tential bugs.

1 Introduction

Peer-to-peer (P2P) systems have recently gained popularity in both academia and industry because they allow distributed computation and storage without the need for centralized control. This paper investigates Chord, a set of protocols for a P2P system. We model and analyze the Chord maintenance protocols described in [1], using the Alloy Analyzer [2]. We discuss some anomalies uncovered by both our precise modeling and automated analysis.

2 Overview of Chord

Chord is a peer-to-peer routing protocol that allows for efficient key lookups. The protocol describes an "ideal" overlay structure for the network that allows for efficient lookups; the procedures for nodes joining the network; and maintenance routines to maintain the structure of the overlay. In the presence of possibly concurrent node arrivals and departures, the idealization routine will return the network overlay to an ideal structure; with high probability in logarithmic time (in the number of nodes).

We give a brief overview of the structure and lookup of a Chord network, the ideal overlay structure of a Chord network, a model of joins, a model of failures, and a description of maintenance algorithms. The pseudo-code for the operations in the protocol appears in Appendix A.

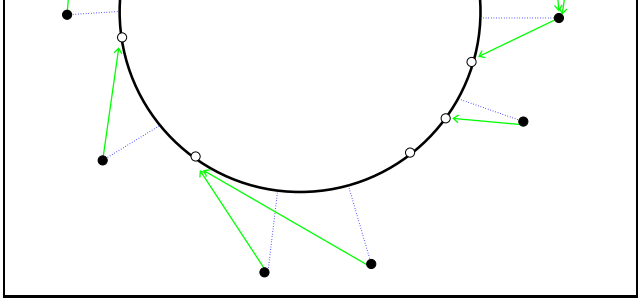

Figure 1: The Chord key-node mapping (from [1]).

2.1 Structure and Lookup

The core structure of a Chord network is a ring containing nodes and key-value pairs. Each node and key are assigned an identifier using a consistent hash function. Henceforth we will assume all keys lie in the identifier space. Each key k is assigned to the first node that is equal to or follows the key in the identifier space (we call this the *successor* of k). The identifier space can be thought of as a ring, and the successor of k is the first node found by traversing the ring in a clockwise direction. Each node maintains a successor list, so that even if a node's successor fails, it can still contact the rest of the network. Each node also maintains the id of the the node immediately preceding it on the ring, which is called the *predecessor*. The predecessor reference is used to properly place new nodes on the ring. A sample network can be seen in Figure 1.

Each node maintains a *finger table*, which is used to resolve requests for data. In short, a finger table maps offsets from the current node to the closest preceding node of that offset. When a request comes in for a particular key, the node looks up the closest preceding

Figure 2: A Loopy Chord Ring (from [1]).

node in its finger table, and forwards the request to that node, which then looks up the key in its table, etc. If each finger table contains mappings for all offsets 2^i, then the lookup time is logarithmic in the number of id's. Our model is not concerned with the efficiency of the protocol. Correctness of lookups only requires that the next node on the ring is in the finger table. Thus, our model is concerned with the correctness of the successors of nodes.

2.2 An Ideal Network

The motivation of a the maintenance protocol for Chord is to allow a Chord network to return to a "good" state even in the face of concurrent joins and failures, and possibly unforeseen errors/bugs that deviate from the protocol. First, we describe a one error that can occur in a Chord network, a *loopy cycle*.

2.2.1 Loopy Cycle

A Chord network is *weakly ideal* if, for all nodes u, $(u.successor).predecessor = u$. A network is *strongly ideal* if it is weakly ideal, and for each node u, there is no node v that is reachable from u so that $u < v < u.successor$. A *loopy* network is one that is weakly ideal but not strongly ideal. The maintenance protocol for Chord only aims for a weakly ideal network. So a loopy network could result. In a loopy network, the same query beginning at two different nodes can result in different behavior. A sample network in the loopy state can be seen in Figure 2.

$u.finger[i]$.

To avoid confusion between ideal, weakly ideal, and strongly ideal states, we shall henceforth refer to the ideal state as the "perfect" state.

2.3 A Pure Join Model

The pseudo-code for the join operation appears in Appendix A. In our Alloy model, we model a protocol that allows only joins. In this model, successor lists are not necessary; each node keeps track of just one successor node. After a node joins the network, it will have a pointer to its successor, but it will not be on the cycle yet, since other nodes on the network do not know of the new node's presence. We define a *cycle-with-appendages* state that restricts such a network to maintain the good properties of an ideal state.

A Chord network is in *cycle with appendages state* if: 1 and 2 are as in the ideal state.

3 [**cycle sufficiency**]

   (a) Of the nodes on the cycle, a subset of size at least $N/2$ is uniformly and independently distributed around the identifier circle.

   (b) For any cycle node u, we have $|A_u| = O(\log N)$.

4 [**non-loopiness**]

   (a) The cycle is non-loopy

   (b) For every node v in the appendage A_u, the path of successors from v to u is increasing.

5 [**successor validity**] For every node v:

   (a) if v is on the cycle, then v.successor is the first cycle node following v.

the idea behind this operation is ...

While the Chord protocol makes guarantees about a strongly ideal state being maintained with high probability, a loopy state could in fact occur if there is a violation of protocol, or a low probability event occurs. For this reason, a strong idealization protocol exists, which will return a Chord network in any state to a strongly ideal state. This maintenance protocol can be run infrequently, in order to ensure that if an error occurs in the network, it will eventually be fixed. This strong idealization protocol is in Appendix A.

3 An Alloy Model of Chord

3.1 Overview of the Model

A complete model is attached in Appendix B. In this section, we describe some of the elements of this model in more detail. This model is a modified version of the model of Chord developed by Hoeteck Wee [3].

The basic components of the model are Id's, Node's, and NodeData.

- **Id**

  The Id sig represents the identifiers of a Chord ring. The identifiers are arranged in a ring, via the next relation.

- **Node**

  The Node sig represents a Node in a Chord ring. The only static state that a Node has is it's Id. All the other state is represented by NodeData, because this state can change with time.

- **NodeData**

sition describes one step of the strong idealization protocol.

Lastly, our model contains descriptions of the different states a Chord ring can be in. These descriptions take the form of predicates on a State element. We have predicates for the condition of loopiness, weak and strong idealness

4 Results

4.1 Ambiguities and Bugs

While modeling these protocols, we discovered some ambiguities and one bug.

- In the pure join model, connectivity only holds for root cycle. In a first pass of our model, we modeled the mechanism for nodes joining a Chord ring. In doing this, we found an ambiguity in Definition 5.4.1 of [1]. The definition defines connectivity as "There is a path using successor lists and finger tables connecting any two nodes." This definition only makes sense for a purely ideal Chord network; there is not path from a node on the ring to a node on an appendage.

- Weakly and strongly ideal restricts the root cycle only. For instance, the weakly ideal criteria states that $u.successor.predecessor = u$. In fact, this only must hold for each node on the root cycle. We checked an assertion that a non-loopy state would not idealize to a loopy state. We found a counterexample in which only the root cycle was loopy in the initial state. An author of [1] confirmed this new definition of loopiness.

a contradiction to the claim that a non-loopy network stays non-loopy through idealization. We discuss this result in more detail in Section 4.3.1.

4.2 Some Examples

In appendix C we present a series of figures showing steps of the various protocols – join, weak idealization, and strong idealization. Each figure represents one state of the system. The shaded ovals represent active Nodes. The successor and predecessor relations are text attributes of the NodeData ovals. The center ring shows the ordering of the key and node id's in their ring.

4.3 Properties We Checked

4.3.1 Idealization Preserves Non-Loopiness

With the original definition of (x, x) (see Section 4.1), we found no counterexamples to the claim that weak idealization preserves non-loopiness. We checked this with 4 nodes and 6 state transitions.

Using the modified definition of (x, x), a counterexample arose. We show the pre- and post-states where a network transitions from a non-loopy state to a loopy state in Figures 3 and 4, respectively. If a node's predecessor is set to itself, any other node may idealize and reset the node's predecessor. In the case when that predecessor link is the only link that keeps the network from being weakly ideal, one transition can cause the network can to become loopy.

We have discussed this issue with one of the authors of [1], who has provided us with a new definition of loopiness. In this definition, a network is in the loopy state if : there is some node n, with associated appendage node u (the first node on the root cycle from

idealness and the root cycle, it is not clear if there is a guarantee that strong idealization results in a perfect state (where the network is a ring). We wrote an assertion stating that a network with a strongly ideal root cycle would end up perfect through strong idealization. However, the paper claims that this happens in $O(n^2)$ time. This resulted in scopes that were too large for the Alloy Analyzer, and we were not able to get interesting results from this assertion. Another assertion stated that a network in a loopy state would end up non-loopy through strong idealization. We were also unable to check this for large enough scopes for the same reason.

4.3.3 Cycles in Idealization

One error that could arise in idealization is the possibility of a cycle in strong idealization. That is, if successive idealizations could result in a state that is isomorphic to a previous state, then the network could run the idealization protocol forever, and not reach a perfect state. Thus, we checked an assertion stating that an equivalent state cannot be reached after a sequence of idealizations beginning and ending in a non-perfect state. We were not able to discover any cycles in idealization. However, limitations of the alloy analyzer did not permit us to investigate this in high enough scopes to make any conclusions.

5 Conclusions and Future Work

This project discovered several ambiguities and potential bugs in the Chord maintenance protocols. In addition to some of the properties that we checked, there are many properties that are guaranteed with

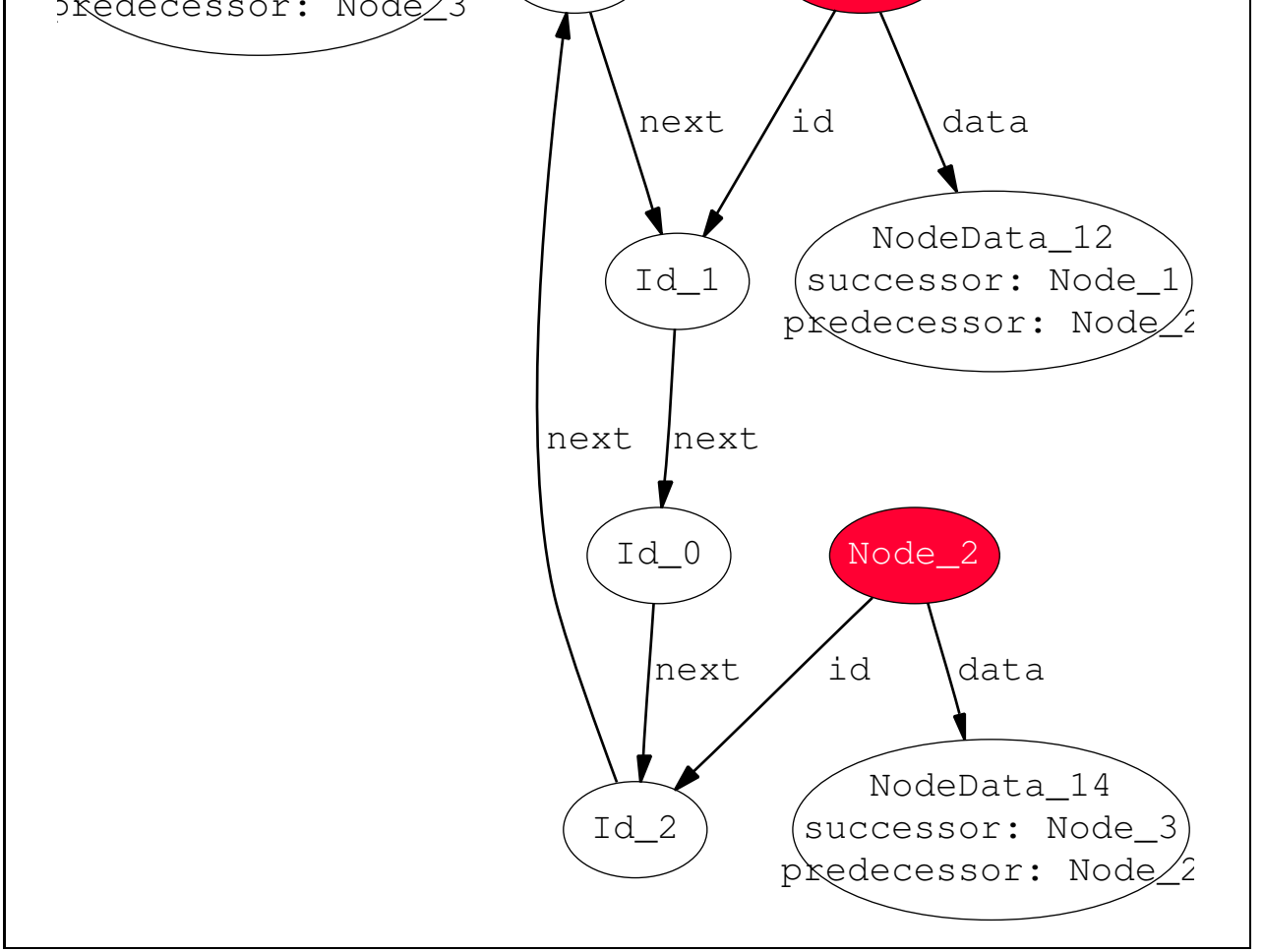

Figure 3: A Chord network in a non-loopy state

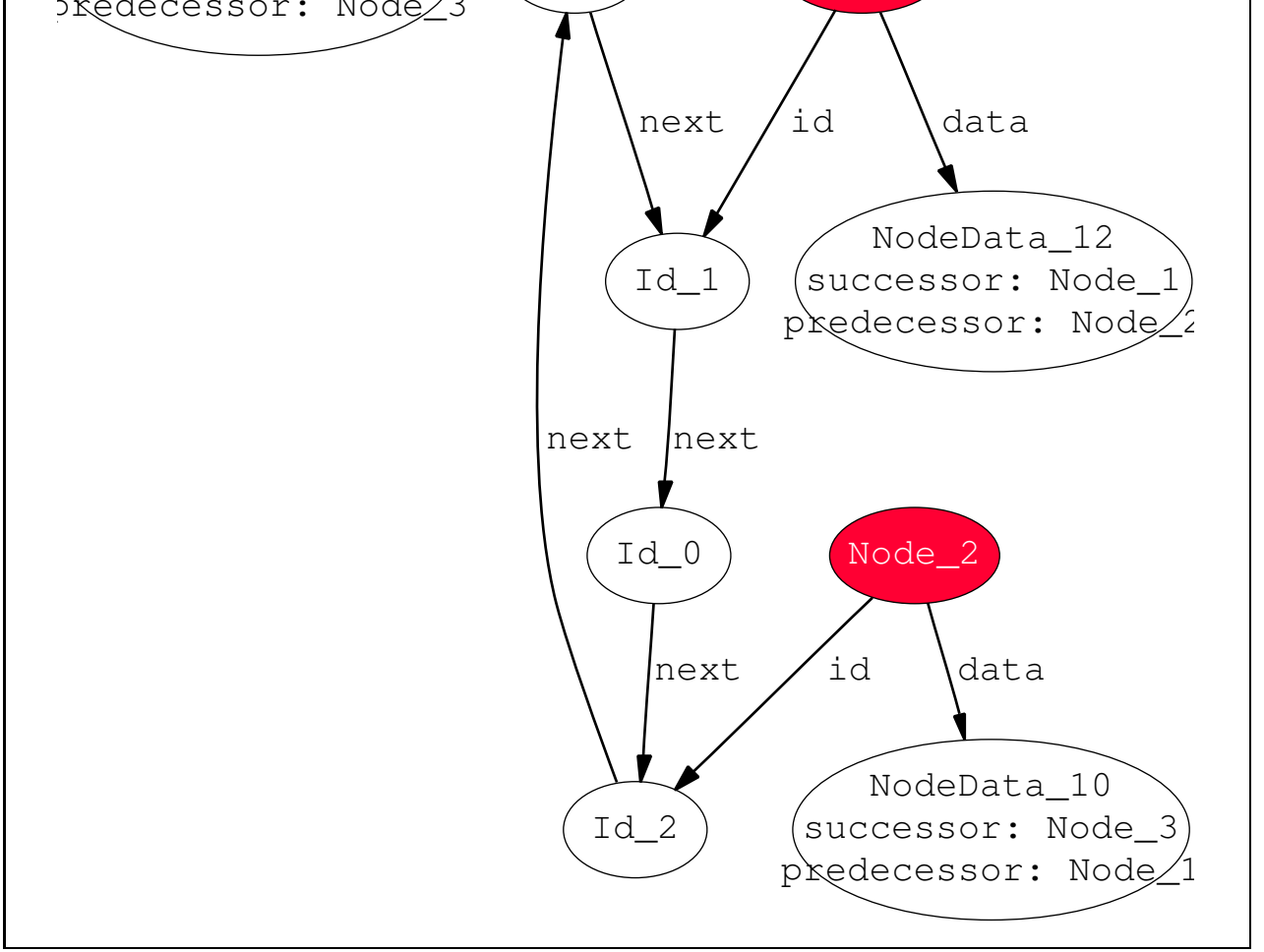

Figure 4: The network after one idealization transition, in a loopy state

tems. To Appear in *Principles of Distributed Computing*, July 2002.

[2] D. Jackson, I. Shlyakhter, M. Sridharan. A Micromodularity Mechanism. In *Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01)*, September 2001.

[3] H. Wee. Alloy model of Chord. Available at http://web.mit.edu/hoeteck/Public/chord

A Pseudocode from [1]

```
// join the system using information from node n'.
n.join(n')
    predecessor := nil;
    successor := n'.fs(n);
    build_fingers(n');

// update finger table via searches by node n'.
n.build_fingers(n')
    i0 := ⌊log(successor − n)⌋ + 1; // first non-trivial finger.

    for each i ≥ i0 index into finger[];
        finger[i] := n'.fs(n + 2^(i−1));

// periodically verify n's successor s, and inform s of n.
// do not run until join() is complete.
n.idealize()
    x := successor.predecessor;
    if (x ∈ (n, successor))
        successor := x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor = nil or n' ∈ (predecessor, n))
        predecessor := n';

n.fix_fingers()
// periodically refresh finger table entries.
n.fix_fingers()
    build_fingers(n);
```

Figure 5: Pseudocode for Chord.

B Alloy Model

See attached Alloy code.

C Example Alloy Analyzer Visualizations

Figure 6 shows a new node joining a one-node network. Figure 7 a non-loopy network that starts out non-perfect, and through weak idealization reaches a perfect state. Figure 8 shows a three-node network that starts out loopy and progresses towards a non-loopy state through strong idealization.

Figure 6: A new node joins a one-node network.

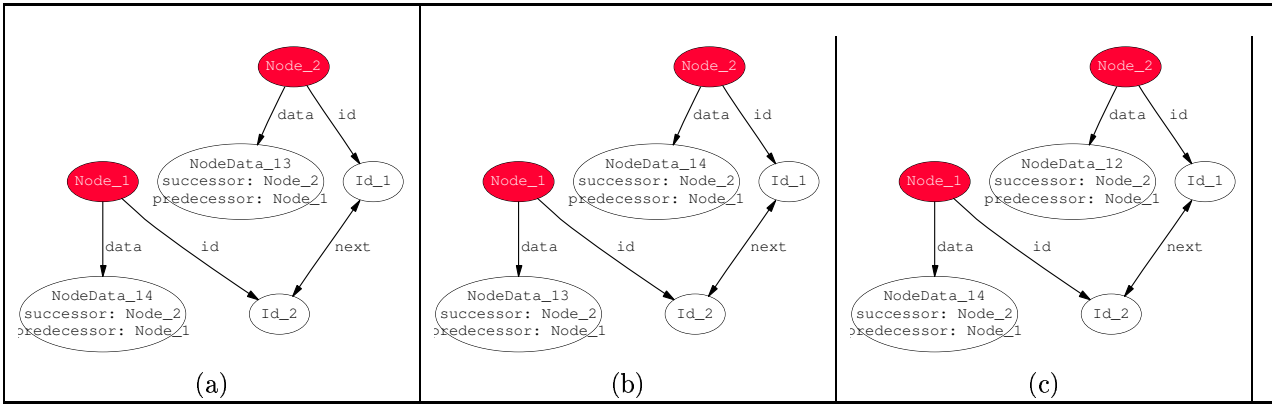

Figure 7: The network starts out non-loopy in (a). After two steps of weak idealization, the network is perfect.

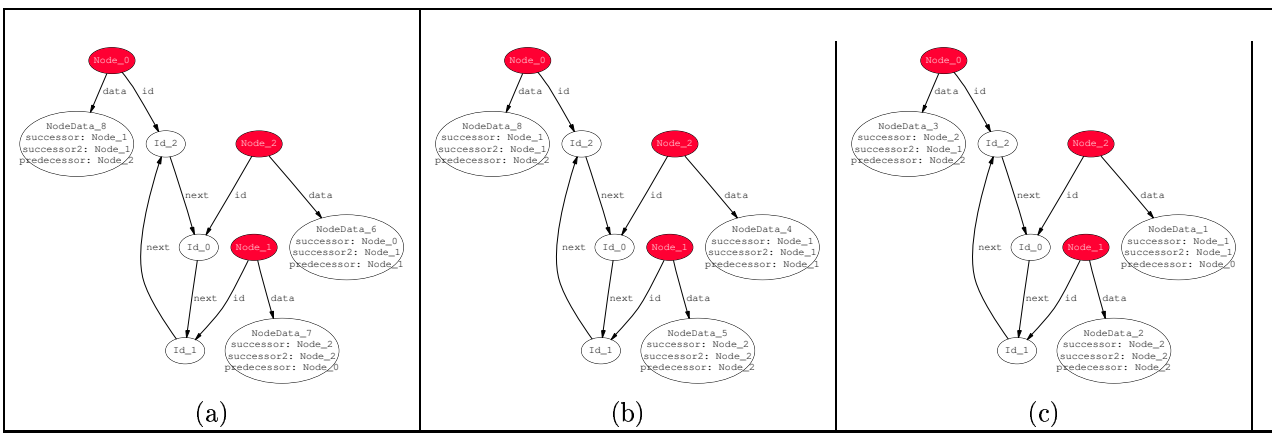

Figure 8: The network starts out loopy in (a). It progresses through two steps of strong idealization in (b) and (c).