Open access • Journal Article • DOI:10.1145/2185376.2185383

Using lightweight modeling to understand chord — Source link

Pamela Zave

Institutions: AT&T Labs

Published on: 29 Mar 2012 - ACM Special Interest Group on Data Communication

Topics: Chord (peer-to-peer) and Chord (music)

Using Lightweight Modeling To Understand Chord

Pamela Zave
AT&T Laboratories—Research
Florham Park, New Jersey USA
pamela@research.att.com

ABSTRACT

Correctness of the Chord ring-maintenance protocol would mean that the protocol can eventually repair all disruptions in the ring structure, given ample time and no further disruptions while it is working. In other words, it is "eventual reachability." Under the same assumptions about failure behavior as made in the Chord papers, no published version of Chord is correct. This result is based on modeling the protocol in Alloy and analyzing it with the Alloy Analyzer. By combining the right selection of pseudocode and textual hints from several papers, and fixing flaws revealed by analysis, it is possible to get a version that may be correct. The paper also discusses the significance of these results, describes briefly how Alloy is used to model and reason about Chord, and compares Alloy analysis to model-checking.

Categories and Subject Descriptors:
C.2.2 [Network Protocols]: Protocol Verification

General Terms:
Verification, Design, Documentation, Performance

Keywords:
Alloy, Spin, distributed hash table (DHT)

1. INTRODUCTION

The distributed hash table Chord requires little introduction. It was first presented in a 2001 SIGCOMM paper [9] which won the 2011 SIGCOMM Test-of-Time Award. This paper was accompanied by a technical report [10], and followed shortly by a paper in Principles of Distributed Computing [7] and a paper in *Transactions on Networking* [11]. For mnemonic purposes, these papers will be referred to as [SIGCOMM], [TR], [PODC], and [TON], respectively.

The papers specify the ring-maintenance protocol by means of some concise pseudocode and some text. [PODC] lists invariants of the ring-maintenance protocol. The introductions of both [SIGCOMM] and [TON] say, "Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance." The papers refer to [TR] for the proof of correctness.

Despite these assurances, there are reasons to question the correctness of Chord. The invariants claimed in [PODC] have inconsistencies and ambiguities. The proofs in [TR] have undefined terms and unstated assumptions. The theorems mix correctness with performance analysis in an overly ambitious way. The breezy, informal reasoning seems prone to the all-too-human misconception that complex systems will work exactly the way we expect them to.

Distributed systems frequently do not work the way we expect them to—but the gap between human intuition and reality can now be bridged by powerful and convenient tools. This led me to wonder whether there might be a better way to reason about protocols such as Chord, without making the investment necessary for full-fledged theorem proving.

In particular, *lightweight modeling* is the process of building a small, abstract formal model of the key concepts of a software system, and then analyzing the model with a fully automated ("push-button") tool that works by exhaustive enumeration over a bounded domain of possibilities. For example, models written in Alloy can be analyzed by the Alloy Analyzer [4], and models written in Promela can be analyzed by the model-checker Spin [3]. This paper reports on the use of Alloy modeling and analysis to study the correctness of Chord.

In this paper, correctness of the ring-maintenance protocol would mean that the protocol can eventually repair all disruptions in the ring structure, given ample time and no further disruptions while it is working. It is a form of "eventual consistency" with respect to reachability. If the protocol is incorrect, then some members of a Chord network can become permanently unreachable from other members.

Eventual reachability is a different notion of correctness from the much-studied properties of key consistency (which means that all members agree about which members store values for which keys) and data consistency (which means that all members agree about the value of a particular key). It is also a more fundamental notion of correctness, as efforts to achieve key and data consistency assume eventual reachability.

The analysis reported here uses exactly the same assumptions about failure as the Chord papers do. Briefly, there is perfect failure detection, and successor lists are long enough to guarantee that a member is never left with no live successor.

Under the same assumptions made in the Chord papers, the [SIGCOMM] version of the protocol is not correct, and not one of the properties claimed invariant in [PODC] is actually invariantly true of it. The [PODC] version satisfies one invariant, but is still not correct. The results are presented by means of counterexamples to the invariants in Section 4. In preparation for the results, Section 2 gives a brief summary of the protocol and failure assumptions, and Section 3 introduces the invariants.

By selecting the right pseudocode from several papers, incorporating the right hints from the text of another paper, and fixing small flaws revealed by analysis, it is possible to come up with a "best" version that may be correct. Some implementors may have discovered this version independently, but it is impossible to tell without reading the code of various implementations, because the available implementations do not provide version information. In addition, there are other ways to implement Chord that might be provably correct or robust under weaker assumptions, with some cost in performance. Chord versions and implementations are discussed in Section 5.

Whether one cares about Chord or not, the significance of these results is that important research protocols such as Chord exist in a haze of uncertainty about their specifications, properties, versions, and performance characteristics. Lightweight modeling can reduce this uncertainty with relatively little effort and without specialized knowledge of verification.

Section 6 is a brief overview of modeling in Alloy and how it applies to Chord.[1] It describes how Alloy was used to reason about the protocol, including both successful and unsuccessful techniques. Because of Alloy's limitations, Section 6 also compares its relative strengths and weaknesses to those of model-checking. Section 7 concludes the paper with recommendations based on what we know so far.

2. THE PROTOCOL

This section summarizes the protocol as specified in the SIGCOMM paper. This version is discussed further in Section 5, along with other versions.

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* pointer, 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 acquires and updates a *predecessor* pointer, which is always shown as a dotted arrow in the figures. It also acquires a list of extra successors. The *second successor* is always shown as a dashed arrow.

The ring-maintenance protocol is specified in terms of events, each of which changes the state of at most one member. In executing an event, the member often queries other members, then updates local pointers if it decides there are better values. All analyses of Chord assume that inter-node communication is reliable.

A machine becomes a member in a *join* event. When a member joins, it contacts an existing member and gets its own correct current successor from that member. Joins cause the ring to have appendages such as those on the right side of Figure 1.

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. Members schedule their own stabilize events periodically.

[1]All the Alloy code for this work is available at `http://www2.research.att.com/~pamela/chord.html`.

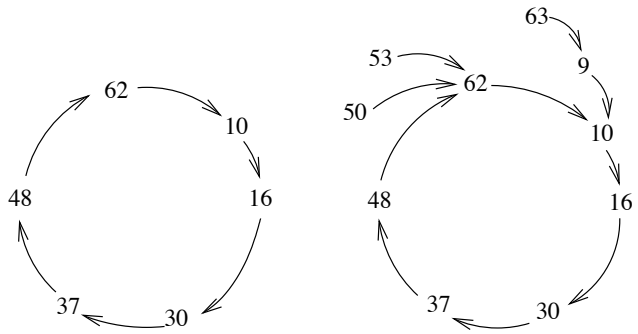

Figure 1: Ideal (left) and valid (right) networks. Members are represented by their identifiers.

After stabilizing, a node notifies its successor of its identity. The notified member adopts the notifying member as its new predecessor if the notifying member is closer in identifier order than its current predecessor. Thus a stabilize event always causes a *notified* event.

The purpose of *stabilize* and *notified* events is to repair disruptions in the ring structure caused by *join* events. The *pure-join* protocol is the version of the Chord protocol with only join, stabilize, and notified events. The pure-join protocol is correct, in the sense that it has the following property: *In any execution state, if there are no subsequent joins, then eventually the network will become ideal and remain ideal.* This theorem has been proven with the help of the Alloy Analyzer [14].

A machine ceases to become a member in a *fail* event, which also represents silent leaving of the network. When a member fails, it no longer responds to queries from other members. Until a member fails, it is responsive to queries. Together these assumptions allow perfect failure detection. Also, a member never fails if its failure would leave another member with no live (*live* means not failed or dead) successor in its successor list. In the model, this constraint compensates for the fact that successor lists are short. These assumptions and constraints on failure behavior are the basis for all analyses of Chord, including [PODC] and this paper's.

Failures can produce holes in the ring. These disruptions are repaired with the help of *reconcile, update,* and *flush* events, each of which is executed periodically by each member, according to its own schedule.

When a member *reconciles*, it adopts its successor's successor as its second successor (if successor lists are longer than two, it adopts its successor's entire successor list except for the last entry). When a member *updates*, it replaces a successor pointer to a dead member by the first successor pointer in its list that points to a live member. When a member *flushes*, it discards a dead predecessor.

The full protocol contains all seven events.[2] If it were correct, it would have the following property guaranteeing eventual reachability: *In any execution state, if there are no subsequent join or fail events, then eventually the network will become ideal and remain ideal.*

[2]The full protocol state also includes finger tables to optimize lookups, but these are not relevant to correctness.

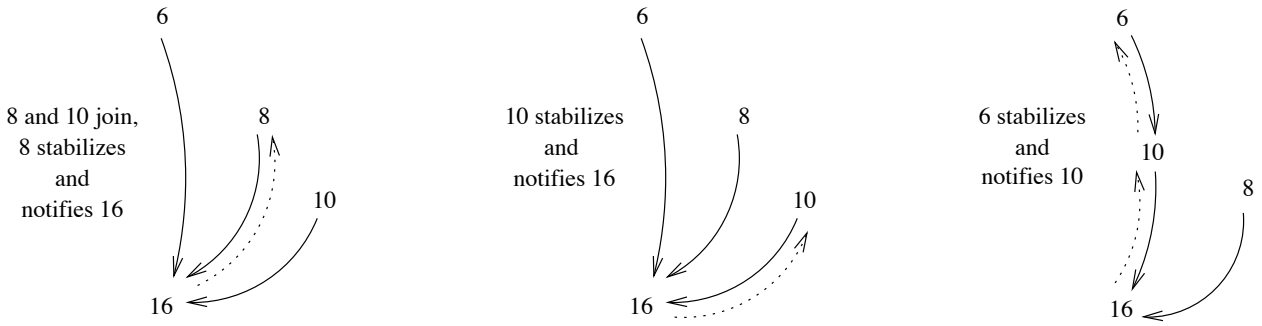

Figure 2: Three stages (left to right) creating a counterexample to *OrderedMerges*.

3. THE INVARIANTS

The invariants claimed in [PODC] are found in Definition 5.6 of that paper. We are not concerned with quantitative or probabilistic properties, nor with finger tables. This leaves us with Parts 1, 4(a), 4(b), 5(b), 5(c), and 5(d) of Definition 5.6, using some terminology defined in Section 5.2.

It takes some reformulation to make these properties complete, consistent, and unambiguous enough to be written in Alloy. For example, *connectivity* (Part 1) is defined as "There is a path using successor lists and finger tables connecting any two nodes." If "path using successor lists" has the obvious meaning of a path traversable by following successor pointers, then only networks without appendages can have connectivity. A node in an appendage cannot be reached by following successor pointers (or finger tables) from any node in the cycle or another appendage.

The spirit of connectivity is most conveniently expressed in Alloy as three properties: *AtLeastOneRing* (there is a cycle), *AtMostOneRing* (the network has not broken into multiple cycles), and *ConnectedAppendages* (each appendage is connected to the cycle).

Part 4(a) is formalized as *OrderedRing*, while Part 4(b) is formalized as *OrderedAppendages*. The spirit of these properties is that the order imposed by successors is consistent with identifier order, in the cycle and in the appendages respectively.

Part 5(b) is difficult to interpret, and is either frequently violated, a tautology, or means the same thing as 4(a) [15]. Part 5(c) is formalized as *OrderedMerges* (an appendage merges into the ring at the right place). Part 5(d) is formalized as *ValidSuccessorList* (see below).

4. RESULTS

The correctness results are given in this section by means of counterexamples to the invariants. Although other researchers have found problems with Chord implementations [1, 5, 13], they have not reported any of the problems reported here.

The seven Alloy invariants are divided into two categories, depending on whether they are needed for key and data consistency or for correctness.

4.1 Useful invariants

The three claimed invariants in this section are all desirable because they help support key and data consistency. Their consequences are presented here primarily in terms of lookups that should succeed but might fail. They are not necessary for correctness because states that do not satisfy them can eventually be repaired by the protocol.

The first claimed invariant is *OrderedMerges*. Informally, it says that an appendage merges into the ring at the right place in identifier order. *OrderedMerges* is easily violated, as shown in Figure 2.

Each figure in Section 4 consists of a sequence of alternating text blocks and diagrams. A text block describes a sequence of events. A diagram following a text block is a snapshot, showing the static state of the network after the events in the text block are finished. The arrows in the diagrams are first or second successors or predecessor pointers, depending on whether they are solid, dashed, or dotted respectively. In Figure 2, only part of the network is shown, and the section from 6 to 16 can be part of a ring of any size.

The scenario in Figure 2 begins when two appendages merge into the ring at 16. As the scenario unfolds, 10 gets incorporated into the ring before 8, and 8 ends up merging into the ring at the wrong place. From 8, lookup of keys 10 through 15 will fail, even though they would have succeeded if 8 were merging at the correct place.

Failure to understand how Chord networks behave also compromises the validity of performance analysis. For example, the performance analysis in [6] assumes that every stabilization event that changes a successor reduces the overall number of wrong successors by one. This assumption is incorrect: the stabilization by 6 in Figure 2 is one of several categories that change a successor but do not reduce the overall number of wrong successors.

The second claimed invariant is *OrderedAppendages*. Informally, it says that members are ordered correctly within an appendage. *OrderedAppendages* can be violated, as shown in Figure 3.

Violations of *OrderedAppendages* are similar in their consequences to violations of *OrderedMerges*. In the final state of Figure 3, from 5, a lookup of key 32 will fail, even though it would have succeeded if the appendage were ordered correctly. Note that when 44 is incorporated into the ring, the network will violate *OrderedMerges*.

The third claimed invariant is *ValidSuccessorList*. It can be paraphrased as saying that if v and w are members, and if w's successor list skips over v, then v is not in the successor list of any immediate antecedent of w. *ValidSuccessorList* can be violated, as shown in Figure 4. In the figure, the section from 9 to 25 can be part of a ring of any size.

In all the figures, "incorporated into the ring" is a shorthand for four events. The joining node must stabilize and

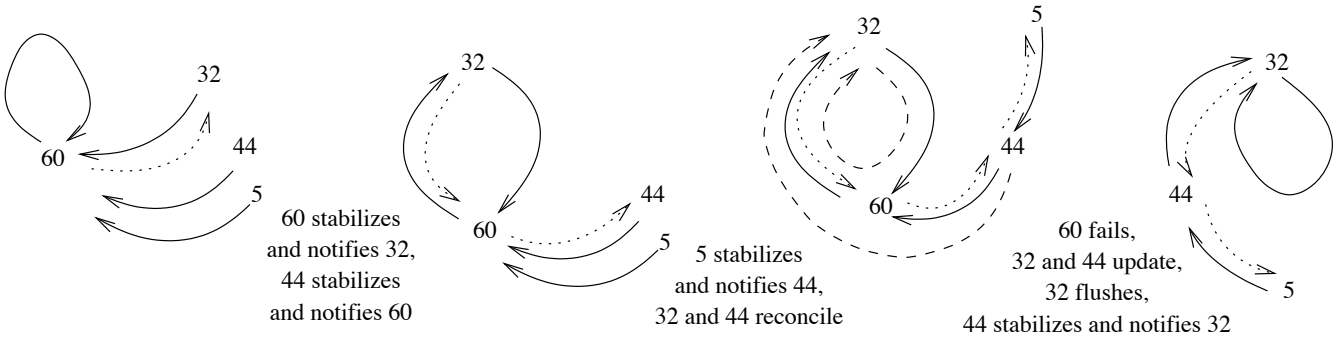

Figure 3: Four stages (left to right) creating a counterexample to *OrderedAppendages*.

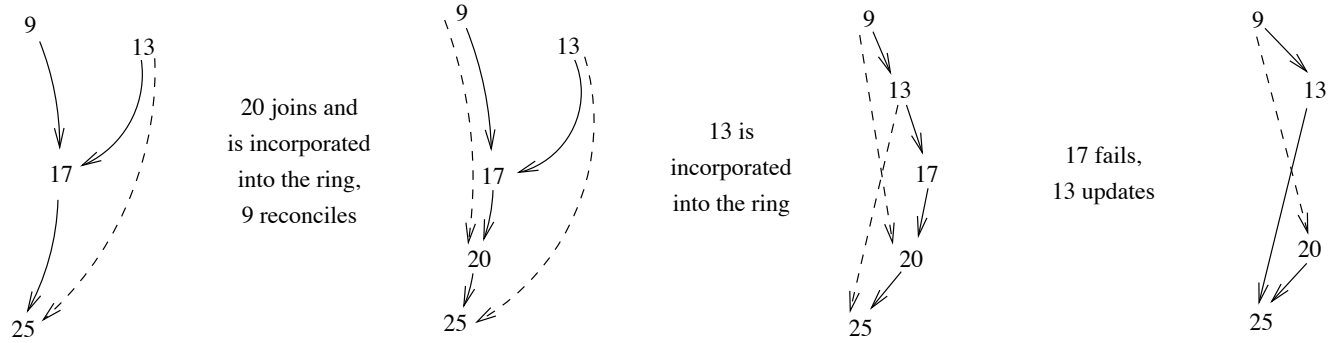

Figure 4: Three stages (left to right) creating a counterexample to *ValidSuccessorList*, and a fourth stage showing its effect after a failure.

notify, and then its predecessor in the ring must stabilize and notify.

It is not easy to tell from its definition why *ValidSuccessorList* is desirable. However, the fourth stage of Figure 4 shows that if it is violated, as it is in the third stage, then a member that was once in the ring (here 20) can revert to being an appendage. Until it is once more incorporated into the ring, its data will be inaccessible. In fact, the same problem can occur when there is a skipped member between a member's first and second successor, and the first successor fails, even if *ValidSuccessorList* is not violated [2].

4.2 Invariants required for correctness

The four claimed invariants in this section are necessary for correctness. If any one of them is violated, it creates a disruption in the ring structure that cannot be repaired by the ring-maintenance protocol, no matter how long it is allowed to run without further *join* and *fail* events to cope with.

The first such invariant is *ConnectedAppendages*. Informally, it says that an appendage to the ring stays connected to the ring. *ConnectedAppendages* can be violated, as shown in Figure 5. In the figure, the section from 5 to 22 can be part of a ring of any size.

In Figure 5, member 7's only connection to the network is through ex-member 9. When 5 updates to improve its successor, the ring as defined by successors no longer includes 9.

The next claimed invariant is named *AtLeastOneRing*. Informally, it says that there is always a ring of members, all reachable from each other. *AtLeastOneRing* can be violated,

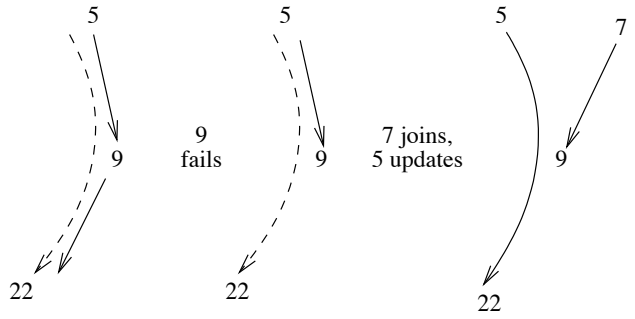

Figure 5: Three stages (left to right) creating a counterexample to *ConnectedAppendages*.

as shown in Figure 6. In the figure, the section from 6 to 12 can be part of a ring of any size.

Before the first stage of Figure 6, 10 has joined, stabilized, and notified 12. After 10 fails and 6 stabilizes, there is a gap in the ring, with no successor from 10 to 12. Although the gap is more apparent after 12 flushes its predecessor pointer to 10, this is not necessary for the counterexample.

The next claimed invariant is *OrderedRing*. Informally, it says that the ring is always ordered by identifiers. There is actually a class of counterexamples to this property, one for each odd ring size above 2. The counterexample for ring size 3 is shown in Figure 7.

In the multi-event transition from the first stage in Figure 7 to the second, three new members join the network and become incorporated into the ring. In the multi-event

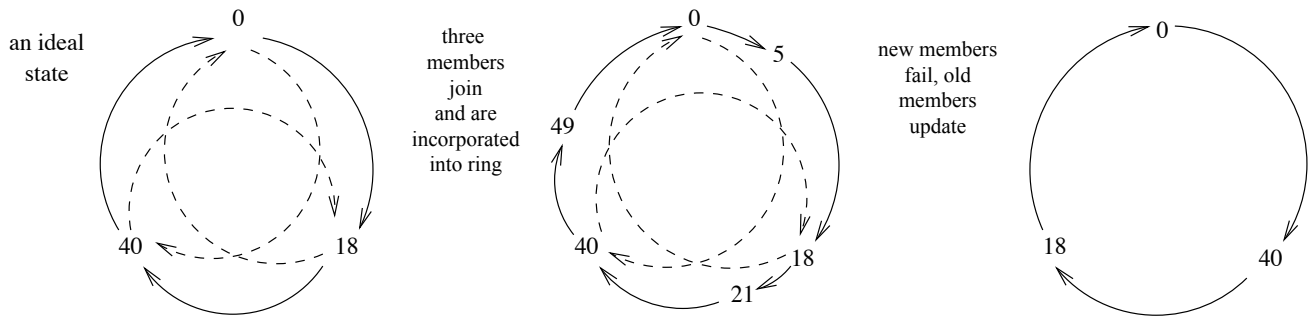

Figure 7: Three stages (left to right) creating a counterexample to *OrderedRing*.

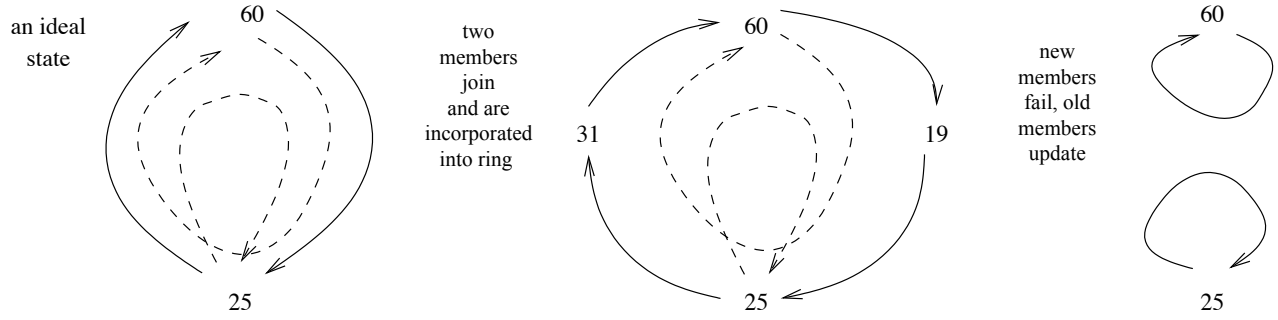

Figure 8: Three stages (left to right) creating a counterexample to *AtMostOneRing*.

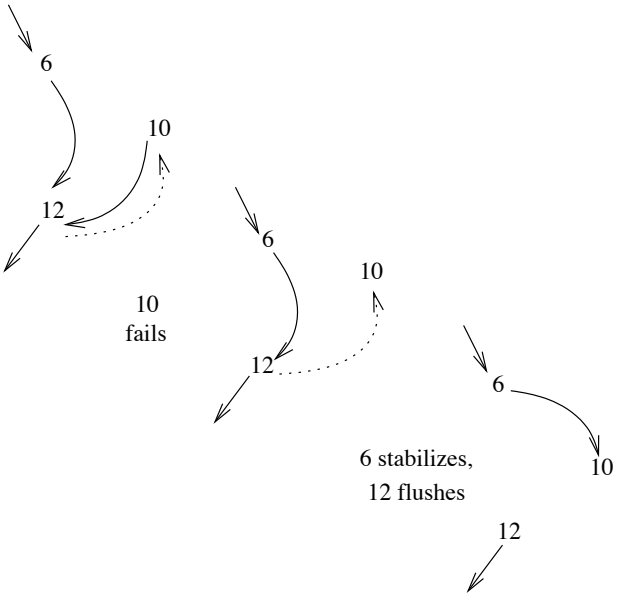

Figure 6: Three stages (left to right) creating a counterexample to *AtLeastOneRing*.

transition from the second stage to the third, all three new members fail, and all three old members update to promote their second successors. The result is a ring that is not ordered by identifiers.

The final claimed invariant is *AtMostOneRing*. Informally, it says that the network does not break apart into two or more separate rings. There is a class of counterexamples to this property, one for each even ring size at or above 2. The counterexample for ring size 2 is shown in Figure 8.

In the multi-event transition from the first stage in Figure 7 to the second, two new members join the network and become incorporated into the ring. In the multi-event transition from the second stage to the third, both new members fail, and both old members update to promote their second successors. The result is two disconnected rings.

5. VERSIONS OF CHORD

5.1 Versions of the specification

The version of the protocol that yields these counterexamples is based on the pseudocode for *join, stabilize,* and *notified* from [SIGCOMM], which is the same as in [TON]. These papers do not provide details about failure recovery. [PODC] is the only published paper with pseudocode for failure recovery, so the specification of *reconcile, update,* and *flush* events has been filled in from [PODC].

[PODC] improves the pseudocode of a join event to include the effects of a reconcile event. Because this forces a joining member to contact its new successor, and populates its successor list immediately, we can assume that it is implemented to prevent the problem in Figure 5. All the

other counterexamples can still occur, and the protocol is still incorrect.

I have constructed a "best" version of the protocol by selecting the pseudocode from [PODC] rather than [SIGCOMM]/[TON] and improving it with the help of a few selected hints in the text of [TON]. In this version stabilize and update events also include the effects of reconcile events. The specifications of both join and stabilize events have been altered to prevent the problems in Figures 5 and 6.

Although the "best" version still has the problems in Figures 2 and 3, the other counterexamples can no longer occur, and this version may be correct. Note that the counterexamples in Section 4 are mere artifacts of the analysis process, so getting rid of these problems is no guarantee that other problems are absent.

5.2 Chord implementations

Examining the Web sites of ten Chord implementations,[3] not one of them comments on the version or specification used. It is possible that implementors have discovered the "best" version independently, or made other improvements. If they have, there is no way for others to benefit from their knowledge. It is impossible to tell which version of the specification is used by any implementation without reading the code.

In fact, there is no guarantee that any known version of the specification is implemented. Work reported in [12] concerns techniques for automatically discovering system invariants from system implementations. The authors give as an example an invariant automatically discovered by analyzing an implementation of "Chord." Yet this invariant cannot possibly be true of a Chord implementation, because if it were true, the implementation could never create a ring with more than one member [15]. This means that the analyzed implementation is fundamentally different from Chord.

The uncertainty about versions is especially damaging to research progress because different versions can have very different performance characteristics and levels of correctness. To give an extreme example, [TR] and [PODC] include a different ring-maintenance protocol called "strong stabilization." It is far slower than the normal protocol, but it has the capacity to repair almost any structural defect. A Chord implementation that invoked strong stabilization at appropriate times or intervals would be correct by almost any definition, but it might not perform well enough to be useful.

To give another example, the OverLog (P2) implementation has specific time constraints on the frequencies of various maintenance events, although the Chord papers leave timing unconstrained [8]. Under certain assumptions about the frequencies of disrupting *join* and *fail* events, these time constraints might ensure correctness of the P2 implementation.

6. ON LIGHTWEIGHT MODELING IN ALLOY

6.1 Modeling

The Alloy language is a well-designed, object-oriented blend of first-order predicate logic and relational algebra, with the second-order operator of transitive closure built in. For anyone with a basic understanding of these concepts, Alloy is easy to learn.[4]

```
sig Node {
   succ: Node lone -> Time,
   succ2: Node lone -> Time,
   prdc: Node lone -> Time,
   bestSucc: Node lone -> Time }

open util/ordering[Node] as nodeOrder

fact JoinEvent {
   all j: Join, n: j.node, t: j.pre |
      NonMember[n,t]
   && (some m: Node |   Member[m,t]
                     && Between[m,n,m.succ.t]
                     && Member[m.succ.t,t]
                     && n.succ.(j.post) = m.succ.t
      )
   && no n.prdc.(j.post)                          }

pred OneOrderedRing [t: Time] {
   let ringMembers =
   {  n: Node | n in n.(^(bestSucc.t))  } |
      some ringMembers        -- at least one ring
   && (all disj n1, n2: ringMembers |
         n1 in n2.(^(bestSucc.t))   )   -- not two
   && (all disj n1, n2, n3: ringMembers |
         n2 = n1.bestSucc.t => ! Between[n1,n3,n2]
                        -- ring is globally ordered
      )                                           }
```

Figure 9: Fragments from an Alloy model of Chord.

Building an operational model of a protocol in Alloy is also easy, provided that the model is abstract enough to omit many implementation details. In Alloy, it is easiest to model distributed communication by means of shared state. The models written for this paper all allow members to read the states of other members, which avoids many implementation details while preserving the central concepts of Chord.

Figure 9 shows four fragments from the model of Chord in Alloy. This is not intended as an Alloy tutorial, but rather just a brief taste of the language. The first fragment says that there are individuals of type *Node*. At any time, each node has one or zero of each of these four pointers: *succ, succ2, prdc,* and *bestSucc*. Separate constraints say that a node is a *member* of the network (*i.e.,* live) if and only if it has a *succ* pointer, and that the value of *bestSucc* is the first successor pointing to a member, if any.

The second fragment invokes a library component. The result is that node individuals themselves are totally ordered, and there is no need for a separate node identifier.

The third fragment specifies a *join* event. Bound variable j is the event, n is the joining node, t is the time preceding the event, and m is the member from which n gets its successor. There are four preconditions that must be true at time t: n is not a member, m is a member, n is between m and its successor in identifier order, and m's successor is a

[3]From `http://pdos.csail.mit.edu/chord/faq.html`.

[4]See `http://alloy.mit.edu` for documentation and downloads.

member. There are two postconditions that must be true immediately after t: the successor of n is the successor of m, and n has no predecessor pointer. The full protocol, including all declarations and all seven events, requires about 100 lines.

It is equally important to formalize the properties that the protocol is believed to satisfy. Formalizing properties is easy when the modeling language is a good match for the desired properties. Alloy is particularly well-suited to expressing graph properties of a network state, which matches the [PODC] invariants.

For example, the fourth fragment is a combined definition of the properties *AtLeastOneRing, AtMostOneRing,* and *OrderedRing*. The set *ringMembers* contains all members that can reach themselves by following *bestSucc* pointers. The set must not be empty, or there would be no ring. Any two distinct members of it must be able to reach each other, or the network would have broken into multiple rings. If there are members *n1* and *n2* where *n2* is the best successor of *n1*, then no third member can come between them in identifier order.

With a succinct yet complete and analyzable model like this available as the specification for Chord, it would be easy to document versions and improvements so that all implementations could converge toward the best version. This is well worth doing, whether the protocol is proven correct or not.

6.2 Analysis

The other half of lightweight modeling is analysis to see if the model is internally consistent, and if it satisfies the asserted properties. Analyzers are designed to fit their modeling and assertion languages, so that any asserted property can be checked with a button push. If a model is consistent, then the analyzer will return an example of it. If a property is not satisfied by a model, then the analyzer will return a counterexample that violates it. Alloy has excellent visualization tools for customized display of examples and counterexamples as graphs.[5]

Push-button analysis is not as powerful as true verification because it works by exhaustive enumeration over a bounded—and usually small—domain of possibilities. For example, Chord properties can only be checked by the Alloy Analyzer for networks with 5-8 members. Experience shows, however, that analysis of small systems detects most problems. For systems with as much symmetry as a ring network, there is overwhelming theoretical and experimental evidence that analysis of small systems is sufficient [14].

The limitation of the Alloy Analyzer with respect to protocols is that time and temporal properties are not built in or optimized, so that the Analyzer can only search short traces. This is not an obstacle if we know an Alloy formula that characterizes all the states that the network can enter during execution—in other words, a *global invariant*. For example, there is an Alloy-supported proof that the pure-join version of Chord is correct [14]. For this proof, the Analyzer establishes that:

- The initial state satisfies the proposed global invariant.
- If the proposed global invariant holds before any event, it also holds after the event. In other words, it really is an invariant.
- If a state satisfies the global invariant but is not ideal, then some event is enabled that will improve the state.
- If a state is ideal, then no event (except for *join*) is enabled that will change the state.

Because a network is finite and can only require a finite number of improvements to become ideal, and because the proof only need apply when there are no new join events, these are sufficient to prove correctness.

Ideally, a protocol would be designed with a global invariant in mind, so that the invariant is part of the design from the beginning, and is constantly being checked as protocol events are designed. At the very least, this would provide designers with realistic feedback about how far the protocol diverges from their expectations.

6.3 Limitations and other approaches

The problem with using Alloy on Chord, even with the "best" version of Chord, is that we do not know the global invariant (if there is one). It can be extremely difficult to reverse-engineer a global invariant. For example, most of the work to get the results in Section 4 proceeded as follows, starting with a proposed global invariant:

- Analyze to see if the proposed global invariant is preserved by all events. Find an event that causes it to be violated.
- On the presumption that the pre-state could not have arisen in a real execution, make the invariant stronger to exclude the pre-state.
- Analyze to see if the proposed global invariant is preserved by all events. Find an event that causes it to be violated.
- On the presumption that the post-state (the one that violates the invariant) is benign, make the invariant weaker to allow the post-state.
- Repeat.

When this process does not converge, the only escape is to find a real trace from the initial state to a property violation, proving that the property is violated. Because this trace will be too long to be discovered by the Alloy Analyzer alone, it must be discovered manually, by working backward from a violation. After discovery, the long trace can be checked for validity by the Analyzer, because no search is required. For example, the counterexamples in Section 4 have been checked; a typical trace is 20 events, far beyond the searchable length of under 5.

As an alternative, a model-checker such as Spin with its modeling language Promela is also a good tool for lightweight modeling, with weaknesses and strengths that are complementary to those of Alloy. On the one hand, graph properties cannot be expressed in Promela, so it is necessary to express each property indirectly by means of a C program that checks for it. Spin traces do not look anything like snapshots of graphs, so it is necessary to write another C program to display counterexamples. These barriers make it difficult to experiment with graph properties, something that is easy with Alloy.

On the other hand, model-checkers are designed for the expression of temporal properties, and optimized for checking *all* traces. When a model-checker runs, it constructs an internal representation of all the states that can occur during execution. This substitutes for a global invariant. Note that the internal representation constructed by a model-checker cannot be used as a global invariant in the sense of Section 6.2, because it is very large and not human-readable.

[5]In the visualizer, click "Magic Layout" to get a time sequence of state snapshots.

At the moment, the best approach to producing an after-the-fact proof of correctness is not known. In future work, we will explore the use of model-checking for this purpose, and compare it with the use of Alloy.

7. CONCLUSIONS

For complex protocols such as Chord, there is every reason to use lightweight modeling as a design and documentation tool. It will be easy, and is essentially guaranteed to increase the quality of specifications and implementations. For protocols such as Chord where viewing a network as a graph is the most interesting abstraction, Alloy is the best choice for lightweight modeling.

In the design of distributed hash tables and other complex distributed systems, there are difficult trade-offs among correctness, performance, and operating assumptions. Correctness by construction often comes at the cost of performance. Assumptions about the operating conditions for a system, including load and failure modes, have a big effect on both performance and correctness requirements.

Although lightweight modeling does not answer any of the questions posed by these trade-offs, it can help create an environment in which both experiments and verification contribute to answering them. Simply put, the exact specification of a protocol really does affect its performance and its correctness. If an implementation implements an unknown specification, then nothing fundamental can be learned by experimenting with it.

Acknowledgments

Helpful discussions with Trevor Jim, Arvind Krishnamurthy, Yun Mao, and Jennifer Rexford have contributed greatly to this work.

8. REFERENCES

[1] 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*, pages 55–60. USENIX, 2005.

[2] 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.

[3] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

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

[5] 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*, pages 243–256, 2007.

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

[7] 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*, pages 233–242. ACM, 2002.

[8] 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*, pages 75–90. ACM, 2005.

[9] 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 SIGCOMM*. ACM, August 2001.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. MIT LCS Technical Report 819, `www.pdos.lcs.mit.edu/chord/papers/chord-tn`, 2001.

[11] 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*, 11(1), February 2003.

[12] M. Yabandeh, A. Anand, M. Canini, and D. Kostić. Almost-invariants: From bugs in distributed systems to invariants. Technical report, EPFL NSL-REPORT-2009-007, 2009.

[13] 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.

[14] P. Zave. Lightweight modeling of network protocols: The case of Chord. Technical report, AT&T Laboratories—Research, January 2010.

[15] P. Zave. Experiences with protocol description. Technical report, AT&T Laboratories—Research, June 2011. Presented at the 1st International Workshop on Rigorous Protocol Engineering, October 2011, Vancouver, British Columbia.