How to Make Chord Correct
(Using a Stable Base)

Pamela Zave

AT&T Advanced Technology, Cloud Technologies and Services Research
Bedminster, New Jersey, USA
pamela@research.att.com

Abstract. The Chord distributed hash table (DHT) is well-known and frequently 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, formal modeling has shown that the Chord ring-maintenance protocol is not correct under its original operating assumptions [25]. It has not, however, discovered whether Chord could be made correct with reasonable operating assumptions. The principle contribution of this paper is to provide the first specification of a correct version of Chord, using the assumption that there is a small “stable base” of permanent members. The paper presents a simple, sufficient, and arguably necessary inductive invariant, and a partially automated proof of correctness.

1 Introduction

Peer-to-peer systems are distributed systems featuring decentralized control, self-organization of similar nodes, and scalability. A distributed hash table (DHT) is a peer-to-peer system that implements a persistent key-value store. It can be used for shared file storage, group directories, and many other purposes.

The distributed hash table Chord was first presented in a 2001 SIGCOMM paper [22]. 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 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 pointer 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.

The ring structure is disrupted when nodes join, leave, or fail. The original Chord papers [22, 23] specify a ring-maintenance protocol whose minimum correctness property is eventual reachability: given ample time and no further disruptions, the ring-maintenance protocol can repair all disruptions in the ring structure. If the protocol is not correct in this sense, then some nodes of a Chord network can become permanently unreachable from other nodes.
The introductions of the original Chord papers 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 [14] 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 [20], Tapestry [27], CAN [19], and Kademlia [16]. There is no attempt to specify or enforce fairness on distributed nodes. There are no atomic operations involving multiple nodes.

The ease of implementing Chord is probably the reason for its popularity as a component of peer-to-peer systems. Its fundamental simplicity is probably the reason for its popularity as a basis for building DHTs with stronger guarantees and additional capabilities, such as protection against malicious peers [2, 5, 18], key consistency and data consistency [7], range queries [8], and atomic access to replicated data [15, 17].

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 [14] is actually an invariant [25]. This was revealed by modeling the protocol in the Alloy language and checking its properties with the Alloy Analyzer [10], an exercise that illustrates rather clearly the importance of formal modeling of protocols.

The Alloy language combines first-order predicate logic, relational algebra, and transitive closure. The Alloy Analyzer verifies properties by means of exhaustive enumeration of instances over a bounded domain. This push-button analysis either yields a counterexample, or proves that the property holds in the bounded domain. The reasons for using Alloy in this work, as well as its limitations, are discussed in [26].

The principal contribution of this paper is to solve the problems that were revealed in [25] by providing the first specification of a correct version of Chord with reasonable operating assumptions, an inductive invariant for it, and a proof of correctness.

The proof has both manual and automated parts. The automated parts establish the invariant and guarantee that, if the state of the network is non-ideal, some repair operation is enabled that will change the network state. The manual part defines a measure, which is a non-negative integer, of the error in a non-ideal network. It also shows that every state change due to a repair operation reduces the error. Together these parts show that enabled repair operations will eventually reduce the error to zero, at which time the network state will be ideal.

As mentioned above, the Alloy Analyzer can only verify the lemmas automatically within a bounded domain. Although there is strong evidence that the size of the domain is sufficient for high confidence, a fully automated proof for systems of any size would be a welcome improvement.

The results in this paper are significant in three ways:

(1) Many people implement Chord, or use Chord as a component of their distributed systems. They should have a correct version of Chord to use. They
should also have a correct invariant for Chord, as this is a design principle for enhancing DHT security [21].

(2) Many people build on Chord, and reason about Chord behavior, for the purposes of their research. This reasoning should have a sound foundation. For example, the performance analysis in [12] makes incorrect assumptions about Chord behavior [25]. The research on augmenting and strengthening Chord, as referenced above, relies on informal descriptions of Chord and informal reasoning about its behavior. As automated proof checking increasingly becomes the norm in distributed systems, attempts to prove properties of systems based on original Chord will fail or yield unsound results. Even better than providing a foundation for formal reasoning about improvements to Chord, the new Chord might also provide a basis for synthesizing them.

(3) As will be explained in Section 4, efforts to find the best version of Chord and the best invariant for a proof have led to interesting insights into how Chord works. People who build on Chord should be aware of these properties so as to preserve them. Some principles may be applicable to all systems that use ring-shaped pointer structures in large identifier spaces (e.g., [1, 19]).

The paper begins with an overview of Chord using the revised, correct ring-maintenance operations (Section 2), and a new specification of these operations (Section 3). Although the specification is pseudocode for immediate accessibility, it is a paraphrase of the formal specification in Alloy. The complete Alloy model, including specification, invariant, and all steps of the proof, can be found at http://www2.research.att.com/~pamela/chordbase.als.

Correct operations are necessary but not sufficient. It is also necessary to have an inductive invariant to use in constructing a proof, and to initialize a network in a state that satisfies the invariant. Original Chord is initialized with a network of one node, which is not correct, and Section 4 shows why. Chord must be initialized with a ring containing a minimum of $r + 1$ nodes, where $r$ is the length of each node’s list of successors.

The $r + 1$ initial nodes can be assumed to form a “stable base,” in the sense that these nodes remain members of the network throughout its lifetime. Section 4 shows that the assumption of a stable base is surprisingly powerful. It is sufficient to enforce a simple invariant and allow a straightforward proof of correctness, which is presented in Section 5.

The stable base is an interesting assumption, because a stable base is very small, while a Chord network can have millions of members. Furthermore, there is no requirement on how members of the stable base are distributed around the ring. This means that there can be arbitrarily large sections of the ring that are not close to any member of the stable base. This suggests that all correctness problems arise from irregularities in small rings, that large rings evolved from small rings are well-behaved, and that the assumption of a stable base can be ignored when the ring is large.

Together Section 3 and Section 4 present most of the problems with original Chord reported in [25] (as well as adding some 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.

Although other researchers have found problems with Chord implementations [6, 11, 24], they have not discovered any problems with the specification of Chord. Other work on verifiable ring maintenance operations [13] uses multi-node atomic operations, which are avoided by Chord.

Section 6 discusses future work, including prospects for finding a proof of correctness that does not assume a stable base. Another interesting direction for future work concerns additional desirable extensions, properties, and optimizations not covered by eventual reachability in the basic protocol. Because the Chord protocol is simple and the code is compact, extending the protocol and establishing stronger properties might be a good test case for program synthesis.

2 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 three operations, each of which changes the state of at most one member. In executing an operation, the member queries another member or sequence of members, then updates its own pointers if necessary. 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 node is also referred to as live. When a member joins, it contacts an existing member and gets its
own current successor from that member. (It also contacts the current successor to get a full successor list.) 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 checks whether its current predecessor is still a member, and then adopts the notifying member as its new predecessor if the notifying member is closer in identifier order than its current predecessor (or if it has no live predecessor). 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.

One operating assumption of the protocol is that a member in good standing always responds to queries in a timely fashion. A node ceases to become a member in a fail event, 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 dead. Another operating assumption is that, after a member fails, it no longer responds to queries from other members. With these assumptions, members can detect the failure of other members perfectly by noticing whether they respond to a query before a timeout occurs. A third assumption about failure behavior is that successor lists are long enough, and failures are infrequent enough, to ensure that a member is never left with no live successor in its list.
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.

Defining a member’s best successor as its first successor pointing to a live node (member), 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 [14] (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 ($\text{AtLeastOneRing}$);
– 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 ($\text{AtMostOneRing}$);
– on the unique ring, the nodes must be in identifier order ($\text{OrderedRing}$);
– from each appendage member, the ring must be reachable by following the chain of best successors ($\text{ConnectedAppendages}$).

If any of these rules is violated, there is a disruption in the structure that the ring-maintenance protocol cannot repair. The inevitable result is that some members will be permanently unreachable from some other members.

A network is ideal when each pointer is globally correct. For example, on the right of Figure 1, the globally correct successor of 48 is 50 because it is the nearest member in identifier order. The minimum correctness criterion for the ring-maintenance protocol is simple: In any execution state, if there are no subsequent join or fail events, then eventually the network will become ideal and remain ideal. This is not a particularly stringent requirement, as it allows the protocol ample time and no further disruptions while it works to repair the ring.

The Chord papers define the lookup protocol, which is not discussed here. They also define the maintenance and use of finger tables, which improve lookup speed. Although finger tables are an optimization and the correctness of ring maintenance is not supposed to depend on them, they are discussed further in Section 3.

3 Specification of ring-maintenance operations

This section contains pseudocode, derived from the Alloy model, for the join, stabilize, and rectify operations. The code is much more complete and explicit than the original specification of Chord, particularly with respect to communication between nodes.

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 \texttt{between} is used to check the order of identifiers, and will be referred to in Section 4. Because identifier order wraps around at zero, it is meaningless to compare two identifiers—each precedes and succeeds the other. This is why \texttt{between} has three arguments:

\begin{verbatim}
Boolean function between (n1, n2, n3: Identifier) {
    if (n1 < n3) return ( n1 < n2 && n2 < n3 )
    else return ( n1 < n2 || n2 < n3 )
}
\end{verbatim}

It is important to note that, for all distinct \(x\) and \(y\), \texttt{between}(x,y,x) is always true, and \texttt{between}(x,x,y) and \texttt{between}(y,x,x) are always false.

The function

\begin{verbatim}
Identifier function lookupSucc (joining: Identifier) { }
\end{verbatim}

takes the identifier of a joining node and returns the identifier of its proper successor. In other words, for two members \(n\) and \(\text{lookupSucc}(\text{joining})\) that are adjacent in the ring, \texttt{between}(n,joining,\text{lookupSucc}(joining)).

Each node has the following variables:

\begin{verbatim}
myIdent: Identifier;
known: Identifier;
pred: Identifier;
succList: list Identifier; // length is r
\end{verbatim}

where \texttt{myIdent} is the hash of its IP address, \texttt{known} is a member of the Chord network known to the node when it joins, and \texttt{pred} is the node’s predecessor. \texttt{succList} is its entire successor list; the head of this list is its first successor or simply its successor. The parameter \(r\) is the fixed length of all successor lists.

To join, a node executes the following pseudocode.

\begin{verbatim}
// Join operation

newSucc: Identifier;

query known for lookupSucc(myIdent);
if (query returns before timeout) {
    newSucc = lookupSucc(myIdent);
    query newSucc for newSucc.succList;
    if (query returns before timeout) {
        succList =
            append(newSucc,
                butLast(newSucc.succList));
        pred = null;
    } else retry Join later;
} else retry Join later;
\end{verbatim}
First, the node asks the known node to look up the node’s identifier and get its proper successor, storing the value in `newSucc`. The node then queries `newSucc` for its successor list. Finally the node constructs its own successor list by concatenating `newSucc` and `newSucc`’s successor list, with the last element of the list trimmed off to produce a result of length `r`. If either of the queries fail the node has no choice but to retry again later.

To stabilize, a node executes the following pseudocode.

```
// Stabilize operation

newSucc: Identifier;

while (succList is not empty) {
    query head(succList) for head(succList).pred and
        head(succList).succList;
    if (query returns before timeout) {
        newSucc = head(succList).pred;
        succList =
            append(head(succList),
                butLast(head(succList).succList));
        if (between(myIdent,newSucc,head(succList))
            query newSucc for newSucc.succList;
        if (query returns before timeout)
            succList =
                append(newSucc,
                    butLast(newSucc.succList));
        }
    notify head(succList) of myIdent;
    break;
}
else succList = tail(succList);
```

In the outer loop of this code, 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 promotes its second successor to first and tries again. Once it has contacted a live successor, it executes inner code ending in a break out of the loop. The loop is guaranteed to terminate before `succList` is empty, based on the assumption that successor lists are long enough so that each list contains at least one live node.

Once it has contacted a live successor, the node first updates its successor list with its successor’s list. It then checks to see if the new pointer it has learned, its successor’s predecessor, is an improved successor. If so, and if `newSucc` is live, it adopts `newSucc` as its new successor. Thus the stabilize operation requires one or two queries for each traversal of the outer loop. Whether or not there is a live improved successor, the node notifies its successor of its own identity.
A node rectifies when it is notified, thereafter executing the following pseudocode:

```java
// Rectify operation
newPred: Identifier;

receive notification of newPred;
if (pred = null) pred = newPred;
else {
    query pred to see if live;
    if (query returns before timeout) {
        if (between(pred, newPred, myIdent))
            pred = newPred;
    }
    else pred = newPred;
}
```

When a node fails or leaves, it ceases to stabilize, notify, or respond to queries from other nodes. When a node rejoins, it re-initializes its Chord variables.

The join, stabilize, and rectify operations are quite different from the original pseudocode specification of Chord. There are two major types of change. First, multiple smaller operations are assembled into larger 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 not have a live successor. If the successor list belongs to an appendage member, this will cause a violation of `ConnectedAppendages` [25].

Second, before incorporating a pointer to a node into its state, a member usually checks that it 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 `AtLeast-OneRing`. Together these changes also prevent more exotic (albeit improbable) scenarios in which the ring becomes disordered or breaks into two rings of equal size [25].

Including both major and minor changes, the new specification has three sources:

- pseudocode and text fragments selected from the three papers [14, 22, 23];
- clarifications about the original implementation of Chord [3];
- other changes based on Alloy modeling and analysis, as described here and in the next two sections.

Chord also has finger tables, which optimize lookup by providing pointers that cross the ring like chords of a circle. Finger tables are built from successor lists and the correctness of ring maintenance does not depend on them, so
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 [25].

they are not included in this specification. It would be interesting, however, to synthesize their code or to provide and verify it.

4 The invariant

4.1 Derivation of the invariant

Correct operations are necessary but not sufficient. We also need an inductive invariant to use in constructing a proof, and the network must be initialized to a state that satisfies the invariant. Conjuncts of the invariant will be presented as informal paraphrases of the Alloy invariant.

Of the four conjuncts defined in Section 2, three of them constrain the network to have a single ordered ring (AtLeastOneRing, AtMostOneRing, and OrderedRing), while ConnectedAppendages constrains the appendage members to be able to reach the ring. All are necessary in the invariant, but even together are not sufficient.

As stated previously, a node’s successor list must have \( r \) entries because this is necessary to guarantee, under the protocol’s operating assumptions, that the node will always have a live successor. For the same reason, each successor list must have \( r \) distinct non-self entries.

Original Chord initializes a network with a single member that is its own successor, i.e., the initial network is a ring of size 1. This is not correct, as shown in Figure 3 with \( r = 2 \). 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. For members to be able to have \( r \) distinct non-self entries in ideal successor lists, a Chord network must be initialized and maintained with a minimum ring size of \( r + 1 \).

A minimum ring size can be maintained as an operating assumption, but it is not enforceable by normal Chord operations. These operations are local, and a member of a Chord network does not know how many members or ring members there are.
In search of other conjuncts of the invariant, consider node n’s *extended successor list* append(n, n.succList) of size r + 1. Clearly we need a conjunct *NoDuplicates* stating that a node’s extended successor list has no duplicated entries (in other words, r distinct non-self entries). Because a node’s successor list is ideally intended to replicate and/or become the ring structure, it seems necessary to have a conjunct *OrderedSuccessorList* saying that for all contiguous sublists (x, y, z) of a node’s extended successor list, between(x, y, z) holds. Ordered successor lists are a foundation of correct Chord.

Unfortunately the four original conjuncts, the two new conjuncts *NoDuplicates* and *OrderedSuccessorList*, and the operating assumption of a minimum ring size, are not sufficient to provide an inductive invariant. To give one of a multitude of counterexamples, consider Figure 4, which is another example with r = 2. The first stage satisfies the trial invariant, having duplicate-free and ordered extended successor lists such as (52, 3, 45) and (45, 3, 20). The appendage node 45 does not merge into the ring at the correct place, but that is part of normal Chord operation (see [25]). The second successor of ring node 52 points outside the ring, but that is also part of normal Chord operation. Once 3 fails, however, the ring of best successors becomes disordered.

At this time, the only inductive invariant that has been found relies on an operating assumption that a Chord network is initialized with a set of members containing a “stable base” of at least r + 1 members. The typical range for r is 3-5, so the typical stable base would require 4 to 6 members. These members are “stable” in the sense that they continue to be members throughout the life of the network, without ever leaving or failing and rejoining. Because a member’s identifier is derived from its IP address, this means that there is always a live IP host at that address, with a copy of the member state for that identifier.

To explain what property is preserved by a stable base, we say that a member n1 skips another member n2 if n2 is not in the successor list of n1, yet between(n1, n2, n1.lastSucc), where n1.lastSucc is the last member of the successor list. The member n1 typically skips n2 if n2 became a member after n1 last updated its successor list. So the invariant property supported by an assumption of a stable base is the property *BaseNotSkipped*, which says that no member of a Chord network ever skips a member of the stable base.
An invariant consisting of the four original conjuncts plus BaseNotSkipped is the basis of the proof of correctness described in Section 5. BaseNotSkipped eliminates the first stage of Figure 4, because 52 skips 20 and 31. Of the four ring members 3, 20, 31, and 52, at least three must be in the stable base, so 52 cannot skip two of them and still satisfy the invariant.

4.2 Discussion of the stable base

The operating assumption of a stable base allows a very simple invariant. The four original conjuncts plus BaseNotSkipped imply all of the minimum-ring-size operating assumption, NoDuplicates, and OrderedSuccessorList. OrderedSuccessorList is the best property to monitor for security, in the spirit of [21].

Although the assumption of a stable base seems strong, it can be considered necessary on the grounds that a stable base is the only operational mechanism that absolutely guarantees a minimum ring size. Note also that known is an important part of a Chord implementation, as explained in Section 3. If every potential member must know the identifier of a current member, surely that member or set of members can be considered part of a stable base.

The most interesting aspect of the stable-base assumption is that a stable base has few nodes and a Chord network can have millions. Furthermore, there is no requirement on how members of the stable base are distributed around the ring. This means that there can be arbitrarily large sections of the ring that are not close to any member of the stable base, and that the base members can have no local effect on.

This strongly suggests that all correctness problems arise from irregularities in small rings, and that large rings evolved from small rings are well-behaved. Thus the enforcement of a stable base can be neglected when the ring is large, except insofar as it is needed to provide potential members with known members.

5 Proof of correctness

This section presents the proof of eventual consistency:

Theorem: In any execution state of a Chord network, if there are no subsequent join or fail events, then eventually the network will become ideal and remain ideal.

5.1 Modeling concurrency

The formal model uses shared memory communication between nodes to simulate queries. An event is an atomic operation, executed by a single node and altering only its own state, that may use the result of a single query. Concurrency has an interleaving semantics. Thus the interleaved events model local computations performed by nodes between or after queries.

In the model, fail and rectify operations are independent events. Joins correspond to two events at the same node:
1. The node queries a known member for its current successor and executes an event of type `JoinLookup` if it gets one.
2. The node queries its current successor for a successor list and executes an event of type `Join` if it gets one.

A stabilize operation corresponds to one or two events at the same node:

1. The node queries its first successor for a predecessor and successor list, and executes an event of type `StabilizeFromOldSuccessor` if it gets them.
2. Otherwise the node queries subsequent successors in its list as above, until it succeeds in querying a live successor and executing an event of type `StabilizeFromOldSuccessor`.
3. If the acquired predecessor appears to be a better first successor, the node queries it for its successor list and executes an event of type `StabilizeFromNewSuccessor` if it gets the list.

As the event types are modeled in the form of logical constraints, it is necessary to use Alloy analysis to check that the constraints are consistent, i.e., that events of the types can exist or occur. This has been done, as is shown in full in [http://www2.research.att.com/~pamela/chordbase.als](http://www2.research.att.com/~pamela/chordbase.als). All other proof steps are also included.

A `JoinLookup` event establishes a precondition for its subsequent `Join` event. How can we be sure that the precondition still holds when the `Join` event occurs, knowing that other events can occur between this event pair? The precondition is

\[
\text{no } b \in \text{Network.base} \mid \text{Between[ } n, b, j\text{.newSucc }]
\]

where `no` is a quantifier meaning \(\neg \exists\), `Network.base` is the set of members of the stable base, `j` is the actual event of type `Join` (an Alloy object), `n` is the node executing `j`, and `j\text{.newSucc}` is the new successor of `n`. The precondition says that no member of the stable base lies between `n` and its new successor in identifier order. No term of this condition is mutable or time-dependent, so interleaved events cannot falsify it.

### 5.2 Establishing the invariant

The next step of the proof is to establish the inductive invariant, named `Valid` in the model, by proving that it is preserved by events of every type. This is done automatically by the Alloy Analyzer.

As mentioned in the introduction, the Alloy Analyzer verifies properties by means of exhaustive enumeration of instances over a bounded domain. The event lemmas and all other automatically-verified lemmas have been proven for \(r \leq 2\) and \(n \leq 8\), where \(n\) is the number of nodes (including ring members, appendage members, and non-members).

The bound \(r \leq 2\) is small and should be improved (see Section 6). For \(r = 2\) the bound \(n \leq 8\) seems comfortably large. There are three reasons for believing this:
As discussed in Section 4, there can be arbitrarily large sections of the ring unaffected by the stable base. This suggests that correctness problems arise from irregularities in small rings, and that large rings are well-behaved.

Ring structures have many symmetries. For example, it has been proved by Emerson and Namjoshi that for all properties of adjacent pairs of nodes, rings of size 4 are sufficient to exhibit all counterexamples [4]. This is not directly relevant because Chord’s properties are global rather than pairwise, but it does indicate the power of symmetry.

During the experience of model exploration with Alloy, many new behaviors were found by increasing the number of nodes from 5 to 6, and no new behaviors were ever found by increasing the number of nodes from 6 to 7. This makes 8 look safe as a limit.

5.3 Guaranteeing progress

For each type of event that repairs the ring structure, there is a predicate EffectiveEventTypeEnabled\([n, t]\). For a node \(n\) and state timestamped \(t\), EffectiveEventTypeEnabled\([n, t]\) is true if and only if at time \(t\), an event of that type can occur at \(n\), and if it does occur it will change the state of \(n\).

The definitions of these predicates must be checked for correctness. For each predicate, this is done by proving a lemma that if the predicate is true, the state is valid, and the event occurs, then the state of \(n\) is different after the event.

The purpose of these definitions is to use the Alloy Analyzer to prove two crucial lemmas. The predicate NetworkIsImprovable is true whenever some effective repair event is enabled:

\[
pred \text{NetworkIsImprovable} \[t: \text{Time}\] \{ \\
( \text{some } n: \text{Node} \mid \text{EffectiveSFOSenabled} \[n, t\] ) \\
|| ( \text{some } n: \text{Node} \mid \text{EffectiveSFNSenabled} \[n, t\] ) \\
|| ( \text{some } n, \text{newPrdc: Node} \mid \text{EffectiveRectifyEnabled} \[n, \text{newPrdc}, t\] ) \\
\}
\]

The predicate is used to assert that when the network is valid and not ideal, it can be improved by an enabled repair event:

assert ValidNetworkIsImprovable \(\{\)
\(\text{all } t: \text{Time} \mid \text{Valid}[t] \&\& \text{Ideal}[t] \Rightarrow \text{NetworkIsImprovable}[t] \}\)

We assume that if a repair event is enabled, it will eventually be scheduled and executed. Furthermore, once a network has become ideal, no executed repair event will change the state:

assert IdealNetworkIsNotImprovable \(\{\)
\(\text{all } t: \text{Time} \mid \text{Valid}[t] \&\& \text{Ideal}[t] \Rightarrow \neg \text{NetworkIsImprovable}[t] \}\)

Together these lemmas establish that whenever a network is in a non-ideal state, an effective repair event will eventually be executed and change the state.
The final step is to show that a sequence of effective repair events must eventually terminate by making the network ideal. This step is informal. We define a measure of the error in a Chord network, such that the measure of an ideal network is 0 and the measure of a non-ideal network is a positive integer. We will also show that every effective repair event reduces the measure. This completes the proof that a network with no new joins or fails will eventually become ideal.

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 any repair operations, so it remains the same throughout a repair phase. The error of a pointer is defined as follows:

- The error of a predecessor or first successor is 0 if it points to the globally correct member (in the sense of identifier order), 1 if it points to the next-most-correct member, \( \ldots s - 1 \) if it points to the least globally correct member, \( s \) if there is no pointer (possible only for a predecessor), and \( s + 1 \) if it points to a non-member.
- The error of a second or later successor is 0 if its node’s successor is live and the pointer matches the corresponding pointer of its node’s successor’s successor list. This holds regardless of whether the value of the pointer is globally correct or not. The error of the second or later successor is 1 otherwise.

The total error or just "error" of a network is defined as the sum over all members and all pointers of the pointer error.

- There are two cases of effective StabilizeFromOldSuccessor events (see Section 5.1). In one case, the member’s old successor was dead and is replaced by a live successor. In this case the error of the member’s first successor changes from \( s + 1 \) to something less than \( s \). The error of its second and later successors changes from 1 to 0.
- In the other case, the member’s old successor was live. In this case the error of the member’s first successor does not change, but the error of at least some of its second and later successors changes from 1 to 0. Note that if the stabilizing member is completely up-to-date and no part of its successor list changes, then this is not an effective event, and need not be considered.
- An effective StabilizeFromNewSuccessor always reduces the error of the first successor. After the event the error of all second and later successors is 0, so it may be decreased and is not increased.
- There are three cases of effective Rectify events (see Section 3). In one case there was no previous predecessor, and the error of the predecessor changes from \( s \) to something less than \( s \).
- In another case of an effective Rectify event, the previous predecessor was dead. In this case the error of the predecessor changes from \( s + 1 \) to something less than \( s \).
- In the third case of an effective Rectify event, the previous predecessor was live. In this case the error of the predecessor is always reduced.
6 Future work

6.1 Proof techniques

For the automated verification to be completely convincing, it should be carried out for some values of \( r \) greater than 2. This requires more work because currently \( r = 2 \) is hard-coded into the Alloy model. In the future automated verification will be extended to \( r = 3 \) or greater.

For the sake of absolute certainty, it would be desirable to have a fully automated proof for all values of \( r \) and \( n \) such that \( r \geq 2 \) and \( n \geq (r + 1) \).

6.2 Operating assumptions and invariant

In this section we consider the question of whether the improved specification of Chord can be proven correct with any operating assumptions other than a stable base. Extensive experience with model exploration suggests that if there is no stable base then other operating assumptions and code changes will be required. Most serious of the operating assumptions is that a minimum ring size is maintained.

Even with these operating assumptions and changes, the correctness question is still unresolved. We can look at either possibility, and find it undermined by evidence!

Considering the possibility that it is correct, then there must be an inductive invariant. Yet extensive exploration of the model has failed to find it. Many plausible properties have been investigated, and no combination of them has succeeded as an inductive invariant.

Considering the possibility that it is not correct, then there must be a counterexample. This would be a complete event trace, starting from a legal initial state and ending at a fatal error. Experience suggests that such a counterexample would require 6 nodes and at least 50 events.

The Alloy Analyzer cannot be used to find a counterexample because it cannot check traces longer than a few events. However, there are also models of several versions of Chord in Promela, the modeling language of the Spin model checker [9]. Spin has been used to explore the state space of a model with 6 nodes and traces of about 100 events [26]. Although the state space was explored well—if not exhaustively—the exploration did not find any counterexamples.

6.3 Other desirable properties

As mentioned in the introduction, there are other desirable properties for Chord networks in addition to eventual consistency with respect to ring maintenance. It would be interesting to prove that mechanisms for implementing them are correct, or to implement them correctly by code synthesis.
References