interrupted join or leave operation. If a timeout occurs, it either occurs at the successor of a joining or leaving node.

If a timeout occurs at the successor of a joining or leaving node, it will set its lock to free, making it start periodic stabilization. If the predecessor has indeed failed, periodic stabilization will recover from the crash failure, and the relevant locks will eventually be released, in which case we are back to a correct system state, with guarantees lookup consistency. If the timeout is premature, and the predecessor is a leaving node, it will eventually timeout and leave unnoticed, which makes this case identical to the one where the predecessor indeed has failed. If the timeout is premature, and the predecessor is a joining node, periodic stabilization will eventually correct the joining node's *succ* pointer, provided that the joining node has a successor-list, which we assume it has acquired at the same time as it initially acquired its successor's address. Thereafter, the *FixPred* and *FixSucc* mechanisms will incorporate the new node into the ring.

If a timeout occurs at a joining node there are two cases, depending on if $succ =$ `nil`. If the joining node has not set its *succ* pointer, which is required for periodic stabilization, it will restart the join and eventually get a correct successor. If $succ \neq$ `nil`, all locks will eventually be released and periodic stabilization will incorporate the new node into the ring, since it has a *succ* pointer and a successor-list.

If a timeout occurs at a leaving node, it will leave, making it effectively a failure. Eventually all locks will be released, and periodic stabilization will rectify all pointers pointing at the absent node.

6 Related Work

Li, Misra, and Plaxton [23–25] independently discovered a similar approach as us. The advantage of their work is that they use assertional reasoning to prove the safety of their algorithms, and hence have proofs that are easier to verify. Consequently, their focus has mostly been on the theoretical aspects of this problem. They assume a fault-free environment, and do not use their algorithms to provide lookup consistency. Furthermore, they cannot guarantee liveness, as their algorithm is not starvation-free.

In the position paper by Lynch, Malkhi, and Ratajczak [26], it was proposed for the first time to provide atomic access to data in a DHT. They provide an algorithm in the appendix of the paper for achieving this, but give no proof of its correctness. In the end of their paper they indicate that work is in progress toward providing a full algorithm, which can also deal with failures. One of the co-authors, however, has informed us that they have not continued this work. Our work can be seen as a continuation of theirs. Moreover, as Li *et al.* point out, Lynch *et al.*'s algorithm does not work for both joins and leaves, and a message may be sent to a process that has already left the network [23].

The problem of concurrently updating linked lists and other data structures has been studied in the context of *lock-free* algorithms for *shared-memory multiprocessors* [27, 28]. In this context a data structure resides in the shared memory of a computer, but the individual processors strive to correctly update the data structure concurrently without using locks, while guaranteeing that some processor always makes progress in updating the structure. The context is, however, different, which has led us to believe that these results are not directly applicable to our problems. First, failures in such contexts imply that individual processors have failed, while the memory storing the data structure is intact. This is not the case in distributed systems, where the data structure is distributed over many nodes, each holding part of the data structure in their local memory. Furthermore, the mentioned research provides lock-free implementations of singly-linked lists, while our data structure is a doubly-linked list. We believe that this subtle difference significantly complicates the problem.

The dining philosophers' problem has been widely studied as we previously mentioned. A widely adopted solution to the problem is to use randomization as suggested by Lehmann and Rabin [29]. They propose that each philosopher randomly choose whether to first pick right or left fork. This solution can, however, lead to a deadlock when the system size is small, which is the case at some point for every DHT. For example, if there are two nodes in the system and both pick left fork first, there will be a deadlock.

7 Conclusion

Even though we earlier had worked on the problem of providing lookup consistency, we became seriously aware of the problems during a joint project at SICS. DKS was being coupled with a decentralized authorization server called Delegent, which was storing digital certificates and access policies into the DHT provided by DKS. Some developers noticed strange behavior, when nodes were joining and leaving, some lookups would temporarily report inconsistent results, depending on where they were issued. This motivated us to look into the issue of lookup consistency, as nodes were joining and leaving the overlay network.

Our solution to this problem was divided into two steps. First, we proposed a locking mechanism, similar to the one used in the dining philosophers' problem [30], that would ensure that two neighboring nodes on a DHT ring would never be joining and/or leaving concurrently. Second, we introduced the notion of a *join point* and a *leave point*, which denoted the atomic join, respective atomic leave, of a node. Provided the locking scheme, we showed algorithms that would guarantee that all lookups reported results that were consistent with the join and leave point of the system. The first such solution was based on lock queues, which had some efficiency problems. Therefore, we provided a second solution which was probabilistic.

We showed how atomic ring maintenance could be augmented to handle arbitrary additional routing pointers. Accounting algorithms were presented that ensure that routing failures never occur as nodes join and leave the system.

The atomic ring maintenance was also considered in the context of node failures. We showed that it is impossible to provide lookup consistency in an asynchronous network that can partition. Hence, we showed that Brewer's conjecture [19] applies to lookup consistency. Our lookup consistency guarantees can therefore be violated during failures. In spite of this, we showed how the algorithms could be made fault-tolerant, by showing how they could be extended and coupled with periodic stabilization. Hence, in absence of failures, the algorithms provide lookup consistency. If failures occur, inconsistent lookup results may be returned. It is left to periodic stabilization to correct the pointers, after which lookup consistency can be guaranteed again.

The presented work advances the state of the art on lookup consistency. Li, Misra, and Plaxton [23–25] independently discovered a similar approach to ours. An advantage of their work is that they use assertional reasoning to prove safety properties of their atomic ring maintenance algorithms. Their focus has, however, mostly been on the theoretical aspects of this problem. Hence, they assume a fault-free environment. They do not use their algorithms to provide lookup consistency. Furthermore, they cannot guarantee liveness, as their algorithms are not starvation-free. Lynch, Malkhi, and Ratajczak [26] proposed for the first time to provide atomic access to data in a DHT. They provide an algorithm in the appendix of the paper for achieving this, but give no proof of its correctness. As Li *et al.* point out, Lynch *et al.*'s algorithm does not work for both joins and leaves, and a message may be sent to a process that has already left the network [23].

References

1. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking **11**(1) (2003) 17–32
2. Kaashoek, M.F., Karger, D.R.: Koorde: A Simple Degree-optimal Distributed Hash Table. In: Proc. of the 2nd IPTPS. Volume 2735 of LNCS., Berkeley, CA, USA, Springer (2003) 98–107
3. Harvey, N., Jones, M.B., Saroiu, S., Theimer, M., Wolman, A.: Skipnet: A scalable overlay network with practical locality properties. In: Proc. of the 4th USENIX Symp. on Internet Technologies and Systems, Seattle, WA, USA, USENIX (March 2003)
4. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proc. of the 2nd ACM/IFIP Int. Conf. on Middleware. Volume 2218 of LNCS., Heidelberg, Germany, Springer (November 2001) 329–350
5. Zhao, B.Y., Huang, L., Rhea, S.C., Stribling, J., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A Global-scale Overlay for Rapid Service Deployment. IEEE Journal on Selected Areas in Communications (JSAC) **22**(1) (January 2004) 41–53
6. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proc. of the 21st Annual ACM Symp. on PODC, New York, NY, USA, ACM Press (2002)
7. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: Supporting Scalable Multi-Attribute Range Queries. In: Proc. of the ACM SIGCOMM 2004, Portland, OR, USA, ACM Press (March 2004) 353–366
8. Leong, B., Liskov, B., Demaine, E.: EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In: 12th Int. Conf. on Networks, Singapore, IEEE (November 2004)
9. H. Johansen, A. Allavena, R.v.R.: Fireflies: Scalable Support for Intrusion-Tolerant Overlay Networks. In Zwaenepoel, W., ed.: Proc. of Eurosys 2006, ACM European Chapter (April 2006)
10. Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: OpenDHT: a public DHT service and its uses. In: Proc. of the ACM SIGCOMM 2005, New York, NY, USA, ACM Press (2005) 73–84
11. Li, J., Stribling, J., Morris, R., Kaashoek, M.F.: Bandwidth-efficient management of DHT routing tables. In: Proc. of the 2nd USENIX Symposium on NSDI, Boston, MA, USA, USENIX (May 2005)

12. Liben-Nowell, D., Balakrishnan, H., Karger, D.R.: Analysis of the Evolution of Peer-to-Peer Systems. In: Proc. of the 21st Annual ACM Symp. on PODC, New York, NY, USA, ACM Press (2002) 233–242
13. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: Proc. of the 9th Annual ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'97), New York, NY, USA, ACM Press (1997) 311–320
14. Aspnes, J., Shah, G.: Skip graphs. In: Fourteenth Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'03). (January 2003) 384–393
15. Naor, M., Wieder, U.: Novel architectures for P2P applications: the continuous-discrete approach. In: Proc. of the 15th Annual ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03), ACM Press (2003) 50–59
16. Ghodsi, A.: Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden (December 2006)
17. Ghodsi, A., Haridi, S.: Atomic Ring Maintenance in Distributed Hash Tables. Technical Report T2007:05, Swedish Institute of Computer Science (SICS) (May 2007) also available at http://www.sics.se/∼ali/publ/atomic-tr.pdf.
18. Brewer, E.: Towards Robust Distributed Systems, invited talk at the 19th Annual ACM Symp. on PODC (2000)
19. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM Special Interest Group on Algorithms and Computation Theory News **33**(2) (2002) 51–59
20. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
21. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2) (1985) 374–382
22. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. Technical Report TR-819, Massachusetts Institute of Technology (MIT) (January 2002)
23. Li, X., Misra, J., Plaxton, C.G.: Concurrent maintenance of rings. Distributed Computing **19**(2) (2006) 126–148
24. Li, X., Misra, J., Plaxton, C.G.: Brief Announcement: Concurrent Maintenance of Rings. In: Proc. of the 23rd Annual ACM Symp. on PODC, NY, USA, ACM Press (2004) 376
25. Li, X., Misra, J., Plaxton, C.G.: Active and Concurrent Topology Maintenance. In: Proc. of the 18th Int. Conf. on Distributed Computing, London, UK, Springer (2004) 320–334
26. Lynch, N.A., Malkhi, D., Ratajczak, D.: Atomic Data Access in Distributed Hash Tables. In: Proc. of the First IPTPS. LNCS, London, UK, Springer (2002) 295–305
27. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proc. of the 14th Annual ACM Symp. on PODC, New York, NY, USA, ACM Press (1995) 214–222
28. Harris, T.L.: A Pragmatic Implementation of Non-blocking Linked-Lists. In: Proc. of the 15th Int. Conf. on Distributed Computing, London, UK, Springer (2001) 300–314
29. Lehmann, D., Rabin, M.: On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. In: Symp. on Principles of Programming Languages (POPL'81). (1981) 133–138
30. Dijkstra, E.W.: Hierarchical Ordering of Sequential Processes. Acta Informatica **1** (1971) 115–138

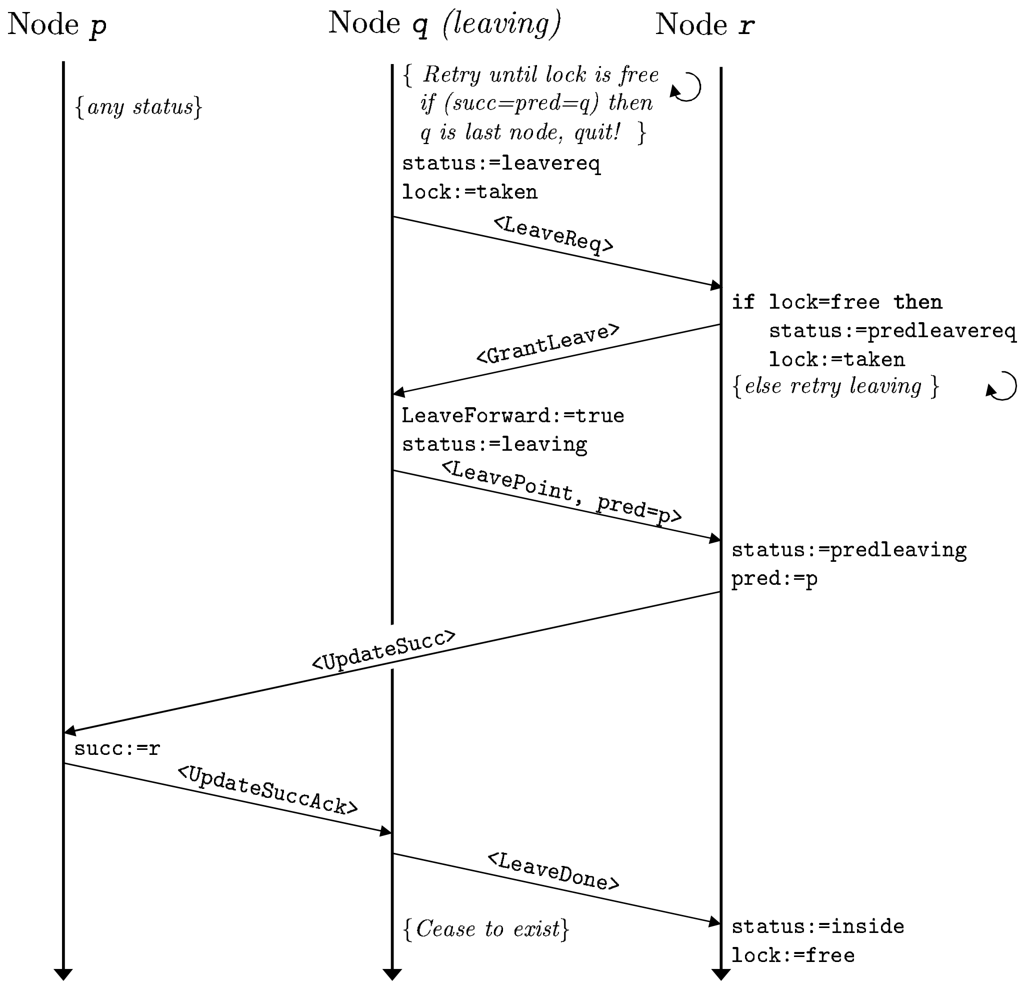

Fig. 6. Time-space diagram of the successful leave of a node.