This is an updated version of V. March, Y. M. Teo, H. B. Lim, P. Eriksson and R. Ayani, *Collision Detection and Resolution in Hierarchical Peer-to-Peer Systems*, Proceedings of the 30th IEEE Conference on Local Computer Networks (LCN 2005), IEEE Computer Society Press, pp. 2–9, Sydney, Australia, November 2005.

Collision Detection and Resolution in Hierarchical Peer-to-Peer Systems

Verdi March[1], Yong Meng Teo[1,2], Hock Beng Lim[2], Peter Eriksson[3] and Rassul Ayani[3]
[1] Department of Computer Science, National University of Singapore
[2] Singapore-MIT Alliance, National University of Singapore
[3] Dept. of Microelectronics and Information Technology, The Royal Institute of Technology, Sweden
E-mail: [verdimar, teoym]@comp.nus.edu.sg

Abstract

Structured peer-to-peer systems can be organized hierarchically as two-level overlay networks. The top-level overlay consists of groups of nodes, where each group is identified by a group identifier. In each group, one or more nodes are designated as supernodes and act as gateways to the nodes at the second level. A collision occurs during join operations, when two or more groups with the same group identifier are created at the top-level overlay. Collisions increase the lookup path length and the stabilization overhead, and reduce the scalability of hierarchical peer-to-peer systems. We propose a new scheme to detect and resolve collisions, and we study the impact of the collision problem on the performance of peer-to-peer systems. Our simulation results show the effectiveness of our scheme in reducing collisions and maintaining the size of the top-level overlay close to the ideal size.

Keywords: DHT, structured overlay network, collision detection and resolution algorithm

1. Introduction

Structured peer-to-peer systems are self-organized distributed systems that provide an efficient and scalable lookup service even though their membership changes dynamically. Such systems organize nodes as structured overlay networks, and map data to nodes based on their identifiers.

There are two main types of structured peer-to-peer architectures: *flat* and *hierarchical*. The flat structure [2, 7, 9, 10, 11, 15] organizes nodes into one overlay network, in which each node has the same responsibility and uses the same rules for routing messages.

The hierarchical structure [3, 5, 6, 8, 12, 13, 14] organizes nodes into a two-level overlay network. The nodes are grouped according to a certain property to achieve a specific objective. For instance, grouping by administrative domains [5, 8, 14] improves the administrative autonomy and reduces latency. Grouping by physical proximity [12, 13] reduces network latency, and grouping by services [6] promotes the integration of services into one system. The groups are logically organized in the top-level overlay network, and each group is identified by a *group identifier*. Also, each group has one or more *supernode* that act as gateways to the nodes at the second-level. Within each group, the nodes can form a second-level overlay network.

Compared to the flat structure, the hierarchical structure reduces the average path length of lookups and improves scalability [3]. In addition, the total number of stabilization messages [1] at the top-level or each of the second-level overlays is lower due to its smaller size compared to the flat structure.

In general, when new nodes join such a hierarchical peer-to-peer system, *collisions of groups* may occur. Due to collisions, the top-level overlay can contain two or more groups with the same group identifier. In a join operation, a new node may fail to locate and join an existing group identified with gid, because the new node joins through a *bootstrap node* from a group identified with gid' and some routing states in the top-level overlay are incorrect. When this happens, the new node creates yet another group identified with gid.

Collisions increase the size of the top-level overlay, which in turn increases the lookup path length and the total number of stabilization messages. In the worst case, col-

1 Stabilization is the process of keeping the topology of an overlay network up to date in the event of dynamic joins, leaves, or fails.

lisions lead to the degeneration of the hierarchical structure into the flat structure, where every node occupies the top-level overlay. If the number of groups is k times larger than the number of ideal groups[2], then the lookup path length is increased only by $O(\log\ k)$ hops, but the total number of stabilization messages is increased by $\Omega(k)$ times.

This paper proposes a new scheme to *detect and resolve* collisions in hierarchical peer-to-peer systems. To reduce overhead, collision detection and resolution are performed together with stabilization. Through simulations, we study the impact of collisions in the top-level overlay and show that (1) if collisions are not resolved then the size of the top-level overlay increases more than ten times, and (2) our proposed scheme reduces the number of collisions significantly and maintains the size of the top-level overlay close to the ideal size.

The rest of this paper is organized as follows. In Section 2, we discuss the related work. Section 3 describes our proposed scheme using Chord [11] as the example. Section 4 presents the analysis of the collision problem. Finally, we conclude this paper in Section 5.

2. Related Work

In hierarchical peer-to-peer systems such as Brocade [14], SkipNet [5], and Mislove et. al. [8], collisions do not occur because a new node always chooses a bootstrap node from the same group. In such systems, nodes are grouped by their administrative domains. Therefore, it is natural for the new node to choose a bootstrap node from the same administrative domain. This guarantees that the new node will not incorrectly create a new group that causes a collision. However, such systems do not address other grouping policies where a new node can choose a bootstrap node from a different group.

In systems such as Garcés-Erice et. al. [3], Diminished Chord [6], Hieras [13], and HONet [12], collisions can occur but the problem is not directly addressed. They assume that collisions can be resolved by mechanisms inherent in the system structure, or the extent of collisions is not studied.

In [6, 13], all nodes in a group are assumed to be supernodes. In such systems, the size of the top-level overlay, with or without collisions, are the same. Hence, the stabilization procedure of the underlying DHT is sufficient to resolve collisions. However, the size of the top-level overlay is larger than in systems where only a subset of nodes become supernodes. Thus, the total number of stabilization messages is larger because more supernodes have to perform stabilization.

2 The size of the top-level overlay with no collision.

In [3, 12], a new node can choose a bootstrap node from a different group. Hence, it is possible that the bootstrap node cannot locate the group associated with the new node, even though the group exists. However, the effect and impact of the collisions were not evaluated.

Our scheme relaxes the assumption that a new node must choose a bootstrap node from the same group and all group members must become supernodes. In addition, our scheme resolves collisions to maintain the size of the top-level overlay close to the ideal size.

3. Proposed Scheme

In a hierarchical peer-to-peer system, each node is assigned a group identifier (gid) and a unique node identifier (nid). Figure 1 shows a hierarchical Chord system (e.g. [3]), where nodes with the same gid form a group and the groups are organized in the top-level overlay network. Each group has one or more supernodes that act as gateways to the second-level nodes. Routing in the top-level and the second-level overlay are based on the group identifier and the node identifier, respectively.

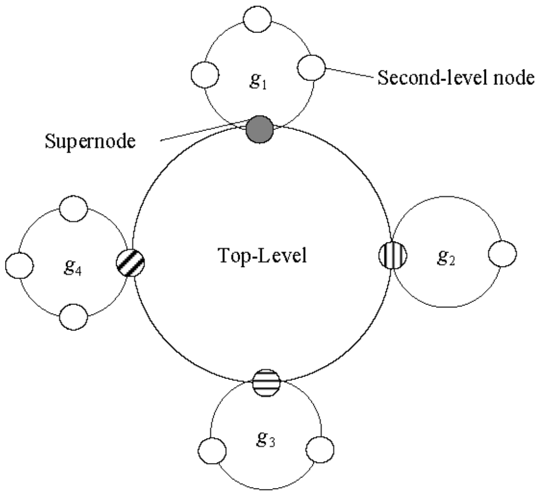

Figure 1. Hierarchical Structured Chord

In the following subsections, we discuss our proposed scheme to detect and resolve collisions. To avoid sending additional messages, collision detection is performed together with stabilization. This is because successful collision detections require the successor pointers in the top-level Chord overlay to be correct, and the correctness of the successor pointers is maintained by stabilization.

The pseudocode follows the same convention as in [11], where the remote procedure calls or variables are preceded by the remote node identifier, while the local procedure calls and variables omit the local node identifier.

3.1. Detecting Collisions

Collisions arise because of join operations. The algorithm for the join operation is shown in Figure 2. A node n, associated with group identifier gid, makes a request to join group g through the bootstrap node n'. With Chord as the top-level overlay, this means finding the successor of gid. If n' can find gid, then n joins group g. If n' returns $gid' > gid$, then n creates a new group with identifier gid. A collision occurs if the new group is created even though a group with identifier gid has already been created. This happens because n' cannot locate the existing group due to $n.gid \neq n'.gid$ and the top-level overlay has not fully stabilized, i.e. some supernodes have incorrect successor pointers. Figure 3 illustrates a collision when n_1 and n_2 belonging to the same group join concurrently.

```
// Node $n$ joins through bootstrap node $n'$
$n$.**join**($n'$)
   $x$ = $n'$.find_successor($gid$);
   **if** ($gid$ == $x.gid$)
      // $x$ is a supernode of group $g$
      join_group($x$);
      $is\_super$ = **false**;
      $supernode = x$
   **else**
      // $n$ creates a new group
      // Can cause a collision
      $pred$ = **nil**;
      $succ = x$;
      $is\_super$ = **true**;
```

Figure 2. Join Operation

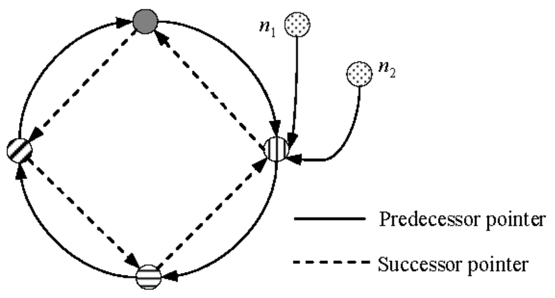

Figure 3. Collision at the Top-Level Overlay

Collision detections are performed during stabilization. We extend Chord's stabilization so that it not only checks and corrects the successor pointer of supernode n, but also detects if n and its new successor should be in the same group.

```
// $n$ periodically verifies its successor pointer,
// and announces itself to the successor.
n.**stabilize**()
   **if** ($succ.is\_super$ == **false**)
      $x$ = $succ$.find_supernode();
      $succ$ = $x$;
   **else**
      $p$ = $succ.pred$;
      **if** (($p \neq n$) **and** ($gid == p.gid$))
         **if** is_collision($p$)
            merge($p$);
      **else if** $p.gid \in (n.gid, succ.gid)$
         $succ$ = $p$;
         $succ$.**notify**($n$);


// $n'$ thinks it might be our predecessor
$n$.**notify**($n'$)
   **if** ( ($pred$ == **nil**) **or** ($pred.is\_super$ == **false**) )
      $pred$ = $n'$;
   **else if** ( ($pred$ == **nil**) **or** ($n' \in (pred, n)$) )
      $pred$ = $n'$;


// Assume one supernode per group
$n$.**is_collision**($n'$)
   **if** ($gid == n'.gid$)
      **return true**
   **else**
      **return false**
```

Figure 4. Collision Detection

Figure 4 presents our collision-detection algorithm, assuming that each group has only one supernode. If $n.succ$ is no longer a supernode then n asks $n.succ$ to find its supernode so that n can update its successor pointer. If $n.succ$ is a supernode then n needs to check if there is a node p located between n and $n.succ$. If p exists, then it becomes the new $n.succ$. Finally, n checks if a collision occurs by comparing the group identifiers of n and $n.succ$. If a collision occurs, we merge the group represented by supernode n with the group represented by supernode $n.succ$.

The following example illustrates the collision-detection process. In Figure 5a, a collision occurred when n_1 and n_2 belonging to the same group join concurrently. In Figure 5b, n_2 stabilizes and causes n_3 to set its predecessor pointer to n_2 (step 1). Then, the stabilization by n_0 causes n_0 to set its successor pointer to n_2 (step 2), and n_2 to set its predecessor pointer to n_0 (step 3). In Figure 5c, the stabilization by n_1 causes n_1 to set its successor pointer to n_2. At this

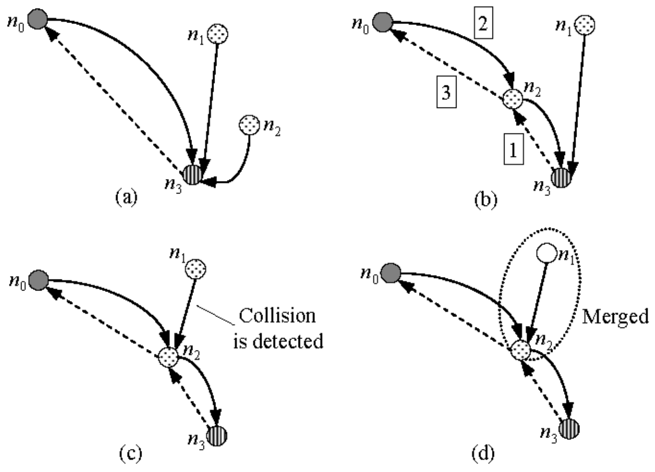

Figure 5. Correcting Successor Pointer

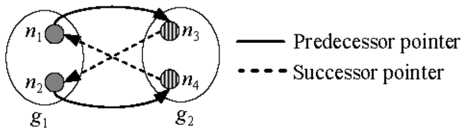

Figure 6. Multiple Supernodes in Each Group

```
$n$.**is_collision**($n'$)
    // L is a set of supernodes in my group
    **if** ($n' \notin L$)
        **return true**
    **else**
        **return false**
```

Figure 7. Collision Detection for Groups with Several Supernodes

time, a collision is detected by n_1 and is resolved by merging n_1 to n_2.

If each group contains more than one supernodes, then the `is_collision` routine shown in Figure 4 may incorrectly detect collisions. Consider the example in Figure 6. When n_1 stabilizes, it will incorrectly detect a collision with n_2 because $n_1.succ.pred = n_2$ and $n_1.gid = n_2.gid$. An approach to avoid this problem is for each group to maintain a set of its supernodes [3, 4] so that each supernode can accurately decide whether a collision has occurred. The modified collision-detection algorithm is shown in Figure 7.

```
// Set predecessor of $n$ to $n'$
$n$.**replace_predecessor**($n'$)
    $pred = n'$;


// Set successor of $n$ to $n'$
$n$.**replace_successor**($n'$)
    $succ = n'$;
```

Figure 8. Announce Leave to Neighbors

3.2. Resolving Collisions

To resolve collisions, groups associated with the same gid are merged. After the merging, some supernodes become ordinary nodes depending on the group policy. Before a supernode changes its state into a normal node, the supernode notifies its successors and predecessors to update their pointers (Figure 8). Nodes in the second level also need to be merged to the new group. We propose two merging approaches, *supernode initiated* and *node initiated*.

3.2.1. Supernode Initiated To merge a group $n.gid$ with another group $n'.gid$, the supernode n notifies its second-level nodes to join group $n'.gid$ (Figure 9). The advantage of this approach is that second-level nodes join a new group as soon as the collision is detected. However, n needs to track its group membership, which may not always be correct. If n has only partial knowledge of group membership, some nodes in the second-level can become orphans.

```
// All nodes joins the group where
// $n'$ is the supernode
$n$.**merge**($n'$)
    $is\_super$ = **false**

    // Since $n$ leaves the top-level overlay,
    // its $pred$ and $succ$ must be notified
    $succ$.replace_predecessor($pred$);
    $pred$.replace_successor($succ$);
    $pred = succ =$ **nil**;

    $n'$.join_group($n$);
    $g$ = group where $n$ is the supernode
    **for each** node $x \in g$ **do**
        $x$.join_group($n'$);
        $x.supernode = n'$
```

Figure 9. Supernode Initiated

3.2.2. Node Initiated In node-initiated merging, each second-level node m periodically checks that its known su-

```
// Supernode $n$ joins another group,
// ignoring its second-level nodes
$n$.**merge**($n'$)
    $is\_super$ = **false**;

    // Since $n$ leaves the top-level overlay,
    // its $pred$ and $succ$ must be notified
    $succ$.replace_predecessor($pred$);
    $pred$.replace_successor($succ$);
    $pred = succ =$ **nil**;


// Second-level node $n$ periodically
// verifies its supernode pointer
$m$.**check_supernode**()
    **if** ($supernode.is\_super$ == **false**)
        $x$ = $supernode$.find_supernode();
        $supernode$ = $x$;
        join_group($x$);
```

Figure 10. Node Initiated

pernode n' is still a supernode (Figure 10). If n' is no longer a supernode, then m will ask n' to find the correct supernode and join a new group through the new supernode. This approach does not require supernodes to track group membership. However, after a collision is detected, it may take some time before nodes join a new group.

4. Analysis

Consider the total number of stabilization messages of the top-level Chord overlay at time t. Let G denote the number of groups. *Without collisions*, the total number of stabilization messages S is $O(G \cdot \log^2 G)$ because there are G groups that perform stabilization, each group corrects $O(\log G)$ fingers, and the cost of correcting each finger is $O(\log G)$. *With collisions*, let G_C denote the number of groups and let the cost of stabilization be S_C. If $G_C = kG$, where k is a constant, then S_C is $\Omega(kS)$ because there are kG groups that perform periodic stabilization.

To evaluate the effectiveness of our proposed scheme, we compare hierarchical systems of N nodes without collision detection and resolution (*without detect & resolve*) and with collision detection and resolution (*detect & resolve*) via simulations. We assume that each group contains one supernode. To resolve collisions, we use the supernode-initiated approach. Since the emphasize of this experiment is to study the collisions at the top-level Chord and the purpose of collision resolution is to ensure that the second-level overlays are correct after a collision, the choice of collision-resolution approach will not significantly affect the result of this experiment.

p	*Without Detect & Resolve*		*Detect & Resolve*	
	G**=1000**	G**=2000**	G**=1000**	G**=2000**
30	2,884	4,588	71	51
60	3,221	4,787	267	202
120	3,985	5,613	955	966
240	7,163	8,411	2,026	1,955
480	9,130	12,116	5,180	3,684

(a) N = 50,000

p	*Without Detect & Resolve*		*Detect & Resolve*	
	G**=1000**	G**=2000**	G**=1000**	G**=2000**
30	3,622	5,892	73	85
60	3,744	6,120	145	857
120	5,481	6,529	717	1,681
240	8,111	9,826	1,428	4,228
480	11,566	15,250	3,313	7,091

(b) N = 100,000

Table 1. Number of Collisions

Our simulations are based on the Chord simulator included in Chord SDK [1]. The average inter-arrival time of nodes is exponentially distributed with a mean of 1 second. Each supernode maintains a successor pointer, a predecessor pointer, and $O(\log G_C)$ fingers. In addition, each supernode periodically invokes the stabilization procedure. With a stabilization period parameter of p (in seconds), the stabilization period is uniformly distributed in the interval [$0.5p$, $1.5p$]. Each stabilization corrects the successor pointer and one of the fingers. The link latency between nodes is exponentially distributed with a mean of 50 ms and the request-processing time by each node is uniformly distributed between 5 and 15 ms.

4.1. Extent of Collisions

We simulated hierarchical systems with 50,000 and 100,000 nodes (N). For the number of groups G in the top-level overlay, we chose the values of 1,000 and 2,000. Thus, we evaluated four different system configurations. Table 1 shows the extent of collisions from measuring the total number of collisions for different values of the stabilization period p.

Without resolving collisions, the number of collisions is about 3 to 12 times G. With frequent stabilization, our scheme significantly reduces the number of collisions. But as p increases, the number of collisions grows because of the reduced frequency of collision resolution.

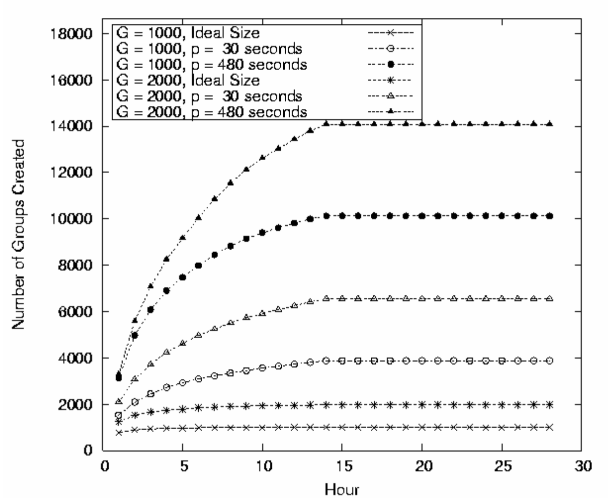

Figure 11. Size of Top-Level Overlay *Without Detect & Resolve* (N = 50,000)

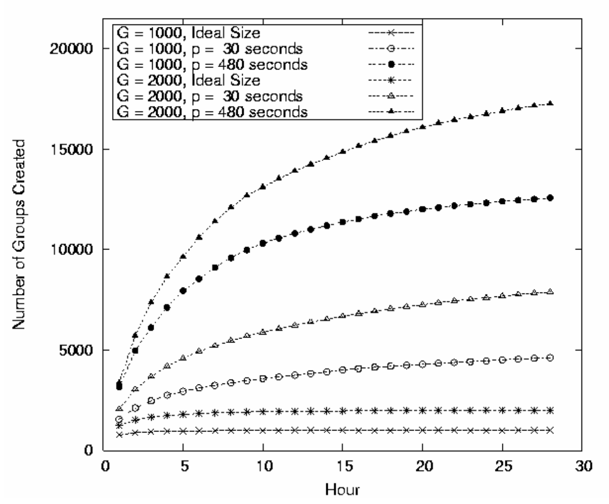

Figure 13. Size of Top-Level Overlay *Without Detect & Resolve* (N = 100,000)

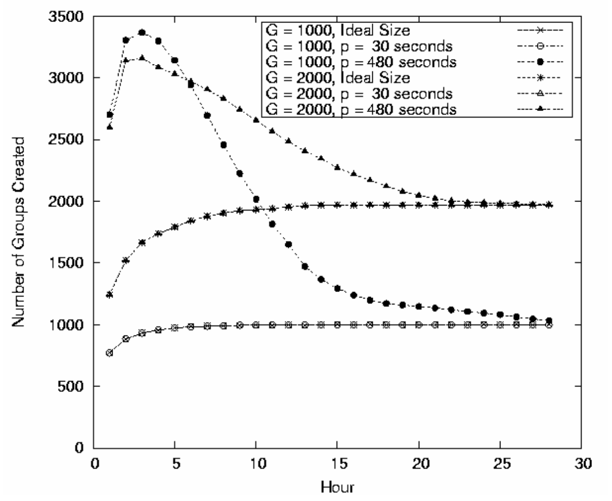

Figure 12. Size of Top-Level Overlay with *Detect & Resolve* (N = 50,000)

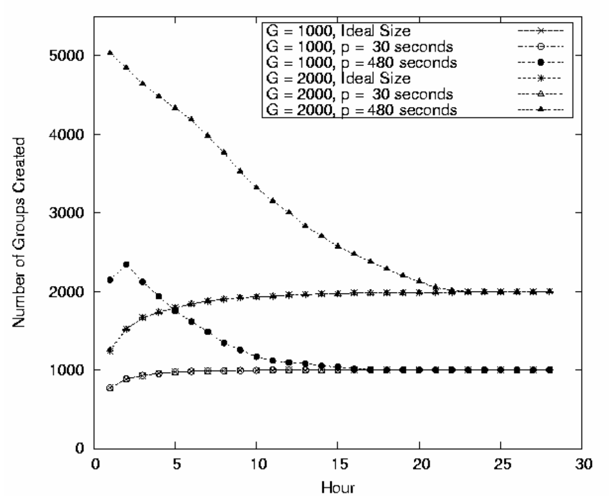

Figure 14. Size of Top-Level Overlay with *Detect & Resolve* (N = 100,000)

4.2. Impact of Collisions

The impact of collisions is measured by the growth in the size of the top-level overlay. Figures 11–14 show the number of groups at an interval of one hour. Without collision resolution, the size of the top-level overlay grows to about 3 to 12 times G (see Figures 11 and 13) because the additional groups caused by collisions will remain in the top-level overlay. If the size of the top-level overlay increases by 12 times, then the *average* lookup path length is increased only by $\frac{1}{2} \log\ 12 = 1.8$ hops, but the total number of stabilization messages is increased $\Omega(12)$ times. On the other hand, *detect & resolve* merges the colliding groups so that the size of the overlay converges to that of the ideal size G (see Figures 12 and 14).

With a larger p, the stabilizations are performed less frequently. Thus, more stabilization rounds are required to correct the successor pointers. Since our scheme is performed together with the stabilizations to reduce overhead, it takes a longer time to reduce the size of the top-level overlay close to the ideal size, as shown in our results.

p	N = 50,000		N = 100,000	
	G=1,000	G=2,000	G=1,000	G=2,000
30	334	562	347	850
60	1,364	1,665	1,096	4,093
120	3,460	6,687	3,005	7,452
240	7,712	11,074	4,937	14,951
480	16,813	16,124	9,061	29,125

Table 2. Average Time to Detect a Collisions, in Seconds

p	N = 50,000		N = 100,000	
	G=1,000	G=2,000	G=1,000	G=2,000
30	0.02	0.01	0.02	0.01
60	0.08	0.04	0.04	0.14
120	0.24	0.17	0.13	0.26
240	0.28	0.23	0.18	0.43
480	0.58	0.30	0.29	0.46

Table 3. Ratio of Number of Collisions

4.3. Efficiency and Effectiveness

The efficiency and effectiveness of our scheme depends on the frequency of detection and resolution, which is determined by the stabilization period p.

4.3.1. Detection The efficiency of collision detection is measured by the average time required to detect a collision. This is defined as the period between a join and a stabilization procedure that detects the collision. It is desirable to detect collisions as soon as possible to minimize the impact of collisions. Table 2 shows that the average time to detect collisions increases as p increases. From the results, the ratio of the collision detection time to the stabilization interval (p) is more than ten times. This indicates that the collision detection time is quite significant.

Table 3 shows the effectiveness of our scheme. We compute the ratio of the number of collisions in the *detect & resolve* case to the number of collisions in the *without detect & resolve* case. With frequent stabilization when p is 30 seconds, the value of this ratio is 0.02 for $N = 50,000$ and $G = 1,000$. This means that our scheme reduces the number of collisions by 98%. As p increases, the effectiveness of the scheme decreases. However, even when p is 480 seconds, our scheme still reduces the number of collisions by at least 40%.

4.3.2. Resolution There are two main factors that affect the cost of collision resolutions. The first factor is the number of groups and nodes to be merged. Table 4 shows the average number of nodes corrected in each collision resolution. The effectiveness of collision resolution improves with a higher frequency of stabilization. Overall, our results indicate that the average number of nodes corrected can be reduced to less than 10% of the average group size.

p	N = 50,000		N = 100,000	
	G=1,000	G=2,000	G=1,000	G=2,000
30	2.9	2.6	2.9	3.0
60	5.1	3.7	4.8	6.1
120	8.5	9.0	8.0	9.2
240	16.4	12.7	11.4	13.6
480	19.1	16.0	18.0	23.9

Table 4. Average Number of Nodes Affected by a Collision

The second factor is the overhead of correcting stale finger pointers and the cost of updating fingers to point to the new group after merging. As each group is pointed by $O(\log\ G_C)$ groups and the correction of each finger pointer required $O(\log\ G_C)$, the total cost to update the fingers pointing to the merged group is $O(\log^2\ G_C)$.

Summary: The results in this section suggest that the efficiency and effectiveness of our scheme can be improved by having more frequent detections and resolutions to reduce collisions. This will reduce the cost of correcting collisions.

5. Conclusion

We have presented a scheme to detect and resolve collisions in hierarchical peer-to-peer systems. Collisions increase the size of the top-level overlay by a factor k, which increases the lookup path length by only $O(\log\ k)$ hops, but increases the total number of stabilization messages by $\Omega(k)$ times. Our scheme performs collision detection together with stabilization to avoid introducing additional messages. Two approaches are proposed to resolve collisions: supernode-initiated merging and node-initiated merging.

Our simulation results show that if collisions are not resolved, the size of the top-level overlay increases more than ten times. With our scheme, the number of collisions is reduced at least by 40%. In addition, the size of the top-level overlay remains close to the ideal size; otherwise it can be up to 12 times larger, which increases the average lookup path length by 1.8 hops and the total number of stabilization messages by $\Omega(12)$ times. The results also reveal the importance of minimizing collisions as it takes several stabilizations to detect collisions. Thus, more frequent stabilization reduces collisions and keeps the top-level overlay close to the ideal size.

References

[1] The Chord Project. `http://www.pdos.lcs.mit.edu/chord`.

[2] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *Proc. of the 3rd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGRID'03)*, pages 344–350, May 2003.

[3] L. Garcés-Erice, E. W. Biersack, P. A. Felber, K. W. Ross, and G. Urvoy-Keller. Hierarchical peer-to-peer systems. In *Proc. of ACM/IFIP Intl. Conf. on Parallel and Distributed Computing (Euro-Par 2003)*, Aug. 2003.

[4] I. Gupta, K. Birman, P. Linga, A. Demers, and R. V. Renesse. Kelips: Building an efficient and stable p2p DHT through increased memory and background overhead. In *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 160–169, Feb. 2003.

[5] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, pages 113–126, Mar. 2003.

[6] D. R. Karger and M. Ruhl. Diminished Chord: A protocol for heterogeneous subgroup. In *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems (IPTPS'04)*, pages 288–297, Feb. 2004.

[7] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of the 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS'02)*, pages 53–65, Mar. 2002.

[8] A. Mislove and P. Druschel. Providing administrative control and autonomy in structured peer-to-peer overlays. In *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems (IPTPS'04)*, pages 162–172, Feb. 2004.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM 2001*, pages 161–172, Aug. 2001.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, Nov. 2001.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM 2001*, pages 149–160, Aug. 2001.

[12] R. Tian, Y. Xiong, Q. Zhang, B. Li, B. Y. Zhao, and X. Li. Hybrid overlay structure based on random walks. In *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems (IPTPS'05)*, Feb. 2005.

[13] Z. Xu, R. Min, and Y. Hu. HIERAS: A DHT based hierarchical p2p routing algorithm. In *Proc. of the 2003 Intl. Conf. on Parallel Processing (ICPP'03)*, pages 187–194, Oct. 2003.

[14] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 34–44, Mar. 2002.

[15] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Department, UC Berkeley, Apr. 2001.