Chord: A Peer-to-Peer Protocol

Hoeteck Wee

May 1, 2003

1.1 Motivation

We analyze a distributed lookup service used in Internet peer-to-peer applications [1, 2]. Given that each node only stores partial information about the global state, how do we ensure that lookup queries are correctly passed on from node to node and that they will terminate with the correct answer?

It is often difficult to predict the behavior of large distributed systems and to analyze the performance of algorithms in these systems. Problems often surface with boundary cases and are hard to detect since these cases do not occur often in practice. Errors are also occasionally introduced in the transition from design to implementation.

The present work was initiated to address this problem. By having a small compact model of the Chord system, we could quickly verify the correctness of the lookup routines in small networks and in boundary cases. In addition, this model could be quickly extended to check new routines as well as claims and hypotheses on the behavior of the system (prior to a theoretical analysis) without the need for a complete implementation. They could also be used to check that different variations of the same routine do in fact return the same results. Finally, having a concrete model provides a good middle-ground in the transition from abstract algorithm design to actual code development in providing a formal specification of the algorithm.

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. ACM SIGCOMM 2001, San Diego, CA, August 2001, pp. 149-160.

[2] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*. IEEE/ACM Transactions on Networking, vol. 11 no. 1, February 2003, pp. 17-32.

1.2 Description

We begin with a ring of identifiers in a cyclic ordering. Each node in a Chord network has a unique identifier in the ring, and the main routine in Chord that we are interested in is the one called `find_successor` that starting with any node and given a destination

identifier, finds the node closest to the given identifier (more precisely, the nearest node whose identifier comes *after* the destination identifier).

Each Chord node only stores partial information about the global state. More precisely, it only knows about the identifiers of a small number of nodes in the network, and this information is stored locally in a "finger table". In addition, we require that each node knows about the node closest to it. Upon receiving a find_successor query, a Chord node routes the query to the node closest to the destination identifier in the finger table. This process is repeated until the query reaches the node in the network closest to the destination identifier, when returns a pointer to itself.

In a typical peer-to-peer file-sharing application, Chord nodes correspond to hosts on the network, and files are stored on the host whose node identifier is closest to the identifier assigned to the object (for instance, a hash of the file contents). The find_successor routine can then be used to look up file locations on the network.

In our model, the configuration of the Chord network at each time step is identified with a state. In each state, we have a set of active nodes, and we associate each active node with a piece of node data that stores its finger table. (Note: the notion of an active node is not defined in the references; it's something that we introduced.) Once we fix a state (and therefore the network configuration), the find_successor routine is fully deterministic, and returns a fixed answer for each starting node and identifier. Therefore, we model the routine as a relation find_successor in the node data that maps identifiers to the corresponding nearest nodes.

In specifying the finger table, we drop the constraint that the finger table entries must be spaced out according to some exponential relation. We allow any "finger table" settings, as long as it contains a pointer to the successor node. The lookup routines are still guaranteed to work correctly in this setting, except with a loss of efficiency, which we are not concerned with in our analysis.

1.3 Model Commentary

1.3: Here, we define the identifiers, along with a next relation.

1.4: This reachability constraint is necessary and sufficient to describe a ring-like structure with a single connected component.

1.6: The less-than relation holds if and only if i strictly precedes j in the order starting at from (which corresponds to i lying in the set [from,j) in [2]). Notice that we cannot simply specify that j lies in the transitive closure of the next relation starting at i, because of transitive closure is the entire ring. Instead, we use the transitive closure of the modified relation next', in which we remove next relation pointers into the from node.

1.10: The less-than-equal relation holds if and only if i precedes or equals j in the order starting at from (which corresponds to i lying in the set [from,j] in [2]).

1.14: In an earlier version, nodes were defined to be a subset of id. Here, Node is defined as a separate signature with a unique identifier.

1.17: The NodeData signature models the local state information in each node, as well as the method calls closest_preceding_finger and find_successor, which are modeled as relations. The next field is a pointer to what the node thinks is its successor node, and the finger relation corresponds to entries in the finger table; it maps an identifier to what the node thinks is the closest node to that identifier, or to nothing, if there is no finger table entry corresponding to that identifier.

1.20 By representing the closest_preceding_finger and find_successor as relations within a NodeData signature, we avoid more complex ternary relations that take a node, an identifier (and maybe even a state if we want to model temporal transitions) and return a node. On the other hand, calling these routines becomes a little clumsier since we will have to retrieve the corresponding NodeData. An alternative (as was done in an earlier version) is to represent the closest_preceding_finger and find_successor routines as functions, but the recursive nature of these routines means that we could easily end up requiring a higher-order quantifier.

1.24: The State signature models the overall configuration of a Chord network at a fixed point in time. Each such configuration has an active set of nodes, and a relation data that maps each active node to NodeData. Note that we have a fixed set of Node across different states; what changes from state to state is the set of active nodes in each state, and the binding between the node and its NodeData.

1.25: The notion of inactive nodes is not explicitly presented in [1,2], and does in fact complicate the specifications since we need to add the additional restriction that a node is active. In fact, the omission of this restriction caused a number of subtle bugs in the development of this model!

1.29: This captures exactly the only constraint we impose on the finger table, that it contains a pointer to the (alleged) successor node, and this value, by definition (as that used in [1]), is the same as the value in the next field.

1.34: This describes the correctness condition for the next fields, namely that they do point to the respective successor nodes. The criteria is that for all active nodes: (1) there is no other active node between this active node and its successor; (2) this active node is not the same as its successor unless there is exactly one node in the network; and (3) the successor node is an active node. There are no constraints on inactive nodes.

1.42: This describes the correctness condition for the finger table entries. The criteria is similar to that defined in fun-next-correct, and states that for all active nodes and each

non-degenerate entry in its finger table: (1) the finger table entry points to an active node; and (2) there is no other active node between the identifier and the node in the finger table entry.

1.48: This function essentially defines the closest_preceding_finger relation by specifying the properties that we will like the closest_preceding_finger routine to return, which is identical to the correctness condition for the finger table entries, except that we want no active node between the node in the finger table entry and the identifier (instead of no active node between the identifier and the node in the finger table entry). In [1,2], the closest_preceding_finger routine works differently; it runs a for loop through the finger table entries.

1.57: Here, we define the find_successor relation by using the recursive find_successor routine in [2].

1.67: We require that our definitions of closest_preceding_finger and find_successor does apply in every state.

1.74: Here, we check that we can in fact generate examples containing exactly one node and correct next and finger table entries to ensure that we have not over-constraint the system. This could happen for instance, if any of NextCorrect, FingersCorrect, find_successor or closest-preceding-finger were incorrectly defined, leading to an over-constraint that cannot be satisfied with exactly one node. The analyzer generated an example here.

1.81: We extend the over-constraint checks to systems with 4 identifiers, 2 nodes and one state. The analyzer also generated an example here.

1.88: This describes the correctness conditions that we want the find_successor routine to satisfy, namely that it does in fact return the closest node.

1.96: Here, we check that we can generate examples with correct finger table entries and where find_successor returns the right answer.

1.103: Here, we check that we can generate examples with correct next entries but some incorrect finger table entry.

1.110: FindSuccessorWorks asserts the correctness claim. It says that if the finger table entries are correct for all the active nodes, then find_successor works correctly.

1.114: StrongFindSuccessorWorks asserts a stronger correctness claim. It says that as long as the next entries are correct for all the active nodes, then find_successor works correctly, even when the remaining finger table entries may be incorrect.

1.118: This command instructs the analyzer to evaluate the claim FindSuccessorWorks for all situations involving at most 4 identifiers, 4 nodes and one state. No counterexample is found (even if there are inactive nodes). It is important here that we have pre-

viously generated examples with correct finger table entries to ensure that we are not checking the claim against an empty set.

1.118: This command instructs the analyzer to evaluate the claim StrongerFindSuccessorWorks for all situations involving at most 5 identifiers, 5 nodes and one state. Again, no counterexample is found (even if there are inactive nodes).

```
1.1   module published_systems/chord
1.2
1.3   sig Id {next: Id}
1.4   fact {all i: Id | Id in i.*next}
1.5
1.6   fun less_than (from, i,j: Id) {
1.7     let next' = Id$next - (Id->from) | j in i.^next'
1.8   }
1.9
1.10  fun less_than_eq (from, i,j: Id) {
1.11    let next' = Id$next - (Id->from) | j in i.*next'
1.12  }
1.13
1.14  sig Node {id: Id}
1.15  fact {all disj m,n: Node | m.id != n.id}
1.16
1.17  sig NodeData {
1.18    next: Node,
1.19    finger: Id ->? Node,
1.20    closest_preceding_finger: Id ->! Node,
1.21    find_successor: Id ->! Node
1.22  }
1.23
1.24  sig State {
1.25    active: set Node,
1.26    data: active ->! NodeData
1.27  }
1.28
1.29  fact {
1.30    all s: State | all n: s.active |
1.31      n.s::data.next = n.s::data.finger[n.id.next]
1.32  }
1.33
1.34  fun NextCorrect (s: State) {
1.35    all n: s.active | let succ = n.s::data.next {
```

```
1.36        no n': s.active - n | less_than (n.id, n'.id, succ.id)
1.37        succ != n || #s.active = 1
1.38        succ in s.active
1.39      }
1.40    }
1.41
1.42    fun FingersCorrect (s: State) {
1.43      all nd: s.active.s::data | all start: (nd.finger).Node |
1.44        nd.finger[start] in s.active &&
1.45        no n' : s.active | less_than (start, n'.id, nd.finger[start].id)
1.46    }
1.47
1.48    fun ClosestPrecedingFinger (s: State) {
1.49      all n: s.active | let nd = n.s::data |
1.50        all i: Id | let cpf = nd.closest_preceding_finger[i] {
1.51          no n': (nd.finger[Id] + n) - cpf | less_than (cpf.id, n'.id, i)
1.52          cpf in nd.finger[Id] + n
1.53          cpf.id != i || # s.active = 1
1.54        }
1.55    }
1.56
1.57    fun FindSuccessor(s: State) {
1.58      all n: s.active | let nd = n.s::data | all i: Id {
1.59        nd.find_successor[i] =
1.60        if (less_than_eq (n.id, i, nd.next.id) && n.id != i) || # s.active = 1
1.61        then nd.next
1.62        else
1.63        (nd.closest_preceding_finger[i].s::data.find_successor)[i]
1.64      }
1.65    }
1.66
1.67    fact {
1.68      all s : State {
1.69        ClosestPrecedingFinger(s)
1.70        FindSuccessor(s)
1.71      }
1.72    }
1.73
1.74    fun ShowMe1Node () {
1.75      all s : State | NextCorrect(s) && FingersCorrect(s)
1.76      State.active = Node
```

```
}

run ShowMe1Node for 2 but 1 State, 1 Node

fun ShowMeGood () {
  all s : State | NextCorrect(s) && FingersCorrect(s)
  State.active = Node
}

run ShowMeGood for 4 but 1 State, 2 Node

fun FindSuccessorIsCorrect(s: State) {
  all i: Id | all n: s.active |
    let succ = (n.s::data).find_successor [i] {
      succ in s.active
       no n': s.active | less_than (i, n'.id, succ.id)
    }
}

fun ShowMeCorrectSuccessorEg() {
  State.active = Node
  all s: State | FingersCorrect(s) && FindSuccessorIsCorrect(s)
}

run ShowMeCorrectSuccessorEg for 3 but 1 State

fun ShowMe3 () {
  all s : State | NextCorrect(s) && !FingersCorrect(s)
  State.active = Node
}

run ShowMe3 for 5 but 1 State

assert FindSuccessorWorks {
  all s: State | FingersCorrect(s) => FindSuccessorIsCorrect(s)
}

assert StrongerFindSuccessorWorks {
  all s: State | NextCorrect(s) => FindSuccessorIsCorrect(s)
}
```

```
1.118  check FindSuccessorWorks for 4 but 1 State
1.119  check StrongerFindSuccessorWorks for 4 but 1 State
```

1.4 Variations

In the pseudocode presented in [1, 2], there is some ambiguity as to what the expression (n, n.successor] means in boundary cases where there is exactly one node and n.successor = n. The intention of the authors is that the set includes n. We consider variations of the alloy model with the bug where the set (n, n] does not include n, and observe how it affects the closest_preceding_finger and the find_successor routines.

1.4.1 A Faulty Variant of closest_preceding_finger

Suppose we change ClosestPrecedingFinger as follows:

```
1.120  fun ClosestPrecedingFinger' (s: State) {
1.121    all n: s.active | let nd = n.s::data |
1.122      all i: Id | let cpf = nd.closest_preceding_finger[i] {
1.123        no n': (nd.finger[Id] + n) - cpf | less_than (cpf.id, n'.id, i)
1.124        cpf in nd.finger[Id] + n
1.125        cpf.id != i
1.126      }
1.127  }
```

The only change here is in the last line 1.125, where we removed the clause

```
1.128  || #s.active =1
```

from 1.53. The assertion FindSuccessorWorks still holds for scope up to 4, but ShowMe1Node fails to generate an example! This is an example of a over-constraint, where the inconsistency only shows up when there is exactly one node. What happens here is that the model requires that a closest preceding finger node has a distinct identifier from the input identifier, but this cannot happen if there is exactly one node and if the input identifier equals that of the node.

1.4.2 A Faulty Variant of find_successor

Consider the following pseudocode segment from [2]:

```
1.129  n.find_successor(id)
1.130    if (id in (n, n.successor])
1.131      return n.successor;
1.132    else
1.133      n' = closest_preceding_finger(id);
```

```
1.134          return n'.find_successor(id);
```

In the buggy scenario with a single node, the if loop always terminates at 1.131, leading to an infinite loop.

Consider the corresponding change to FindSuccessor as follows:

```
1.135      fun FindSuccessor'(s: State)
1.136        all n: s.active | let nd = n.s::data | all i: Id {
1.137          nd.find_successor[i] =
1.138          if (less_than_eq (n.id, i, nd.next.id) && n.id != i)
1.139          then nd.next
1.140          else
1.141          (nd.closest_preceding_finger[i].s::data.find_successor)[i]
1.142        }
1.143      }
```

The only change here is in the fourth line 1.138, where we removed the clause || # s.active = 1 from 1.60. Again, if there is exactly one node in the network, the if loop in this case always proceeds to the else clause, and since closest_preceding_finger always returns n (the only node in the network), we end up with a tautological statement:

```
1.144          nd.find_successor[i] = n.s::data.find_successor)[i]
```

This means that there is no additional constraint placed on find_successor, other than that its return type is Node. Now, if there is no distinction between active and inactive nodes, that is, we have exactly one active node in the network and no inactive ones, find_successor will return the right answer due to the type constraint, therefore obscuring the bug. On the other hand, since we have introduced inactive nodes, the assertion FindSuccessorWorks now fails with exactly one active node and some inactive node(s), with find_successor returning an inactive node.