Reasoning about Identifier Spaces: How to Make Chord Correct

Pamela Zave
AT&T Laboratories—Research
Bedminster, New Jersey 07921, USA
Email: pamela@research.att.com

***Abstract*—The Chord distributed hash table (DHT) is well-known and often used to implement peer-to-peer systems. Chord peers find other peers, and access their data, through a ring-shaped pointer structure in a large identifier space. Despite claims of proven correctness, i.e., eventual reachability, previous work has shown that the Chord ring-maintenance protocol is not correct under its original operating assumptions. Previous work has not, however, discovered whether Chord could be made correct under the same assumptions. The contribution of this paper is to provide the first specification of correct operations and initialization for Chord, an inductive invariant that is necessary and sufficient to support a proof of correctness, and two independent proofs of correctness. One proof is informal and intuitive, and applies to networks of any size. The other proof is based on a formal model in Alloy, and uses fully automated analysis to prove the assertions for networks of bounded size. The two proofs complement each other in several important ways.**

I. Introduction

Peer-to-peer systems are distributed systems featuring decentralized control, self-organization of similar nodes, fault-tolerance, and scalability. The best known peer-to-peer system is Chord, which was first presented in a 2001 SIGCOMM paper [1]. This paper was the fourth-most-cited paper in computer science for several years (according to Citeseer), and won the 2011 SIGCOMM Test-of-Time Award.

The Chord protocol maintains a network of nodes that can reach each other despite the fact that autonomous nodes can join the network, leave the network, or fail at any time. The nodes of a Chord network have identifiers in an m-bit identifier space, and reach each other through pointers in this identifier space. Because the network structure is based on adjacency in the identifier space, and $2^m - 1$ is adjacent to 0, the structure of a Chord network is a ring.

A Chord network is usually used to maintain a distributed hash table (DHT), which is a key-value store in which the keys are also identifiers in the same m-bit space. In turn, the hash table can be used to implement shared file storage, group directories, and many other purposes. Chord has been implemented many times, and used to build large-scale applications such as BitTorrent. And the continuing influence of Chord is easy to trace in more recent systems such as Dynamo [2].

The basic correctness property for Chord is eventual reachability: given ample time and no further joins, departures, or failures, the protocol can repair all defects in the ring structure. If the protocol is not correct in this sense, then some nodes of a Chord network will become permanently unreachable from other nodes. The introductions of the original Chord papers [1], [3] say, "Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance." An accompanying PODC paper [4] lists invariants of the ring-maintenance protocol.

The claims of simplicity and performance are certainly true. The Chord algorithms are far simpler and more completely specified than those of other DHTs, such as Pastry [5], Tapestry [6], CAN [7], and Kademlia [8]. Operations are fast because there are no atomic operations requiring locking of multiple nodes, and even queries are minimized.

Unfortunately, the claim of correctness is not true. The original specification with its original operating assumptions does not have eventual reachability, and *not one* of the seven properties claimed to be invariants in [4] is actually an invariant [9]. This was revealed by modeling the protocol in the Alloy language and checking its properties with the Alloy Analyzer [10].

The principal contribution of this paper is to provide the first specification of a version of Chord that is as efficient as the original, correct under reasonable operating assumptions, and actually proved correct. The new version corrects all the flaws that were revealed in [9], as well as some additional ones. The proof provides a great deal of insight into how rings in identifier spaces work, and is backed up by a formal, analyzable model.

Some motivations and possible benefits of this work are presented below. They are categorized according to the audience or constituency that would benefit.

For those who implement Chord or rely on a Chord implementation: It seems obvious that implementers should have a precise and correct specification to follow. They should understand the operating assumptions so as not to undermine them. They should also know the invariant for Chord, as dynamic checking of the invariant is a design principle for enhancing security in distributed systems [11].

Critics of this work have claimed that all the flaws in original Chord are either obvious and fixed by all implementers, or extremely unlikely to cause trouble during Chord execution. It is a fact that some implementations retain original flaws, citing [12] not because it is a bad implementation, but simply because the code is published and readable. Concerning whether the flaws cause real trouble or not, Chord implementations are certainly reported to have been unreliable. It is in the nature of distributed systems that failures are difficult to diagnose, and no one knows (or at least tells) what is really going on. Any means for increasing the reliability of distributed systems, especially without sacrificing efficiency, is an unmixed blessing.

For those interested in building more robust or more functional peer-to-peer systems based on Chord: Due to its simplicity and efficiency, it is an attractive idea to extend original Chord with stronger guarantees and additional properties. Work has already been done on protection against malicious peers [13], [14], [15], key consistency and data consistency [16], range queries [17], and atomic access to replicated data [18], [19].

For those who build on Chord, and reason about Chord behavior, their reasoning should have a sound foundation. Previous research on augmenting and strengthening Chord, as referenced above, relies on ambiguous descriptions of Chord and unsubstantiated claims about its behavior. These circumstances can lead to misunderstandings about how Chord works, as well as to unsound reasoning. For example, the performance analysis in [20] makes the assumption that every operation of a particular kind makes progress according to a particular measure, which is easily seen to be false [9].

For those interested in encouraging application of formal methods: This project has already had an impact, as developers at Amazon credit the discovery of Chord flaws [9] with convincing them that formal methods can be applied productively to real distributed systems [21].

The proof of correctness is also turning out to be an important case study. In this paper there are two independent proofs, one informal and one by model checking. The informal proof applies to networks of any size, and provides deep insight into how and why the protocol works. The Alloy model with its automated checking applies only to networks of bounded size, and offers limited insight, but it is an indispensable backup to the informal proof because it guards against human error. Also, it was an indispensable precursor to finding the general proof, because it indicated which theorems were likely to be true.

For those interested in formal proofs, the Alloy-only proof in [22] has been used as a test case for the Ivy proof system [23], and the new proof given here is being used as a test case for the Verdi proof system [24].

Finally, there are other possible uses for ring-shaped pointer structures in large identifier spaces (*e.g.,* [25], [7]). The reasoning about identifier spaces used in this paper may also be relevant to other work of this kind.

The paper begins with an overview of Chord using the revised, correct ring-maintenance operations (Section II), and a specification of these new operations (Section III). Although the specification is pseudocode for immediate accessibility, it is a paraphrase of the formal model in Alloy.

Correct operations are necessary but not sufficient. It is also necessary to initialize a network correctly. Original Chord is initialized with a network of one node, which is not correct, and Section IV shows why. This section also introduces the inductive invariant for the proof, because a Chord network can safely be initialized in any state that satisfies the invariant.

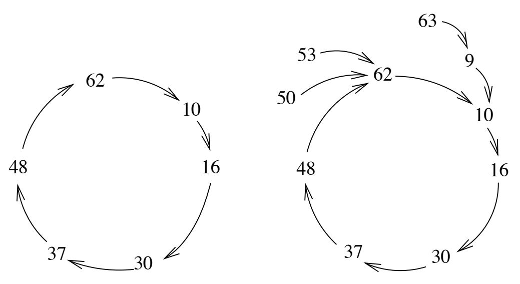

Fig. 1. Ideal (left) and valid (right) networks. Members are represented by their identifiers. Solid arrows are successor pointers.

Summarizing the previous two sections, Section V compares the revised Chord protocol with the original version, explaining how they differ. Together Sections IV and V present most of the problems with original Chord reported in [9] (as well as previously unreported ones). The problems are not presented first because they make more sense when explained along with their underlying nature and how to remove them.

The proof of correctness is largely based on reasoning about ring structures in identifier spaces. Section VI presents some useful theorems about these spaces and shows how they apply to Chord. The actual proof in Section VII follows a fairly conventional outline. Section VIII discusses the formal model and model-checked version of the proof, while Section IX covers related and future work.

II. Overview of correct Chord

Every member of a Chord network has an identifier (assumed unique) that is an m-bit hash of its IP address. Every member has a *successor list* of pointers to other members. The first element of this list is the *successor*, and is always shown as a solid arrow in the figures. Figure 1 shows two Chord networks with $m = 6$, one in the ideal state of a ring ordered by identifiers, and the other in the valid state of an ordered ring with appendages. In the networks of Figure 1, key-value pairs with keys from 31 through 37 are stored in member 37. While running the ring-maintenance protocol, a member also acquires and updates a *predecessor* pointer, which is always shown as a dotted arrow in the figures.

The ring-maintenance protocol is specified in terms of four operations, *join, stabilize, rectify,* and *fail.* Each operation is executed by a member and changes only the state of that member. In executing an operation, the member often queries another member or sequence of members, updating its own pointers as necessary after getting the result of each query. The specification of Chord assumes that inter-node communication is bidirectional and reliable, so we are not concerned with Chord behavior when inter-node communication fails.

A node becomes a member in a *join* operation. A member is also referred to as a *live* node. When a member joins, it contacts some existing member to look up a member that is near to it in identifier space, and gets a successor list from that

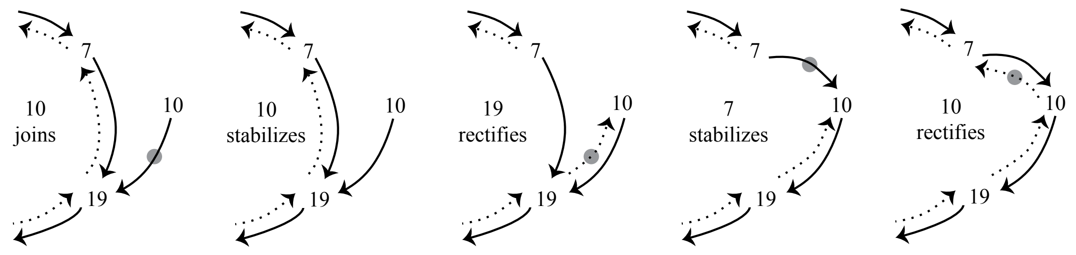

Fig. 2. A new node becomes part of the ring. A gray circle marks the pointer updated by an operation, if any. Dotted arrows are predecessors.

nearby member. The first stage of Figure 2 shows successor and predecessor pointers in a section of a network where 10 has just joined.

When a member *stabilizes*, it learns its successor's predecessor. It adopts the predecessor as its new successor, provided that the predecessor is closer in identifier order than its current successor. Because a member must query its successor to stabilize, this is also an opportunity for it to update its successor list with information from the successor. Members schedule their own stabilize operations, which should be periodic.

Between the first and second stages of Figure 2, 10 stabilizes. Because its successor's predecessor is 7, which is not a better successor for 10 than its current 19, this operation does not change the successor of 10.

After stabilizing (regardless of the result), a node notifies its successor of its identity. This causes the notified member to execute a *rectify* operation. The rectifying member adopts the notifying member as its new predecessor if the notifying member is closer in identifier order than its current predecessor, or if its current predecessor is dead. In the third stage of Figure 2, 10 has notified 19, and 19 has adopted 10 as its new predecessor.

In the fourth stage of Figure 2, 7 stabilizes, which causes it to adopt 10 as its new successor. In the last stage 7 notifies and 10 rectifies, so the predecessor of 10 becomes 7. Now the new member 10 is completely incorporated into the ring, and all the pointers shown are correct.

The protocol requires that a member or live node always responds to queries in a timely fashion. A node ceases to be a member in a *fail* operation, which can represent failure of the machine, or the node's silently leaving the network. A member that has failed is also referred to as a *dead* node. The protocol also requires that, after a member fails, it no longer responds to queries from other members. With this behavior, members can detect the failure of other members perfectly by observing whether they respond to a query before a timeout occurs. Failed nodes can rejoin later by executing a new *join* operation.

Failures can produce gaps in the ring, which are repaired during stabilization. As a member attempts to query its successor for stabilization, it may find that its successor is dead. In this case it attempts to query the next member in its successor list and make this its new successor, continuing through the list until it finds a live successor.

Chord relies on a critical operating assumption that a member always has a live successor in its list. In practice, an implementer must maintain the truth of this assumption by adjusting the length of successor lists (a parameter of the algorithm) and the rate of stabilization (which is not formally specified) to compensate for the failure rate. If successor lists are long enough to provide adequate redundancy, and stabilization occurs often enough to replace dead successors with live ones quickly, then a successor list should always have at least one live member remaining, even after an entry in its list fails.

As in the original Chord papers [1], [3], we wish to define a correctness property of eventual reachability: given ample time and no further disruptions, the ring-maintenance protocol can repair defects so that every member of a Chord network is reachable from every other member. Note that a network with appendages (nodes 50, 53, 63, 9 on the right side of Figure 1) cannot have full reachability, because an appendage cannot be reached by a member that is not in the same appendage. The correctness property here is slightly stronger, being based on the definition of the property *Ideal.*

Definition: A Chord network is in the *Ideal* state if:

- every pointer points to a live node;
- every successor pointer is its node's first successor in identifier order;
- every predecessor pointer is its node's first predecessor in identifier order; and
- the tail of the successor list of a node (past its head entry or successor) is the successor's successor list, with the last entry removed.

For example, on the right of Figure 1, the ideal successor of 48 is 50 because 50 is the closest successor to 48 in identifier order. The correctness property we need to prove is:

Starting in any execution state, if there are no subsequent join or fail operations, then eventually the network will become Ideal and remain Ideal.

Defining a member's *best successor* as its the first entry in its successor list pointing to a live node, a *ring member* is a member that can reach itself by following the chain of best successors. An *appendage member* is a member that is not a ring member. Of the seven invariants presented in [4] (and all violated by original Chord), the following four are necessary for correctness.

- There must be a ring, which means that there must be a non-empty set of ring members (*AtLeastOneRing*).
- There must be no more than one ring, which means that from each ring member, every other ring member is reachable by following the chain of best successors (*AtMostOneRing*).
- On the unique ring, the nodes must be in identifier order (*OrderedRing*).
- From each appendage member, the ring must be reachable by following the chain of best successors (*ConnectedAppendages*).

If any of these properties is violated, there is a defect in the structure that the ring-maintenance protocol cannot repair. If there is no ring, then ring-based reachability will not work. If there is more than one ring, then the network has separated into disjoint subnetworks. If appendage members cannot reach the ring, then the appendage is a disconnected fragment. A network that violates *OrderedRing* does not seem as catastrophic, but it impedes lookup, and the protocol cannot fix it [4]. It follows that any inductive invariant must imply these properties.

The Chord papers define the lookup protocol, which is used to find the member primarily responsible for a key, namely the ring member with the smallest identifier greater than or equal to the key. The lookup protocol is not discussed further here. Chord papers also define the maintenance and use of finger tables, which greatly improve lookup speed by providing pointers that cross the ring like chords of a circle. Because finger tables are an optimization and they are built from successor lists, their correctness relies on the correctness of successor lists. Finger tables are not discussed further here.

III. Specification of ring-maintenance operations

In this section, the operations, data structures, and related material are presented in pseudocode. Although pseudocode is not analyzable as the Alloy model is, it translates more directly to implementation code.

A. Identifiers and node state

There is a type *Identifier* which is a string of *m* bits. Implicitly, whenever a member transmits the identifier of a member, it also transmits its IP address so that the recipient can reach the identified member. The pair is self-authenticating, as the identifier must be the hash of the IP address according to a chosen function.

The Boolean function *between* is used to test the order of identifiers. Because identifier order wraps around at zero, it is meaningless to test the order of two identifiers—each precedes and succeeds the other. This is why *between* has three arguments:

```
Boolean function between (n1,nb,n2: Identifier)
{  if (n1 < n2) return ( n1 < nb && nb < n2 )
   else         return ( n1 < nb || nb < n2 )
}
```

For *nb* to be *between n1* and *n2*, it must be equal to neither. Further properties of identifier spaces are presented in Section VI.

Each node that is a member of a Chord network has the following state variables. They are all initialized by the *join* operation.

```
myIdent: Identifier;
prdc: Identifier;
succList: list Identifier;      // length is r
```

Note that *myIdent* is the hash of its IP address, and *prdc* is the node's predecessor. *succList* is the node's entire successor list; the head of this list is its *successor*. The parameter *r* is the fixed length of all successor lists.

B. Maintaining a shared-state abstraction

Reasoning about Chord requires reasoning about the global state, so the protocol must maintain the abstraction of a shared, global state. To do this, the algorithmic steps of the protocol must behave as if atomic and interleaved. In each algorithmic step, a node reads the state of at most one other node, and modifies only its own state.

In an implementation, a node reads the state of another node by querying it. If the node does not respond within a time parameter *t*, then it is presumed dead. If the node does respond, then the atomic step associated with the query is deemed to occur at the instant that the queried node responds with information about its own state.

To maintain the shared-state abstraction, the querying node must obey the following rules:

- The querying node does not know the instant that its query is answered; it only knows that the response was sent some time after it sent the query. So the querying node must treat its own state, between the time it sends the query and the time it finishes the step by updating its own state, as undefined. The querying node cannot respond to queries about its state from other nodes during this time.
- If the querying node is delaying response to a query because it is waiting for a response to its own query, it must return interim "response pending" messages so that it is not presumed dead.
- If a querying node is waiting for a response, and is queried by another node just to find out if it is alive or dead, it can respond immediately. This is possible because the response does not contain any information about its state.

This covers all possibilities except that of a deadlock due to circular waiting for query responses. Freedom from deadlock is covered in the proof of correctness in Section VII.

C. Join and fail operations

When a node is not a member of a Chord network, it has no Chord state variables, and does not respond to queries from Chord members. To join a Chord network, a node must first calculate its own Chord identifier *myIdent*. It must also know some member of the network—it does not even matter whether it is a ring member or appendage—and must ask the member to use the lookup protocol to find a member *newPrdc* such that *between (newPrdc, myIdent, head(newPrdc.succList))*.

Provided with this information, the node joins in a single atomic step, by executing the following pseudocode:

```
// Join step

// newPrdc has value from previous lookup
newPrdc: Identifier;

query newPrdc for newPrdc.succList;
if (query returns before timeout) {
   succList = newPrdc.succList;
   prdc = newPrdc;
}
else abort;
```

If the query fails then *newPrdc* has died, and the node has no choice but to try joining again later.

A fail operation is also a single atomic step. When a member node fails or leaves a Chord network, it deletes its Chord state variables and ceases to respond to queries. Fortunately, the proof of correctness shows that a node can re-join safely even if other nodes still have pointers to it from its former episode of membership.

D. Stabilize and rectify operations

A stabilize operation may require a sequence of steps. First, the stabilizing node executes a *StabilizeFromSuccessor* step:

```
// StabilizeFromSuccessor step

// newSucc not initialized
newSucc: Identifier;

query head(succList) for
      head(succList).prdc and
      head(succList).succList;
if (query returns before timeout) {
   // successor live, adopt its list as mine
   succList =
      append (
         head(succList),
         butLast(head(succList).succList)
      );
   newSucc = head(succList).prdc;
   if (between(myIdent,newSucc,head(succList)))
      // predecessor may be a better successor
      next step is StabilizeFromPredecessor;
      // else stabilization is complete
}
// successor is dead, remove from succList
else
   succList =
      append(tail(succList),last(succList)+1);
   next step is StabilizeFromSuccessor again;
```

First the node queries its successor for its successor's predecessor and successor list. If this query times out, then the node's successor is presumed dead. The node removes the dead successor from its successor list and does another *StabilizeFromSuccessor* step.[1] We know that eventually it will find a live successor in its list, because of the operating assumption (from Section II) that successor lists are long enough so that each list contains at least one live node.

Once the node has contacted a live successor, it adopts its successor list (all but the last entry) as its own second and later successors. It then tests the successor's predecessor to see if it might be a better first successor. If so, the node then executes a *StabilizeFromPredecessor* step. If not, the stabilization operation is complete.

The *StabilizeFromPredecessor* step is simple. The node queries its potential new successor for its successor list. If the new successor is live, the node adopts it and its successor list. If not, nothing changes. Either way, the stabilization operation is complete.

```
// StabilizeFromPredecessor step

// newSucc value came from previous step
newSucc: Identifier;

query newSucc for newSucc.succList;
if (query returns before timeout)
   // new successor is live, adopt it
   succList =
     append(newSucc,butLast(newSucc.succList));
   // else new successor is dead, no change
```

At the completion of each stabilization operation, regardless of the result, the stabilizing node sends a message to its successor notifying the successor of its presence as a predecessor. On receiving this notification, a node executes a single-step rectify operation, which may allow it to improve its predecessor pointer.

```
// Rectify step

// newPrdc value came from notification
newPrdc: Identifier;

if (between (prdc, newPrdc, myIdent))
   // newPrdc presumed live
   prdc = newPrdc;
else {
   query prdc to see if live;
   if (query returns before timeout)
      no change;
   // live newPrdc better than dead old one
   else prdc = newPrdc;
};
```

IV. Initialization and invariant

An *inductive invariant* is an invariant with the property that if the system satisfies the invariant before any event, then the system can be proved to satisfy the invariant after the event. By induction, if the system's initial state satisfies the invariant, then all system states satisfy the invariant.

Original Chord initializes a network with a single member that is its own successor, *i.e.,* the initial network is a ring of

[1]The empty place in the successor list is filled with an artificial entry at the end, created by adding one to the last real entry. The reason for this entry will be made clear by the proof.

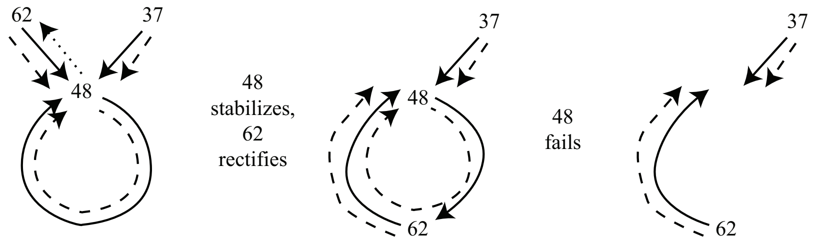

Fig. 3. Why the ring cannot be initialized at size 1. Dashed arrows are second-successor pointers. Predecessor pointers are not shown in the last two stages, as they are irrelevant. This problem was not reported in [9].

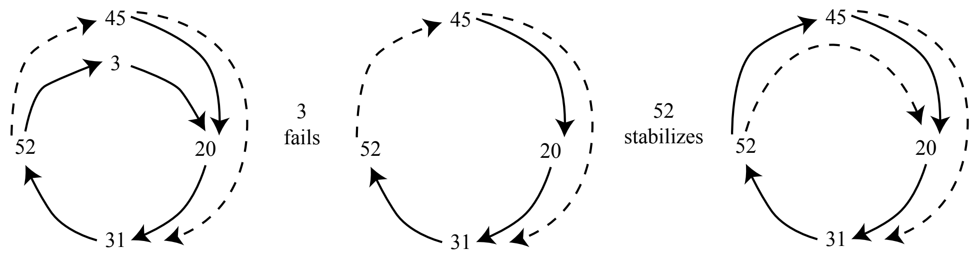

Fig. 4. A counterexample to a trial invariant. Only the relevant pointers are drawn.

size 1. This is not correct, as shown in Figure 3 with successor lists of length 2. Appendage nodes 62 and 37 start with both list entries equal to 48. Then 48 fails, leaving members 62 and 37 with insufficient information to find each other.

Clearly the spirit of the operating assumption in Section II is that the chosen length of successor lists should provide enough redundancy to ensure safe operation. But we can hardly expect the successor lists to work if the redundancy is thrown away by filling them with duplicate entries. This is the problem with Figure 3—that 62 and 37 have no real redundancy in their successor lists, so one failure disconnects them from the network.

For a member n of a network with successor list length r to enjoy full redundancy, n must have r entries in its successor list that are distinct from each other and from n. For this to be possible, the network must have at least $r + 1$ members, and the inductive invariant must imply that this is so.

The inductive invariant for Chord is the result of a very long and arduous search, some of which is described in [22]. As an indication of the difficulty, consider Figure 4, which is a counterexample to a trial invariant consisting of the conjunction of *AtLeastOneRing, AtMostOneRing, OrderedRing, ConnectedAppendages, NoDuplicates,* and *OrderedSuccessorLists*. Again $r = 2$. Let an *extended successor list* be the concatenation of a node with its successor list. *NoDuplicates* has the obvious meaning that the entries in any extended successor list are distinct. *OrderedSuccessorLists* says that for any ordered sublist *[x, y, z]* drawn from a node's extended successor list, whether the sublist is contiguous or not, *between [x, y, z]* holds.

In Figure 4, the first stage satisfies the trial invariant, having duplicate-free and ordered extended successor lists such as *[52, 3, 45]* and *[45, 20, 31]*. The appendage node 45 does not merge into the ring at the correct place, but that is part of normal Chord operation (see [9]). The second successor of ring node 52 points outside the ring, but that is also part of normal Chord operation. In the case shown in the figure, after two operations the ring has become disordered.

Note that the four important properties *AtLeastOneRing, AtMostOneRing, OrderedRing,* and *ConnectedAppendages* are all stated in terms of which nodes are ring members and which are appendage members. Unfortunately, ring *versus* appendage is not a stable characteristic of a member, but rather a fluid, context-dependent property that changes easily. In the figure, 45 changes from being an appendage member to a ring member just because 3 fails. In other examples, a ring member becomes an appendage member just because a node fails. So much of the difficulty of reasoning about Chord comes from the fact that the obvious properties have no intrinsic stability or persistence.

The final inductive invariant is much simpler than the earlier invariant used in [22]. It also has the major advantage of not requiring an extra operating assumption that is difficult to implement.

To explain the invariant, we must introduce the concept of a *principal node*. A principal node is a member that is not skipped by any member's extended successor list. For example, if 30 is a principal node, then *[30, 34, 39]* and *[27, 30, 34]* and *[21, 27, 29]* can all be extended successor lists, but *[27, 29, 34]* cannot be, because 30 is between 29 and 34, and would therefore be skipped.

The inductive invariant is the conjunction of only two properties, *OneLiveSuccessor* and *SufficientPrincipals*. *OneLiveSuccessor* simply says that every successor list has at least one live entry. *SufficientPrincipals* says that the number of principal members is greater than or equal to $r + 1$, where

r is the length of successor lists. A Chord network can be initialized in any state that satisfies the invariant.

The proofs in Section VI will show that this deceptively simple invariant implies all of *AtLeastOneRing, AtMostOneRing, OrderedRing, ConnectedAppendages, NoDuplicates,* and *OrderedSuccessorLists*. Needless to say, it also implies that the network has a minimum size. (Note that the first stage of Figure 4 has no principal members, so the figure is not a counterexample to the real invariant.)

A typical Chord network has r from 2 to 5, so the set of principals need only have 3 to 6 nodes. Nevertheless, the existence of these few nodes protects the correctness of a network with millions of members. They wield great and mysterious powers!

V. Comparison of the versions

In the original version of Chord, the *join, stabilize,* and *notified* operations are defined as pseudocode in [1] and [3], as is the initialization. These papers do not provide details about failure recovery, so the definition of the original version of Chord is completed by adding the pseudocode for *reconcile, update,* and *flush* operations from [4]. The "new" version of Chord is the one specified in this paper. The following table shows how operations of the two versions correspond. Although *rectify* in the new version is similar to *notified* in the original version, it seems more consistent to use an active verb form for its name.

original	**new**
join + reconcile	join
stabilize + reconcile + update	stabilize
notified + flush	rectify

In both original and new versions of Chord, members schedule their own maintenance operations except for *notified* and *rectify*, which occur when a member is notified by a predecessor. Although the operations are loosely expected to be periodic, scheduling is not formally constrained. As can be seen from the table, multiple smaller operations from the old version are assembled into larger new operations. This ensures that the successor lists of members are always fully populated with r entries, rather than having missing entries to be filled in by later operations. An incompletely populated successor list might lose (to failure) its last live successor. If the successor list belongs to an appendage member, this would mean that the appendage can no longer reach the ring, which is a violation of *ConnectedAppendages* [9].

Another systematic change from the old version to the new is that, before incorporating a pointer to a node into its state, a member checks that the node pointed to is live. This prevents cases where a member replaces a pointer to a live node with a pointer to a dead one. A bad replacement can also cause a successor list to have no live successor. If the successor list belongs to a ring member, this will cause a break in the ring, and a violation of *AtLeastOneRing*. Together these two systematic changes also prevent scenarios in which the ring becomes disordered or breaks into two rings of equal size (violating *OrderedRing* or *AtMostOneRing*, respectively [9]).

A third systematic change was necessary because the original version does not say anything precise about communication between nodes, and does not say anything at all about atomic steps and maintaining a shared-state abstraction. The new operations are specified in terms of atomic steps, and the rules for maintaining a shared-state abstraction are stated explicitly.

The other major difference is the initialization, as discussed in Section IV.

In addition to these systematic changes, a number of small changes were made. Some were due to problems detected by Alloy modeling and analysis of the original version. Others were required to ensure that, after each atomic step of a stabilize operation, the global state satisfies the invariant.

These differences do not change the efficiency of Chord operations in any significant way. Checking some pointers to make sure they point to live nodes (new version) requires more queries than in the original version. On the other hand, in the original version *stabilize, reconcile,* and *update* operations are all separate, and can all entail queries. In this respect the original version requires more queries than the new version.

There is an additional bonus in the new version for implementers. Consider what happens when a member node fails, recovers, and wishes to rejoin, all of which could occur within a short period of time. It was previously thought necessary for the node to wait until all previous references to its identifier had been cleared away (with high probability), because obsolete pointers could make the state incorrect. This wait was included in the first Chord implementation [26]. Yet the wait is unnecessary, as Chord is provably correct even with obsolete pointers.

In the spirit of [11], it is a good security practice to monitor that invariants are satisfied. Both the conjuncts of the inductive invariant are global, and thus unsuitable for local monitoring. The right properties to monitor are *NoDuplicates* and *OrderedSuccessorLists*, which can be checked on individual successor lists. These are properties that must be true for Chord networks of any size.

Although the new initialization with $r + 1$ principal nodes may not be inefficient, it is certainly more difficult to implement than initialization of original Chord. An alternative approach might be to start the network with a single node, and monitor the network as a whole until it has $r + 1$ principal nodes. For example, all nodes might send their successor lists (whenever there is a change) around the ring, to be collected and checked by the single initial node. Once the initial node sees a sufficient set of principal nodes, it could send a signal around the ring that monitoring is no longer necessary. This scheme is discussed further in Section VII-B.

VI. Reasoning about ring structures in identifier spaces

A. Theorems about identifier spaces

An identifier space is built from a finite, totally-ordered (in the usual binary sense) set. An identifier space also has a total ternary order *between,* defined in Alloy as:

```
pred between[n1,nb,n2: Node] {
   lt[n1,n2] =>   ( lt[n1,nb] && lt[nb,n2] )
              else ( lt[n1,nb] || lt[nb,n2] ) }
```

where `lt`, `&&`, `||` are the notations for less than (in the total binary order), logical and, and logical or, respectively. The definition has the form of an if-then-else expression. This definition has the same semantics as the pseudocode predicate *between* in Section III-A.

Informally, order in the identifier space "wraps around" from the last element of the binary order to the first. Because of this wraparound, two elements cannot be compared, which is why order in an identifier space must be ternary.

In this section definitions and theorems about identifier spaces are presented in the Alloy syntax. In the Alloy model the concepts of identifier and node (potential network member) are conflated, so that `Node` is declared as a totally ordered set upon which an identifier space is built. Details about the Alloy model and bounded verification can be found in Section VIII. These theorems have been proven for unbounded identifier spaces using merely substitution and simplification.

Here is a simple theorem in Alloy syntax:

```
assert AnyBetweenAny {
   all disj n1,n2: Node | between[n1,n2,n1] }
```

AnyBetweenAny says that for any distinct (disjoint) *n1* and *n2, n2* is between *n1* and *n1*.

For proofs, we also need a different predicate *includedIn*, which is like *between* except that the included identifier can be equal to either of the boundary identifiers:

```
pred includedIn[n1,nb,n2: Node] {
  lt[n1,n2] =>   ( lte[n1,nb] && lte[nb,n2] )
             else ( lte[n1,nb] || lte[nb,n2] ) }
```

In the *AnyIncludedInAny* theorem, the two arguments need not be disjoint:

```
assert AnyIncludedInAny {
   all n1,n2: Node | includedIn[n1,n2,n1] }
```

A very useful theorem allows us to reason about the fact or assumption that *between* does *not* hold.

```
assert IncludedReversesBetween {
   all disj n1,n2: Node, nb: Node |
       ! between[n1,nb,n2]
   <=> includedIn[n2,nb,n1]        }
```

Provided that the boundaries of an interval are distinct, if an identifier *nb* cannot be found in the portion of the identifier space from *n1* to *n2* (exclusive), then it must be found in the portion of the identifier space from *n2* to *n1* (inclusive).

The viewpoint of this paper is that identifier spaces have less structure than algebraic rings. Algebraic rings are generalizations of integer arithmetic, with operators such as sum and product that combine quantities. In Chord identifiers are not quantities, and it makes no sense to add or multiply them. This is in contrast to the formalization of Pastry [27], where distance in the identifier space is assumed to be meaningful and is used in the protocol.

B. Theorems about successor lists

This section introduces definitions and theorems about ring-shaped networks whose structure is based on successor lists in an identifier space. A number of terms concerning successor lists in network states were introduced briefly in Section IV. For clarity, they will be redefined here.

Definition: An *extended successor list* (ESL) is a successor list with the node that owns it prepended to the list. The length of an ESL is $r + 1$.

Definition: A *principal node* is a member that is not skipped by any ESL. That is, for all principal nodes *p*, there is no contiguous pair *[x, y]* in any ESL such that *between [x, p, y]*.

Definition: The property *OneLiveSuccessor* holds in a state if every member has at least one live entry in its successor list.

Definition: The property *SufficientPrincipals* holds in a state if the number of principal nodes is greater than or equal to $r + 1$.

Definition: The property *Invariant* is the conjunction of *OneLiveSuccessor* and *SufficientPrincipals*.

Definition: The property *NoDuplicates* holds in a state if every ESL has $r + 1$ distinct entries.

Definition: The property *OrderedSuccessorLists* holds in a state if for all sublists *[x, y, z]* of ESLs, whether contiguous sublists or not, *between [x, y, z]*.

The remainder of this section proves that *Invariant* implies the successor-list properties *NoDuplicates* and *OrderedSuccessorLists*.

Theorem: In any ring structure whose state is maintained in successor lists, *Invariant* implies *NoDuplicates*.

Proof:

Contrary to the theorem, assume that there is a network state for which *Invariant* is true and *NoDuplicates* is false. Then some node has an extended successor list with the form *[..., x, ..., x, ...]* for some identifier *x*.

From *AnyBetweenAny*, for all principal nodes *p* distinct from *x, between [x, p, x]*. Because of the definition of principal nodes, all of the principal nodes distinct from *x* must be listed in the ellipsis between the two occurrences of *x* in the successor list.

From *SufficientPrincipals*, the portion of the extended successor list *[x, ..., x]* must have length at least $r + 2$, because there are at least r principal nodes distinct from *x*. But the length of the entire extended successor list is $r + 1$, which yields a contradiction. □

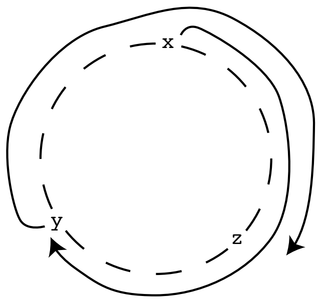

Fig. 5. The dashed line depicts the identifier space. The solid arrows show the path around the identifier space of a segment of an ESL *[x, ..., y, ..., z]*.

Theorem: In any ring structure whose state is maintained in successor lists, *Invariant* implies *OrderedSuccessorLists*.

Proof:

Contrary to the theorem, assume that there is a network state for which *Invariant* is true and *OrderedSuccessorLists* is false. Then some node has an ESL with the form *[..., x, ..., y, ..., z, ...]* where not *between [x, y, z]*. From the previous theorem, *x, y,* and *z* are all distinct.

From *IncludedReversesBetween*, *includedIn [z, y, x]*. If we visualize an identifier space as a ring ordered clockwise (as in Figure 5), the disordered ESL segment *[x, ..., y, ..., z]* wraps around the identifier ring touching the identifier space first at *x*, passing by *z*, touching at *y*, passing by *x* again, then finally touching at *z*.

It is easy to see that the only identifier in the entire identifier space that is not skipped by this ESL is y. Yet there must be more than one principal node, which is a contradiction. □

C. Theorem about networks built on successor lists

This section is concerned with proving that *Invariant* implies the four necessary properties introduced in Section II.

Definition: A network member's *best successor* is the first live node in its successor list.

Definition: A *ring member* is a network member that can be reached by following the chain of best successors beginning at itself.

Definition: An *appendage member* is a network member that is not a ring member.

Definition: The property *AtLeastOneRing* holds in a state if there is at least one ring member.

Definition: The property *AtMostOneRing* holds in a state if, from every ring member, it is possible to reach every other ring member by following the chain of best successors beginning at itself.

Definition: The property *OrderedRing* holds in a state if on the unique ring, the nodes are in identifier order. That is, if nodes *n1* and *n2* are ring members, and *n2* is the best successor of *n1*, then there is no other ring member *nb* such that *between [n1, nb, n2]*.

Definition: The property *ConnectedAppendages* holds in a state if, from every appendage member, a ring member can be reached by following the chain of best successors beginning at itself.

Theorem: In any ring structure whose state is maintained in successor lists, *Invariant* implies *AtLeastOneRing*, *AtMostOneRing*, *OrderedRing*, and *ConnectedAppendages*.

Proof:

The best-successor relation *bestSucc* is a binary relation on network members. We define from it a relation *splitBestSucc* that is the same except that every principal node *p* is replaced by two nodes p_s and p_d, where p_s (*s* for source) is in the domain of the relation but not the range, and p_d (*d* for destination) is in the range of the relation but not the domain. Figure 6 displays as graphs the *bestSucc* and *splitBestSucc* relations for the same network. It is possible to deduce many properties of the *splitBestSucc* graph, as follows:

(1) From *Invariant*, every member has a best successor. So the only nodes with no outgoing edges are p_d nodes representing principal members only as *being* best successors.

(2) p_s nodes have no incoming edges, as they represent principal nodes only as *having* best successors. There can be other nodes with no incoming edges, because there can be members that are no member's successor.

Note: The next few points concern maximal paths in the *splitBestSucc* graph. These are paths beginning at nodes with no incoming edges. By definition, they can only end at p_d nodes, and can have no internal nodes representing principal nodes.

(3) Just as a successor list does not skip principal nodes, a maximal path of best successors does not skip principal nodes. That is because an adjacent pair *[x, y]* in a chain of best successors is taken from the successor list of *x*, and if there are any entries in the successor list between *x* and *y*, they are dead.

(4) A maximal path is acyclic. Contrary to this statement, assume that the path contains a cycle *x leads to x*. By definition, the path has no nodes representing principal nodes. Yet the path traverses the entire identifier space, so it skips all principal nodes, which contradicts (3).

From (1-4), we know that the graph of *splitBestSucc* is an inverted forest (a "biological" forest, with roots on the bottom and leaves on the top). Each tree is rooted at a p_d node.

(5) A maximal path is ordered by identifiers. Contrary to this statement, let the path contain *[x, ..., y, ..., z]* where not *between [x, y, z]*. Because the path is acyclic, *x, y,* and *z* are all distinct. From *IncludedReversesBetween*, *includedIn [z, y, x]*. So the disordered path segment *[x, ..., y, ..., z]* wraps around the identifier ring exactly as the ESL does in Figure 5.

As in the proof of *OrderedSuccessorLists*, this segment skips every identifier in the entire identifier space except y. There must be more than one principal node, so this segment skips a principal node, contradicting (3).

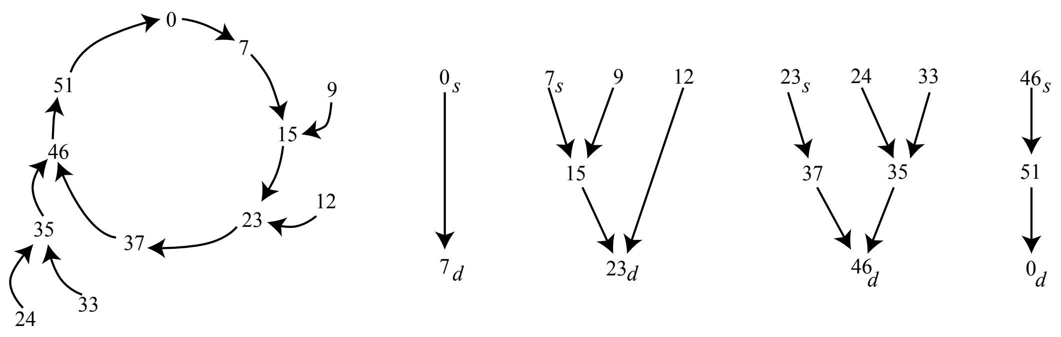

Fig. 6. For a network, the *bestSucc* relation is pictured on the left, and the *splitBestSucc* relation is pictured on the right. Although it cannot be seen from best successors only, the principal nodes are 0, 7, 23, and 46.

Note: The next two points concern the mapping from p_s nodes to p_d nodes derived from maximal paths in *splitBestSucc*. They show that it is a bijection, *i.e.,* one-to-one and onto.

(6) Every p_s node is a leaf of exactly one tree. It must be a leaf of some tree, because it begins a path of best successors that must end at a p_d node. It cannot be a leaf of more than one tree, because no node has more than one best successor.

(7) Every tree rooted at a p_d node has at least one leaf p_s, which is the principal node closest to p_d in reverse identifier order. It cannot have two such leaves $p1_s$ and $p2_s$, because the source principal closer to the destination principal would be skipped by the path of the farther source principal.

Summary:

In terms of *splitBestSucc*, a ring is formed by the concatenation of the unique maximal paths, one from each tree in the forest, starting at p_s nodes. This proves *AtLeastOneRing*. The relation cannot separate into two rings, because each of them would have a maximal path that skips a principal node in the other; this proves *AtMostOneRing*. From (5) we know that the ring is ordered by identifiers, so *OrderedRing* holds. All the nodes not on these unique maximal paths are appendage members, and each has a path to a principal node on the ring, so *ConnectedAppendages* holds. □

VII. Proof of Chord correctness

This section presents the proof of the theorem given in Section II:

Theorem: In any execution state, if there are no subsequent join or fail operations, then eventually the network will become Ideal and remain Ideal.

The most important part of this theorem is knowing that *Invariant* holds in all states, because this property and the properties it implies are the ones that all Chord users can count on at all times. We do not expect churn (joins and failures) to ever stop long enough for a network to become *Ideal.* Rather, this part of the theorem simply tells us that the repair algorithm always makes progress, and cannot get into unproductive loops.

A. Establishing the invariant

First it is necessary to prove that *Invariant,* which is true of any initial state (as specified in Section IV), is preserved by every atomic step of the protocol.

We begin with a failure step, because it requires a constraint based on the operating assumption in Section II: a member cannot fail if it would leave another member with no live successor. In other words, failure steps preserve the property of *OneLiveSuccessor* by operating assumption. No other kind of step can violate *OneLiveSuccessor*.

The other conjunct of *Invariant* is *SufficientPrincipals*, which says that the number of principal nodes must be at least $r + 1$. Rectify operations cannot violate this property, as they do not affect successor lists. In this section we will show that failure steps of non-principal nodes, join steps, *StabilizeFromSuccessor* steps, and *StabilizeFromPredecessor* steps do not cause principal nodes to become skipped in successor lists. This is the only way that they could violate *SufficientPrincipals.* The remaining case, that of failures of principal nodes, will be discussed in the next section.

Failure of a non-principal member *n* causes the disappearance of *n*'s successor list. But only being skipped in a successor list can make a node non-principal, so the disappearance of *n*'s successor list cannot make another node non-principal.

In a successful join, the new ESL created is *[myIdent, newPrdc.succList].* We know that there is no principal node between *myIdent* and *head (newPrdc.succList)*, because at the time of the query there is no principal node between *newPrdc* and *head (newPrdc.succList)*, and *myIdent* is between those two. We also know that *newPrdc.succList* cannot skip a principal node, by definition.

There are two cases in a *StabilizeFromSuccessor* step where a successor list is altered. In the first case the new ESL is a concatenation of pieces of the ESLs of the stabilizing node and its first successor, joined where they overlap at the first successor. Since neither of the original ESLs can skip a principal node, their overlap cannot, either.

In the second case a dead entry is removed from the stabilizing node's list, which cannot cause it to skip a principal. This leaves an empty space at the end which is temporarily padded with the last real entry plus one. This is the only value choice that preserves the invariant by guaranteeing that no principal node is skipped by accident. It does not matter whether the artificial entry points to a real node or not, as it will be gone by the time that the stabilization operation is complete.

There is only one case in a *StabilizeFromPredecessor* step where a successor list is altered. The new ESL created is *[myIdent, newSucc, butLast (newSucc.succList)]*. In the previous *StabilizeFromSuccessor* step, this node tested that *between [myIdent, newSucc, head (succList)]*. This node cannot make any other changes to its successor list between that step and this *StabilizeFromPredecessor* step, so it is still true. Therefore we know that there is no principal node between *myIdent* and *newSucc*, because there is no principal node between *myIdent* and *head (succList)*, and *newSucc* is between those two. We also know that *[newSucc, butLast (newSucc.succList)]* cannot skip a principal node, because it is part of the ESL of *newSucc*.

B. Failure of principal nodes

The fundamental reason why a Chord network must have $r + 1$ principal nodes is the need to prove *NoDuplicates*. Without *NoDuplicates* we cannot justify the operating assumption that a member always has a live successor, because the assumption is based on the full redundancy provided by ESLs with $r + 1$ distinct entries.

Apart from initialization, a member of a Chord network becomes a principal node when it has been a member long enough so that every node that should know about it does know about it. More specifically, it should appear in the successor lists of its r predecessors, which will happen after a sequence of r stabilizations in which each predecessor learns about the node from its successor.

It is extremely important that Section VII-A showed that none of the operations or steps of operations discussed there can demote a node from principal to non-principal. In other words, the *only* action that can reduce the size of the set of principal nodes is failure of a principal node itself.

As a Chord network grows and matures, a significant fraction of its nodes will be members long enough to become principals. This means that the number of principal nodes is proportional to the size of the network; once the network is large enough there is no possibility that *SufficientPrincipals* will be violated. Section V presented the idea of global monitoring of small Chord networks as a way to implement initialization with $r + 1$ principal nodes. It is a simple change to continue monitoring until the number of principal nodes has reached some multiple of r, after which the network is safe.

C. Queries have no circular waits

Section III-B explained how inter-node queries must be organized to maintain a shared-state abstraction. Sometimes a node must delay answering a query because it is waiting for the answer to its own query, which raises the specter of deadlock due to circular waiting.

Note that a rectify step only queries to see if a node is still alive, and does not read any of the node's state. Queries like these can always be answered immediately, so cannot cause waiting.

Note also that a join step requires a query, but no other node can be querying a node that has not joined yet. So the joining node, also, cannot be part of a circular wait.

This leaves queries due to the two stabilization steps, which are always directed to first successors or potential first successors. This means that, if there is a circular wait due to queries, it must encompass the entire ring. This possibility is sufficiently remote to ignore.[2]

D. Proving progress

This section shows that in a network satisfying *Invariant*, if there are no join or fail operations, then eventually the network will become *Ideal* and remain *Ideal* (as defined in Section II).

Progress proceeds in a sequence of phases. In the first phase, all leading dead entries are removed from successor lists, so that every member's successor is its best successor. Every time a member with a leading dead entry begins stabilization, it first executes a *StabilizeFromSuccessor* step, which will remove the leading dead entry. It will continue executing *StabilizeFromSuccessor* steps until all the leading dead entries are removed. Eventually all members will stabilize (this is an operating assumption), after which all leading dead entries will be removed from all successor lists.

Needless to say, these effective *StabilizeFromSuccessor* steps can be interleaved with other stabilize and rectify operations. However, rectify operations do not change successor lists. Even if a stabilization operation causes a node to change its successor, the steps are carefully designed so that the node will not change its successor to a dead entry. So, in the absence of failures, eventually all successors will be best successors, and will remain so.

In the second phase, which can proceed concurrently with or subsequent to the first phase, all successors and predecessors become correct. Let s be the current size of the network (number of members). This number is only changed by join and fail operations, and not by repair operations, so it remains the same throughout a repair-only phase as hypothesized by the theorem. The error of a successor or predecessor is defined as 0 if it points to the first successor (respectively, predecessor) in identifier order, 1 if it points to the second successor (predecessor) in identifier order, . . . $s - 1$ if it points to the least correct member, and s if it points to a dead node.

Whenever there is a merge in the *bestSucc* or *splitBestSucc* graph (see Figure 6), there are two nodes *n1* and *n2* with successors merging at *n3*, and for some choice of symbolic names, *between [n1, n2, n3]*. There are three cases: (1) *n3.prdc* (the current predecessor of *n3*) is better (has a smaller error) than *n2*, meaning that *between [n2, n3.prdc, n3]*; (2) *n3.prdc* is *n2*; (3) *n3.prdc* is worse (has a larger error) than *n2*, meaning that *between [n3.prdc, n2, n3]*. In each of these three cases

[2]The formal model uses shared-memory communication as an abstraction of queries. Waiting is not modeled, so this case is not a problem for formal analysis.

there is a sequence of enabled operations that will reduce the cumulative error in the network, as follows:

Case 1: Either *n1* or *n2* stabilizes, adopting *n3.prdc* as its successor and reducing the error of its successor. When the stabilizing node notifies *n3.prdc* and *n3.prdc* rectifies, it will change its predecessor pointer if and only if the change reduces error.

Case 2: n1 stabilizes, adopting *n2* as its successor and reducing the error of its successor. When *n1* notifies *n2* and *n2* rectifies, it will change its predecessor pointer if and only if the change reduces error.

Case 3: n2 stabilizes, which will not change its successor, but will have the effect of notifying *n3*. When *n3* rectifies, it will reduce the error of its predecessor by changing it to *n2*.

These cases show that, as long as there is a merge in the *bestSucc* graph, some operation or operations are enabled that will reduce the cumulative error of successor and predecessor pointers. Equally important, all operations are carefully designed so that a change never increases the error. At the same time, some of these operations will reduce the number of merges. For example, in Figure 6, let the merge of 24 and 33 at 35 be an example of Case 2. When 24 changes its successor to 33, which is not currently the successor of any node, the total number of merges is reduced.

As the network is finite, eventually there will be no merges in the *bestSucc* graph, which means that every node is a ring member. Because the ring is always ordered, the errors of all successors will be 0. The errors of all predecessors will also be 0, because whenever a successor pointer reaches its final value by stabilization, it notifies its successor. That node will update its predecessor pointer, and will never again change it, because no other candidate value can be superior. This is the completion of the second phase.

In the third and final phase, after all successors are correct, the tails of all successor lists become correct (if they are not already). Let the error of a successor list tail of length $r-1$ be defined as the length of its suffix that does not match its member's successor's successor list.

Let *n2* be the successor of *n1*, and let the error of *n2*'s successor list be e. When *n1* stabilizes, the error of its successor list becomes $\mathit{max}(e-1,0)$, as it is adopting *n2*'s successor list, after first prepending a correct entry (*n2*) and dropping an entry at the end. Thus improvements to successor lists propagate backward in identifier order. In the worst case, after a backward chain of $r-1$ stabilizations, the successor list of the last node of the chain will be globally correct. The correct list will continue propagating backward, leaving correctness in its wake. □

VIII. The Alloy model and bounded verification

As introduced in Section I, there is an Alloy model including specification of the operations, correctness properties, and assertions of the proof.[3] The reasons for using Alloy for this purpose can be found in [28].

The Alloy proof is direct rather than insightful. For example, there are assertions of all the theorems in Section VI. The Alloy Analyzer uses exhaustive enumeration to verify automatically that the theorems are true for all model instances up to some size bounds (see below). But unlike Section VI, this verification gives no insight into why the theorems are true.

The Alloy proof treats progress somewhat differently from Section VII-D. The model defines enabling predicates for all operation cases, where an enabling predicate is true if and only if a step or sequence of steps is enabled and will change the state of the network if it occurs. An assertion states that if a network is not *Ideal,* some operation is enabled that will change the state. Another assertion states that if a network is *Ideal,* no operation will change the state.

What is missing from the formal treatment of progress is the argument that every change makes progress. This is provided by the error metrics in the informal proof. In principle the error metrics could be defined and checked in Alloy, but experience suggests that this would be awkward and computationally complex.

The model is and has been an indispensable part of this research, for two reasons: First, it protects against human error in the long informal proof. Second, it was a necessary tool for getting to the proof. Without long periods of model exploration, it would not have been possible to discover that the obvious invariants are not sufficient, nor to discover an invariant that is. Without the formal model and automated verification, one wastes too much time trying to prove assertions that are not true.

The model is analyzed for all instances with $r \leq 3$ and $n \leq 9$, where n is the size of the identifier/node space. For the largest instances, the possible number of nodes is more than twice the sufficient number of principal nodes.

It is worth noting what experimenting with models and bounds is like. With $r = 2$, many new counterexamples (to the current draft model) were found by increasing the number of nodes from 5 to 6, and no new counterexamples were ever found by increasing the number of nodes from 6 to 7 or more. No new counterexamples were ever found by increasing r from 2 to 3. This makes $r = 3$ and $n = 9$ seem more than adequate.

IX. Related and future work

Other researchers have found problems with Chord implementations. Freedman *et al.* found that the assumption of bidirectional network communication can be violated in practice [29]. Model-checking of code has been used to find bugs in implementations of Chord [30], [31]. No previous work except [9], however, has discovered any problems with the specification of Chord.

Although other researchers have verified properties of DHTs [32], [27], they have not considered failures, which are by far the most difficult part of the problem. Other work on

[3]http://www.research.att.com/~pamela > How to Make Chord Correct.

verifiable ring maintenance operations [33] uses multi-node atomic operations, which are avoided by Chord. These studies use a variety of techniques. Adhering to the terminology used so far, in which "informal" means rigorous but not machine-checked, and "formal" means machine-checked: [33] employs informal specification and proof, while [32] has formalizations of a specification and an implementation in π-calculus, plus an informal proof of bisimulation. Only [27] is completely formal, with specification and proof in TLA.

The Alloy-only proof in an earlier version of this work [22] has been used as a test case for the Ivy proof system [23]. The Ivy version of Chord is a significant simplification, as it has larger atomic operations, and limits the length of successor lists to 2. Most importantly, it assumes the property *OneLiveSuccessor* without maintaining the *NoDuplicates* property that justifies the assumption (see Section VII-B). Nevertheless, the study yielded promising results with respect to its goal of automatically generating invariants.

Now that our understanding of the protocol has a firm foundation, it should be possible to exploit this knowledge to improve peer-to-peer networks further. If these efficient networks become more robust, they may find a whole new generation of applications.

For example, the assumptions of reliable network communication, bidirectional network communication, and perfect detection of failures through timeouts are all related and all suspect. With a bit more overhead, it might be possible to weaken these assumptions without compromising Chord's modest invariant.

It is certainly possible to enhance security just by checking local invariants, and it may be possible to improve enhancements such as protection against malicious peers [13], [14], [15], key consistency and data consistency [16], range queries [17], and atomic access to replicated data [18], [19]. The first step is to update this work with the new correct specification, then revisit the possible improvements in light of the new invariant.

X. Conclusion

The Chord ring-maintenance protocol is interesting in several ways. The design is extraordinary in its achievement of consistency and fault-tolerance with such simplicity, so little synchronization overhead, and such a weak assumption about the occurrence of failures. Unlike most protocols, which work according to self-evident principles, it is quite difficult to understand how and why Chord works.

As a case study in practical verification, the Chord project illustrates the value of a variety of techniques. Simple analysis for bug-finding [9], fully automated verification through bounded model-checking [22], and informal mathematical proof, all had important roles to play.

Acknowledgments

Helpful discussions with Bharath Balasubramanian, Ernie Cohen, Patrick Cousot, Gerard Holzmann, Daniel Jackson, Arvind Krishnamurthy, Leslie Lamport, Gary Leavens, Pete Manolios, Annabelle McIver, Ken McMillan, Jay Misra, Andreas Podelski, Emina Torlak, Natarajan Shankar, and Jim Woodcock have contributed greatly to this work. The anonymous reviewers significantly improved its presentation.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proceedings of ACM SIGCOMM*. ACM, August 2001.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2007.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, February 2003.

[4] D. Liben-Nowell, H. Balakrishnan, and D. Karger, "Analysis of the evolution of peer-to-peer systems," in *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*. ACM, 2002, pp. 233–242.

[5] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, January 2004.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*. ACM, August 2001.

[8] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[9] P. Zave, "Using lightweight modeling to understand Chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 50–57, April 2012.

[10] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006, 2012.

[11] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," in *Proceedings of IPTPS*. Springer LNCS 2429, 2002.

[12] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *Proceedings of the 20th ACM Symposium on Operating System Principles*. ACM, 2005, pp. 75–90.

[13] B. Awerbuch and C. Scheideler, "Towards a scalable and robust DHT," in *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2006, pp. 318–327.

[14] A. Fiat, J. Sala, and M. Young, "Making Chord robust to Byzantine attacks," in *Proceedings of the European Symposium on Algorithms*. Springer LNCS 3669, 2005, pp. 803–814.

[15] K. Needels and M. Kwon, "Secure routing in peer-to-peer distributed hash tables," in *Proceedings of the ACM Symposium on Applied Computing*. ACM, 2009, pp. 54–58.

[16] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in Scatter," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, October 2011.

[17] A. Gupta, D. Agrawal, and A. E. Abbadi, "Approximate range selection queries in peer-to-peer systems," in *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, 2003.

[18] N. Lynch, D. Malkhi, and D. Ratajczak, "Atomic data access in distributed hash tables," in *Proceedings of IPTPS*. Springer LNCS 2429, 2002, pp. 295–305.

[19] A. Muthitacharoen, S. Gilbert, and R. Morris, "Etna: A fault-tolerant algorithm for atomic mutable DHT data," MIT CSAIL Technical Report 2005-044, http://hdl.handle.net/1721.1/30555, 2005.

[20] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi, "A statistical theory of Chord under churn," in *Peer-to-Peer Systems IV*. Springer LNCS 3640, 2005.

[21] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, April 2015.

[22] P. Zave, "How to make Chord correct," arXiv:1502.06461v2 [cs:DC], 2015.

[23] O. Padon, K. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Interactive safety verification via counterexample generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016.

[24] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

[25] B. Awerbuch and C. Scheideler, "The hyperring: A low-congestion deterministic data structure for distributed environments," in *Proceedings of SODA*. ACM, 2004.

[26] H. Balakrishnan and I. Stoica, 2013, personal communication.

[27] N. Azmy, S. Merz, and C. Weidenbach, "A rigorous correctness proof for Pastry," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer LNCS 9675, 2016, pp. 86–101.

[28] P. Zave, "A practical comparison of Alloy and Spin," *Formal Aspects of Computing*, 2014, the final publication is available at Springer via http://dx.doi.org/10.1007/s00165-014-0302-2.

[29] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica, "Non-transitive connectivity and DHTs," in *Proceedings of the 2nd Conference on Real, Large, Distributed Systems*. USENIX, 2005, pp. 55–60.

[30] C. Killian, J. A. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *Proceedings of the 4th USENIX Symposium on Networked System Design and Implementation*, 2007, pp. 243–256.

[31] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak, "CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, April 2009.

[32] R. Bakhshi and D. Gurov, "Verification of peer-to-peer algorithms: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 181, pp. 35–47, 2007.

[33] X. Li, J. Misra, and C. G. Plaxton, "Active and concurrent topology maintenance," in *Distributed Computing*. Springer LNCS 3274, 2004, pp. 320–334.