Maintaining the Chord Ring [*]

Xiaozhou Li
Department of Computer Science
University of Texas at Austin
Austin, TX 78712
xli@cs.utexas.edu

Abstract

This short note gives an active and concurrent topology maintenance protocol for the Chord ring. The protocol maintains predecessors and successors, but not fingers. The protocol is a simple extension of a ring maintenance protocol established in a previous paper.

[*]This material is based upon work supported by the National Science Foundation under Grant No. CCR–0310970.

1 Introduction

In a structured peer-to-peer network, nodes (i.e., processes) form a certain topology via their neighbor variables. Over time, nodes may join or leave the network, possibly concurrently. A central problem for structured peer-to-peer networks is topology maintenance, that is, how to properly update neighbor variables to maintain the designated topology when nodes join or leave.

In [1], we have given a protocol, along with its assertional correctness proofs, that maintains a bidirectional ring. The protocol handles both joins and leaves actively (i.e., they update neighbor variables once a join or a leave occurs). In this short note, we show that a simple extension of that protocol actively maintains Chord [3], a ring-based structured peer-to-peer network topology. Chord uses a passive approach for its maintenance [2], where neighbor variables are not immediately updated when a join or a leave occurs, but a repair protocol runs in the background to periodically restore the topology.

2 The Protocol

The protocol in [1] maintains a bidirectional ring where a new node can be inserted between two arbitrary nodes in the ring. The Chord ring, however, has stronger requirements on the arrangements of the nodes in the ring. In Chord, every node has a random binary string as its identifier (or simply ID). The IDs are of the same length and are sufficiently long (say, 128 bits) so that all IDs may be assumed to be unique. Chord arranges nodes in an ID ring with wrap-around. The two basic neighbors that a node has are its predecessor and successor. In addition, a node has fingers, i.e., neighbor variables that allow a node reach another node in the ring. It is worth noting that for Chord to work correctly, it suffices to maintain the predecessors and successors. The fingers improve performance, but do not affect correctness. In what follows, we only discuss how to maintain the predecessors and successors for Chord.

The key difference between maintaining the Chord ring and an arbitrary ring is that when a new node joins the Chord ring, it should be placed between two nodes with proper IDs in the ring. While the protocol in [1] places a new node between two arbitrary nodes, the additional idea needed to maintain the Chord ring is quite straightforward. We simply include the ID of the joining node in the $join$ message and forward the $join$ message using the finger pointers until the node immediately preceding the joining node in the Chord ring is reached.

The protocol that maintains the Chord ring is shown in Figure 1. In the protocol, ϵ denotes the empty string. Compared to the protocol in [1], one noticeable yet nonessential change is the addition of IDs in the message parameters. An alternative presentation of the protocol can remove the need to explicitly mention IDs, but assumes that the reference to a node, say p, includes the ID of p. We opt for explicitly mentioning IDs. Compared to the protocol in [1], several actions are substantially modified.

T_1^j The function $p.genid()$ generates an ID for p. We prefix $genid()$ by "$p.$" to indicate that, in contrast to the $contact()$ function, which is a global function, $genid()$ is locally implementable. We assume that every call to $genid()$ gives a unique ID. This assumption can be provided with high probability, if not absolute guarantee, using some secure hash function like SHA-1. The $contact()$ function returns a pair, a non-out node and its ID, if there is such a node; it returns the calling node and its ID otherwise. A $join$ message takes three parameters, the joining node, the ID of the joining node, and the ID of the receiver of the $join$ message. The reason for including the ID of the receiver is as follows. Since we only assume reliable delivery of messages, when a $join$ message is in transmission, the receiver may leave the ring, and then rejoins with a different ID. Hence, by including the ID of the receiver in the

```
        process $p$
          var   $s : \{in, out, jng, lvg, busy\};\ r, l, t, a : V';\ id, rid, lid$ : identifier
          init  $s = out \wedge r = l = t = \text{nil} \wedge id = rid = lid = \epsilon$
        begin
$T_1^j$         $s = out \rightarrow id := p.genid();\ \langle a, aid\rangle := contact();$
                  if $a = p \rightarrow r, rid, l, lid, s := p, id, p, id, in$
                  [] $a \neq p \rightarrow s := jng;$ send $join(p, id, aid)$ to $a$ fi
$T_1^l$       [] $s = in \rightarrow$
                  if $l = p \rightarrow r, rid, l, lid, s, id := \text{nil}, \epsilon, \text{nil}, \epsilon, out, \epsilon$
                  [] $l \neq p \rightarrow s := lvg;$ send $leave(r, rid)$ to $l$ fi
$T_2^j$       [] rcv $join(a, aid, pid)$ from $q \rightarrow$
                  if $id \neq pid \rightarrow$ send $retry()$ to $a$
                  [] $id = pid \rightarrow \langle b, bid\rangle := p.bestfinger(aid);$
                        if $b = p \wedge s = in \rightarrow$ send $grant(a, aid)$ to $r;\ r, rid, s, t := a, aid, busy, r$
                        [] $b = p \wedge s \neq in \rightarrow$ send $retry()$ to $a$
                        [] $b \neq p \rightarrow$ send $join(a, aid, bid)$ to $b$ fi fi
$T_2^l$       [] rcv $leave(a, aid)$ from $q \rightarrow$
                  if $s = in \wedge r = q \rightarrow$ send $grant(r, id)$ to $a;\ r, rid, s, t := a, aid, busy, r$
                  [] $s \neq in \vee r \neq q \rightarrow$ send $retry()$ to $q$ fi
$T_3$         [] rcv $grant(a, bid)$ from $q \rightarrow$
                  if $l = q \rightarrow$ send $ack(l, lid, id)$ to $a;\ l, lid := a, bid$
                  [] $l \neq q \rightarrow$ send $ack(\text{nil}, \epsilon, \epsilon)$ to $a;\ l, lid := q, bid$ fi
$T_4$         [] rcv $ack(a, aid, qid)$ from $q \rightarrow$
                  if $s = jng \rightarrow r, rid, l, lid, s := q, qid, a, aid, in;$ send $done()$ to $l$
                  [] $s = lvg \rightarrow$ send $done()$ to $l;\ r, rid, l, lid, s, id := \text{nil}, \epsilon, \text{nil}, \epsilon, out, \epsilon$ fi
$T_5$         [] rcv $done()$ from $q \rightarrow s, t := in, \text{nil}$
$T_6$         [] rcv $retry()$ from $q \rightarrow$ if $s = jng \rightarrow s, id := out, \epsilon$ [] $s = lvg \rightarrow s := in$ fi
        end
```

Figure 1: The protocol that maintains the Chord ring.

join message, the receiver can compare its current ID with the ID in the *join* message and accept the message only if they are the same. This checking prevents the situation where a *join* message may be forwarded forever without being able to reach the node with the appropriate ID. An alternative method to avoid the infinite forwarding of a *join* message is to include a time-to-live (TTL) field in the *join* message, and discard the message once the field is decremented to 0.

T_2^j The function $p.bestfinger(aid)$ finds the best finger of p in order to reach aid. We omit how fingers are maintained as they do not affect correctness. Note that the $p.r$ is one of the fingers of p. If the best finger is p itself, then the new node should be inserted between p and $p.r$. In our presentation, the right neighbor is successor and the left neighbor is predecessor.

T_4 If a leaving node has been acknowledged, then it changes its ID to the empty string ϵ, so that in action T_2^j, an *out* node with an ID of ϵ always rejects a join request.

3 Discussions

The correctness proofs for the protocol in Figure 1 are largely similar to those shown in [1] and hence are omitted. We remark that this protocol can be trivially modified to maintain a ring where the nodes are organized based on some other criteria (i.e., those that are not based on node IDs), by changing the implementation of the $bestfinger()$ function. It would be interesting to extend the protocol to maintain fingers as well.

References

[1] X. Li, J. Misra, and C. G. Plaxton. Brief announcement: Concurrent maintenance of rings. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 376–376, July 2004. Full paper available as TR–04–03, Department of Computer Science, University of Texas at Austin, February 2004.

[2] 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, July 2002.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, February 2003.