Chapter 7
Hierarchical Structured Peer-to-Peer Networks

Yong Meng Teo  
National University of Singapore, Singapore

Verdi March  
National University of Singapore, Singapore

Marian Mihailescu  
National University of Singapore, Singapore

ABSTRACT
Structured peer-to-peer networks are scalable overlay network infrastructures that support Internet-scale network applications. A globally consistent peer-to-peer protocol maintains the structural properties of the network with peers dynamically joining, leaving and failing in the network. In this chapter, the authors discuss hierarchical distributed hash tables (DHT) as an approach to reduce the overhead of maintaining the overlay network. In a two-level hierarchical DHT, the top-level overlay consists of groups of nodes where each group is distinguished by a unique group identifier. In each group, one or more nodes are designated as supernodes and act as gateways to nodes at the second level. Collisions of groups occur when concurrent node joins result in the creation of multiple groups with the same group identifier. This has the adverse effects of increasing the lookup path length due to a larger top-level overlay, and the overhead of overlay network maintenance. We discuss two main approaches to address the group collision problem: collision detection-and-resolution, and collision avoidance. As an example, they describe an implementation of hierarchical DHT by extending Chord as the underlying overlay graph.

INTRODUCTION
Structured peer-to-peer systems or distributed hash tables (DHT) are self-organizing distributed systems designed to support efficient and scalable lookups with dynamic network topology changes. Nodes are organized as structured overlay networks, and data is mapped to nodes in the overlap network based on their identifier. There are two main types of structured peer-to-peer architectures: flat and hierarchi-
Hierarchical Structured Peer-to-Peer Networks

cal. A flat DHT (Alima, 2003; Ratnasamy, 2001; Stoica, 2001; Rowstron, 2001; Maymounkov, 2002; Zhao, 2001) organizes nodes into one overlay network, in which each node has the same responsibility and uses the same rules for routing messages. On the other hand, a hierarchical DHT organizes nodes into a multi-level overlay network with the primary aim of reducing the maintenance overhead of its overlay network. In a peer-to-peer system, peers join and leave the system dynamically. A process called stabilization updates the routing information maintained in each peer so as to keep the overlay network up-to-date (Ghinita, 2006).

A hierarchical DHT employs a multi-level overlay network where the top-level overlay consists of logical groups (Garcés-Erice, 2003; Harvey, 2003; Karger, 2004; Mislove, 2004; Tian, 2005; Xu, 2003; Zhao, 2003). Each group, which consists of a number of nodes, is assigned a group identifier with a specific objective such as improving administrative autonomy (Harvey, 2003; Mislove, 2004; Zhao, 2003), reducing network latency (Tian, 2005; Xu, 2003), and integrating various services into one system (Karger, 2004). Within a group, one or more nodes are selected as supernodes to act as gateways to nodes in the groups. Within each group, nodes can further form a second-level overlay network.

In this chapter, we discuss the organization of a hierarchical DHT with the aim of reducing its overlay maintenance overhead. Using a two-level hierarchical Chord as an example, the top-level overlay network consists of groups with distinct group identifiers. However, collision of groups occurs when two or more groups are created with the same group identifier. Collisions increase stabilization overhead and degrade lookup performance. To address the collision problem, we discuss two main approaches: collision detection-and-resolution, and collision avoidance.

The rest of this chapter is organized as follows. Section 2 presents an overview of flat DHT using Chord as the example (Stoica, 2001). Three main approaches to reduce routing maintenance overhead are introduced: hierarchical DHT, varying frequency of stabilizations, and varying number of routing states. Extending Chord into a hierarchical Chord DHT, Section 3 discusses two differing approaches in addressing the collision problem, namely, collision detection-and-resolution, and collision avoidance. Section 4 summarizes this chapter and discusses open issues.

DISTRIBUTED HASH TABLES

Distributed hash table (Gummadi, 2003; Hsiao, 2003; Ratnasamy, 2002; Stribling, 2004) is a decentralized lookup scheme designed to provide scalable lookups, i.e., shorter lookup path length with high result guarantee and reduced number of false negative answers. The DHT protocol provides an interface to retrieve a key-value pair. A key is an identifier assigned to a resource; traditionally this key is a hash value associated with the resource. A value is an object to be stored into DHT; this could be the shared resource itself such as a file, an index or a pointer to a resource, or a resource metadata. An example of a key-value pair is <SHA1(file name), http://peer-id/file>, where the key is the SHA1 hash of the file’s name and the value is the address (location) of the file.

To support scalable lookups with high result guarantee, DHT exploits the following:

1. **Key-to-node mapping:** Assuming that keys and nodes share the same identifier space, DHT maps key $k$ to node $n$ where $n$ is the node closest to $k$ in the identifier space; we refer to $n$ as the responsible node of $k$. The key-to-node mapping improves result guarantee because searching for a key-value pair equals to locating the node responsible for the key (Loo, 2004).
2. **Data-item distribution**: Key-value pairs, also called *data items*, with key equals to *k* are *stored* at node *n* independent of the owners of these key-value pairs. This is implemented in DHT by a *store* operation (Dabel, 2003; Rhea, 2005). The concept of data-item distribution has been further exploited for various optimizations, including load balancing (Godfrey, 2004; Godfrey, 2005; Karger, 2004) and high availability (Dabek, 2001; Ghodsi, 2005a; Kubiatowicz, 2000; Landers, 2004; Leslie, 2006).

3. **Structured overlay network**: Searching and storing a key-value pair requires *routing* of the request to a responsible node. To achieve scalable routing, nodes are organized as structured overlay network. A structured overlay network exhibits two main properties: (i) it resembles a graph and is organized into a network topology such as a ring (Rowstron, 2001; Stoica, 2001), a torus (Ratnasamy, 2001), or a tree (Aberer, 2003; Maymounkov, 2002), and (ii) each node uses its identifier to position itself in the structured overlay network. The tradeoff in different overlay topologies are routing performance and overhead of *maintaining* routing states.

As an example of DHT implementation, we discuss Chord which supports *O*(log *N*)-hop lookup path length and maintains *O*(log *N*) routing states per node, where *N* denotes the total number of nodes (Stoica, 2001). Chord organizes nodes as a ring that represents an *m*-bit one-dimensional circular identifier space, and as a consequence, all arithmetic is modulo 2^*m*. To form a ring overlay, each node *n* maintains two pointers to its immediate neighbors as shown in Figure 1(a). The successor pointer points to *successor*(n), the immediate clockwise neighbor of *n*. Similarly, the predecessor pointer points to *predecessor*(n), the
Hierarchical Structured Peer-to-Peer Networks

In Chord, every piece of data is assigned an $m$-bit identifier called a key. Key $k$ is then mapped onto $\text{successor}(k)$, the first node whose identifier is equal to or greater than $k$ in the identifier space (Figure 1(b)). Thus, node $n$ is responsible for keys in the range of $(\text{successor}(n), n]$, i.e. keys that are greater than $\text{predecessor}(n)$ but smaller than or equal to $n$. For example, node 32 is responsible for all keys in $(21, 32]$. All key-value pairs whose key equals to $k$ are then stored on $\text{successor}(k)$ regardless of who owns the key-value pairs. This distribution of keys is called data-item distribution.

Finding key $k$ implies that we route a request to $\text{successor}(k)$. To achieve scalable routing, each node $n$ maintains a finger table of $m$ entries as shown in Figure 1(c). Each entry in this table is also called a finger. The $i$th finger of $n$ is denoted as $n.finger[i]$ and points to $\text{successor}(n + 2^{i-1})$, where $1 \leq i \leq m$. Note that the first finger is also the successor pointer while the largest finger divides the circular identifier space into two halves. When $N < 2^m$, the finger table consists of only $O(\log N)$ unique entries (Stoica, 2001).

By utilizing finger tables, Chord locates $\text{successor}(k)$ in $O(\log N)$ hops with high probability (Stoica, 2001). Intuitively, the process resembles a binary search where each step halves the distance to $\text{successor}(k)$. Thus, each node $n$ forwards a request to the nearest known preceding node of $k$. This is repeated until the request arrives at $\text{predecessor}(k)$, the node whose identifier precedes $k$, which will forward the request to $\text{successor}(k)$. Figure 1(d) shows an example of finding $\text{successor}(54)$ initiated by node 8. Node 8 forwards the request to its sixth finger which points to node 48. Node 48 is the predecessor of key 54 because its first finger points to node 56 and $48 < 54 \leq 56$. Finally, node 48 will forward the request to node 56.

Figure 2 illustrates the construction of a Chord ring. A new node $n$ joins a Chord ring by locating its own successor. Then, $n$ inserts itself between $\text{successor}(n)$ and the predecessor of $\text{successor}(n)$, illustrated in Figure 2(a). The key-value pairs stored on $\text{successor}(n)$, whose key is less than or equal to $n$, is migrated to node $n$ (Figure 2(b)). Because the join operation invalidates the ring overlay, every node periodically invokes a maintenance process called stabilization to correct its successor and predecessor pointers (Figure 2(c)), and its remaining fingers.

A number of approaches have been proposed to reduce the maintenance overhead of DHT. We classify these approaches into three main categories: hierarchical DHT, varying frequency of stabilizations, and varying number of routing states. The last two approaches are applicable directly to both flat and hierarchical DHTs.
HIERARCHICAL DHT

In hierarchical DHT, nodes are organized as a two-level overlay network. The top-level overlay consists of logical groups of nodes, where each group is identified by a group identifier \( (\text{gid}) \). In each group, one or more nodes are designated as supernodes and act as gateways to the nodes at the second level. Each node is assigned an identifier consisting of two subfields: a unique node identifier as is common in DHT to distinguish different peers, and a group identifier to reflect the node’s group. For example, in compute-cycle sharing, a group identifier denotes the type of shared resource or processor type (March, 2007). Grouping of shared resources by processor types facilitates resource discovery and allocation.

Figure 3 shows a hierarchical Chord system (Garcés-Erice, 2003), where nodes with the same \( \text{gid} \) form a group and the groups are organized in the top-level overlay network. Routing in the top-level and the second-level overlay are based on the group identifier and the node identifier, respectively.

A hierarchical DHT groups nodes based on various properties to achieve specific objectives. Examples include:

1. Grouping by administrative domains improves the administrative autonomy and reduces latency (Harvey, 2003; Mislove, 2004; Zhao, 2002);
2. Grouping by physical proximity reduces network latency (Tian, 2005; Xu, 2003);
3. Grouping by services promotes the integration of services into one system (Karger, 2004).

In terms of topology maintenance, the hierarchical structure has the following advantages compared to the flat structure:

1. **Lower overhead of overlay maintenance:** Maintenance of structured overlay network involves the correction of nodes’ routing states to adapt to dynamic events of node joining, leaving, or failing. Since the hierarchical structure partitions nodes into multiple overlays, each of which is smaller than a flat overlay, maintenance messages are routed only in one of these smaller overlays. This speeds up the correction of routing states while reducing the number of stabilization messages processed by each node.
2. **Isolation of churn:** Topology changes within a group due to churn, i.e., continuous changes due to node joins, leaves, or failures, do not affect the top-level overlay or other groups. Stable overlay topologies improve the result guarantee of DHT lookups.

However, when new nodes join such a hierarchical DHT system, **collisions of groups** may occur. Collisions result in the top-level overlay containing two or more groups with the same group identifier, and increase the size of the overlay. For example, in a join operation, a new node firstly requests a bootstrap node to locate an existing group identified with \( \text{gid} \). However, when the bootstrap node belongs to another group \( \text{gid}' \) and some routing states in the top-level overlay are incorrect, the bootstrap node may fail to locate group \( \text{gid} \). Thus, instead of joining group \( \text{gid} \), the new node creates a new group with the same \( \text{gid} \).

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, collisions 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 \( c \) times larger than the number of ideal groups\(^1\), the lookup path length is increased
There are two main approaches to address the problem of collisions in hierarchical DHT systems:

1. **Collision detection and resolution:** With this approach, collisions are allowed to occur but it is the responsibility of the hierarchical DHT systems to detect collisions and merge these groups into a single group (March, 2005). In systems such as hierarchical Chord-based DHT (Garcés-Erice, 2003), Diminished Chord (Karger, 2004), Hieras (Xu, 2003) and HONet (Tian, 2005), collisions can occur but the problem is not directly addressed. They assume that collisions can be resolved by mechanisms inherent in the system structure, and the extent of collisions is not studied.

2. **Collision avoidance:** In hierarchical DHT systems, schemes can be devised to ensure that collisions do not occur. This can be achieved through collision-free join protocols (Teo, 2008) or collision-free grouping policies (Harvey, 2003; Karger, 2004; Mislove, 2004; Xu, 2003; Zhao, 2003). Collision-free join protocol such as in (Teo, 2008) uses the predecessor node to serialize the join lookup operation. All nodes in the overlay network maintain accurate fingers and new groups are reflected instantaneously by the predecessor supernode. The leave protocol is also modified to ensure the correctness of the finger table, the successor pointers, and the predecessor pointers. Thus, a departing supernode notifies its successor and predecessor to update their pointers accordingly. As long as the fingers are maintained in an accurate state, collisions do not occur.

   - In hierarchical DHT such as Brocade (Zhao, 2003), SkipNet (Harvey, 2003), and hierarchical Scribe (Mislove, 2004), 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 domain. Therefore, it is natural for the new node to choose a bootstrap node from the same administrative domain. This grouping policy guarantees that multiple groups with the same group identifier are not created. However, such systems do not address other grouping policies that can introduce collisions, i.e., when a new node is bootstrapped from a node in
In (Karger, 2004; Xu, 2003), all nodes in a group are assumed to be supernodes. Hence, collisions do not occur. However, the size of the top-level overlay, with or without collisions, is the same. In addition, the top-level overlay is larger than systems where only a subset of nodes becomes supernodes. Thus, the total number of stabilization messages is increased because more supernodes have to perform stabilization.

### VARYING FREQUENCY OF STABILIZATION

Frequency-based approaches such as *adaptive stabilization* (Castro, 2004; Ghinita, 2006), *piggybacking stabilization with lookups* (Alima, 2003; Li, 2005), and *reactive stabilization* (Alima, 2003) reduce the maintenance overhead by reducing the frequency in invoking routing-state correction procedures. Adaptive stabilization adjusts the frequency based on churn rate and the importance of each routing state to lookup performance\(^2\). Systems such as DKS (Alima, 2003) and Accordion (Li, 2005) piggyback stabilization with lookups to reduce the necessity of performing dedicated periodic stabilization; DKS refers to this as *correction-on-use*. Reactive stabilization such as DKS’s *correction-on-change* (Ghodsi, 2005) does away altogether with periodic stabilization. Instead, changes to overlay networks due to membership changes are propagated immediately when membership-change events are detected. However, Rhea et. al. reported that reactive stabilization can increase maintenance overhead under high churn rate and constrained bandwidth availability (Rhea, 2004).

As an example, we discuss the stabilization mechanism in DKS (Distributed \(k\)-ary Search). DKS is proposed as a framework that generalizes different DHT implementations as a \(k\)-ary search, e.g., Chord is an instance of DKS when \(k = 2\) (Alima, 2003). Rather than periodic stabilization, DKS maintains its overlay network based on three main principles: *local atomic operations*, *correction-on-use*, and *correction-on-change*. With the local atomic operations, DKS serializes concurrent node insertions/leaves between two existing adjacent nodes. This reduces the number of incorrect successor and predecessor pointers during churn. However, the local atomic join does not correct other routing states such as fingers affected by the churn. These routing states will be corrected by correction-on-use and correction-on-change.

The correction-on-use technique piggybacks stabilization during lookup processes. If the number of lookup messages is high, then the overlay network can be maintained without a need for dedicated stabilizations. Essentially, a routing table entry is not corrected until it is used during lookups. To realize correction-on-use, every lookup message contains information about the position of the receiver from the sender’s perspective\(^3\). If the receiver determines that the information (i.e. the sender’s perspective regarding the position of the receiver) is wrong, then the receiver advises the sender about the correct information (to the best of the receiver’s knowledge). The disadvantage of correction-on-use is that the speed at which the overlay network is corrected depends on the amount of lookup traffic. To address this disadvantage, DKS also employs correction-on-change: after a new node joins, it notifies all nodes that need to be updated.
VARYING SIZE OF ROUTING TABLES

This approach reduces the size of routing tables so that the number of routing states to correct becomes smaller. Examples of DHT that implement this approach include CAN (Ratnasamy, 2001), Koorde (Kaashoek, 2003), and Accordion (Li, 2005). However, reducing the size of routing tables potentially increases lookup path length (Xu, 2003).

In Accordion (Li, 2005), the size of routing tables is controlled through the process of acquisition and eviction of routing states. The rate of state acquisition is determined by a specified bandwidth budget, while the rate of state eviction is influenced by the churn rate. During acquisition, new states are added into a routing table. Accordion couples DKS’s correction-on-use approach with explicit stabilization. The frequency of explicit stabilization is constrained by the bandwidth budget. During eviction, node removes routing entries that point to nodes perceived to be non-existent. In addition, Accordion favors routing states that points to nodes with a longer live time; pointers to relative newer nodes have a higher probability to be evicted. Thus, a higher bandwidth budget increases routing-table size, whereas a higher churn rate reduces it.

Besides reducing the size of routing tables, DHT can also partition each routing table into two parts: one part consisting of entries that are corrected through stabilization, and the other part consisting of cached entries. This reduces the maintenance overhead while achieving a shorter lookup path length. For example, in the latest implementation of Chord, a finger table consists of $O(\log N)$ fingers, and a number of location caches maintained by a LRU replacement policy (Stoica, 2001).

Hierarchical Chord

A hierarchical Chord partitions its nodes into a multi-level overlay network. Because nodes join a smaller overlay network than in a flat structure, each node maintains and corrects a smaller number of routing states than in a flat structure. Figure 4 shows an example of hierarchical Chord. In hierarchical Chord, each node is assigned a group identifier ($gid$) and a unique node identifier ($nid$). We use the notation $gid|nid$ to denote the group identifier and node identifier of each node.

Nodes with the same $gid$ form a group and groups are organized in the top-level as Chord overlay network. Within each group, nodes are organized as a second-level overlay using the node identifier. The topology and stabilization mechanism can differ from the top-level. In each group, one or more nodes designated as supernodes act as gateways to other nodes in the group. In Figure 4, node 0|5, node 2|7, node 4|2, and node 6|4 are respectively the supernodes of groups $g_0$, $g_2$, $g_4$, and $g_6$.

In hierarchical Chord, a lookup request for key $k$ implies locating the group responsible for $k$. Figure 5 illustrates the process. Firstly, a lookup request for key $k$ is routed to the supernode of the initiating group. Secondly, using Chord lookup algorithm (Chord, 2001), the lookup request is further routed to the supernode of group whose group identifier is $gid = successor(k)$. Thirdly, the lookup request can be further forwarded to one of the second-level nodes in group $k$ based on additional criteria. As shown in Figure 5, a lookup request for key 2, initiated by second-level node 6|6, is forwarded to its supernode 6|4 (step 1). In the top-level overlay, the lookup request is routed to supernode 2|7 of group 2 (step 2). Finally, supernode 2|7 can further forward the request to its second-level nodes (step 3), e.g., lookup for compute resources of type 2 in multiple administrative domains (Teo, 2005).

If new nodes join a hierarchical Chord when some routing states in the top-level overlay are incorrect, i.e., yet to be updated, then the top-level overlay may end up with two or more groups with the same
group identifier. This is called collisions of groups. In the following subsections, we discuss how collisions occur and present a collision detection and resolution scheme, and a collision avoidance scheme. To avoid sending additional overhead messages, collision detection is performed together with successor stabilization, i.e., the process of correcting successor pointers. 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.

In presenting our algorithm, we assume that each node maintains a list of variables shown in Table 1. The algorithm adopts the same convention as in (Stoica, 2001), where remote procedure calls or variables are preceded by the remote node identifier, while the local procedure calls and variables omit the local node identifier.

Figure 4. A two-level overlay network consisting of four groups

Figure 5. Example of lookup in hierarchical chord
Hierarchical Structured Peer-to-Peer Networks

Collisions of group identifiers arise because of join operations invoked by nodes. Figure 6 shows the node-join algorithm for hierarchical Chord. Node $n$, whose group identifier is denoted as $n.gid$, makes a request to join group $g$ through bootstrap node $n'$. In a hierarchical Chord, this means finding $\text{successor}(g|0)$ in the top-level overlay. If $n'$ successfully finds an existing group $g$ then $n$ joins this group using a group-specific protocol (line 5–9). However, if $n'$ returns $g' > g$, then $n$ creates a new group with identifier $g$ (line 11–15). A collision occurs if the new group is created even though a group with identifier $g$ already exists. This happens when $n$ and bootstrap node $n'$ are in two different groups, and the top-level overlay has not fully stabilized, i.e., some supernodes successor pointers are yet to be updated.

Figure 7 illustrates a collision scenario when node 1|2 and node 1|3 belonging to the same group $g_1$, join concurrently. Due to concurrent joins, $\text{find_successor}()$ invoked by both nodes returns node 2|7. As a result, both the new node joins create two groups with the same group identifier $g_1$.

Collisions increase the maintenance overhead in the top-level Chord ring by $\Omega(c)$ times. Let $K$ denotes the number of groups and $N$ denotes the number of nodes. Assuming that each group assigns one supernode, the ideal size of the top-level overlay is $K$ supernodes. Without collisions, the total number of stabilization messages is denoted as $S$. With collisions, the size of top-level overlay is increased by $c$ times, i.e., $cK$ groups. As each group performs periodic stabilization, the cost of stabilization with collisions ($S_C$) is $\Omega(cS)$. The stabilization cost ratio, with and without collisions, is shown in Equation 1.

$$\frac{S_C}{S} = \frac{cK \log^2 cK}{K \log^2 K} = \frac{c \log^2 cK}{\log^2 cK} = \Omega(c)$$

Collisions also increase the lookup path length in the top-level Chord by $O(\log c)$ hops. Without collisions, the top-level Chord ring consists of $K$ supernodes, and hence, the lookup path length is $O(\log K)$. With collisions, the size of the top-level overlay becomes $cK$ and the lookup path length is $O(\log cK) = O(\log c + \log K)$ hops.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gid$</td>
<td>$m$-bit group identifier</td>
</tr>
<tr>
<td>$nid$</td>
<td>$m$-bit node identifier</td>
</tr>
<tr>
<td>successor</td>
<td>pointer to $successor(gid)$ if $n$ is a supernode, $\text{nil}$ otherwise</td>
</tr>
<tr>
<td>predecessor</td>
<td>pointer to $predecessor(gid)$ if $n$ is a supernode, $\text{nil}$ otherwise</td>
</tr>
<tr>
<td>$is_super$</td>
<td>$\text{true}$ if $n$ is a supernode, $\text{false}$ otherwise</td>
</tr>
<tr>
<td>supernode</td>
<td>pointer to supernode of group $gid$ if node is a supernode, $\text{nil}$ otherwise</td>
</tr>
</tbody>
</table>

Table 1. Variables maintained by node $n$ in hierarchical chord
Collisions can be detected during successor stabilization. This is achieved by extending 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. Figure 8 presents a collision detection algorithm. It first ensures that the successor pointer of a node is valid (line 4–5). It then checks for a potential collision.

**COLLISION DETECTION AND RESOLUTION SCHEME**

Collisions can be detected during successor stabilization. This is achieved by extending 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. Figure 8 presents a collision detection algorithm. It first ensures that the successor pointer of a node is valid (line 4–5). It then checks for a potential collision.

**Figure 6. Join operation**

1.  //Node $n$ joins through bootstrap node $n'$
2.  $n$.join($n'$)
3.  $h'$ = suffix($n'$);
4.  $s = h'.\text{find\_successor}(\text{gid}(0));$  //See (Stoica, 2001) for the detail
5.  if (\text{gid} == s.gid)
6.    //s is a supernode of group $g$
7.    \text{join\_group}(s);  //Group-specific join protocol
8.    \text{is\_super} = \text{false};
9.    \text{supernode} = s
10.   else
11.    //n creates a new group
12.    //This can cause a collision
13.    \text{predecessor} = \text{nil};
14.    \text{successor} = s;
15.    \text{is\_super} = \text{true};

**Figure 7. Collision at the top-level overlay**
Figure 8. Collision detection algorithm

```
1. // n periodically verifies its successor pointer, 
2. // and announces itself to the successor. 
3. n.stabilize_successor() 
4. if successor.is_super == false then 
5. successor = successor.supernode(); 
6. 
7. p = successor.predecessor; 
8. if (p ≠ n) and (p.gid == gid) then 
9. if is_collision(p) then 
10. merge(p) 
11. else if n.gid < p.gid < successor.gid then 
12. successor = p; 
13. successor.notify(n); 
```

(a) Main Algorithm

```
14. // n' thinks it might be our predecessor 
15. n.notify(n') 
16. if (predecessor == nil) 
17. or (predecessor.is_super == false) 
18. or (predecessor < n' < n) 
19. then 
20. predecessor = n'; 

21. // Assume one supernode per group 
22. n.is_collision(n') 
23. if (gid == n'.gid) 
24. return true 
25. 
26. return false
```

(line 8–10), before updating the successor pointer to point to the correct node (line 11–13).

Figure 9 illustrates the collision detection process. In Figure 9(a), a collision occurs when nodes 1|2 and 1|3 belonging to the same group, group 1, join concurrently. In Figure 9(b), node 1|3 stabilizes and causes node 2|7 to set its predecessor pointer to node 1|3 (step 1). Then, the stabilization by node 0|5 causes 0|5 to set its successor pointer to node 1|3 (step 2), and node 1|3 to set its predecessor pointer
to node 0|5 (step 3). In Figure 9(c), the stabilization by node 1|2 causes 1|2 to set its successor pointer to node 1|3. At this time, a collision is detected by node 1|2 and is resolved by merging 1|2 to 1|3.

If each group contains more than one supernode, then the `is_collision` routine shown in Figure 8 may incorrectly detect collisions. Consider the example in Figure 10(a). When node \( n \) stabilizes, it incorrectly detects a collision with node \( n' \) because \( n.successor.predecessor = n' \) and \( n.gid = n'.gid \). An approach to avoid this problem is for each group to maintain a set of its supernodes (Garcés-Erice, 2003; Gupta, 2003) so that each supernode can accurately decide whether a collision has occurred. The modified collision detection algorithm is shown in Figure 10(b).

To resolve collisions, groups associated with the same `gid` are merged. After the merging, some supernodes, depending on the group policy, become ordinary nodes. Before a supernode changes its state into a second-level node, the supernode notifies its successors and predecessors to update their pointers (see Figure 11). Nodes in the second level also need to be merged to the new group. We discuss two methods to merge groups, namely `supernode initiated` and `node initiated`. 

---

**Figure 9. Collision detection piggybacks successor stabilization**

![Diagram showing collision detection process]

![Diagram showing supernode stabilization](a)

![Diagram showing node stabilization](b)

![Diagram showing supernode collision](c)

![Diagram showing supernode merge](d)
Hierarchical Structured Peer-to-Peer Networks

**Supernode Initiated**

To merge two groups \( n.gid \) and \( n'.gid \), supernode \( n \) notifies its second-level nodes to join group \( n'.gid \) (Figure 12). The advantage of this approach is that second-level nodes join a new group as soon as a collision is detected. However, \( n \) needs to keep track of its group membership. If \( n \) has only partial knowledge of group membership, some nodes in the second-level can become orphans.

**Figure 11. Announce leave to preceding and succeeding supernodes**

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>//Set predecessor of ( n ) to ( n' )</td>
</tr>
<tr>
<td>2.</td>
<td>( n.replace.predecessor(n') )</td>
</tr>
<tr>
<td>3.</td>
<td>predecessor = ( n' );</td>
</tr>
<tr>
<td>4.</td>
<td>//Set successor of ( n ) to ( n' )</td>
</tr>
<tr>
<td>5.</td>
<td>( n.replace.successor(n') )</td>
</tr>
<tr>
<td>6.</td>
<td>successor = ( n' );</td>
</tr>
</tbody>
</table>

---

Figure 10. Collision detection for groups with several supernodes

(a) Multiple Supernodes in each Group

(b) Modified is_collision Algorithm
Hierarchical Structured Peer-to-Peer Networks

Node Initiated

In node-initiated merging, each second-level node periodically checks that its known supernode \( n' \) is still a valid supernode (Figure 13). If \( n' \) is no longer a supernode, then the second-level node will ask \( n' \) to find the correct supernode. These second-level nodes then join a new group through the new supernode. This approach does not require supernodes to track group membership. However, it introduces an additional overhead to the second-level nodes as they periodically check the status of their supernode.

COLLISION AVOIDANCE SCHEME

Avoiding collision has the following advantages:

1. lower overhead: Runaway collisions are very costly, and detecting and resolving collisions is highly difficult in a decentralized and dynamic peer-to-peer system with high churn rate (Teo, 2008);
2. reduced bootstrap time: New peers can join the network at a faster rate because the time between the join event and the update of the underlying overlay network states is reduced;
3. improved lookup performance: Without collision, the top-level overlay is maintained at the ideal size;
4. faster resource availability: As costly collision resolution is not necessary, resources are available once the nodes join the network.
In the join operation in Figure 6, a node performs a lookup for the group identifier, which is handled by the supernode of the successor group. If the joining node and the supernode that respond to the lookup have the same group identifier, the node joins the second-level overlay. Collisions occur when concurrent joins create multiple new groups with the same group identifier in the first-level overlay. This scenario arises because before the routing states are updated, each joining node is unaware of the existence of other joining nodes.

To avoid collisions due to join requests, the join protocol is modified such that the predecessor node handles the join lookup request instead of the successor node. The rationale behind this change is that all join requests are serialized at the predecessor. If the group identifier of the successor’s supernode is different from the group identifier of the joining node, then the predecessor immediately changes its successor pointer to reflect the new group created by the joining node. Thus, this modification allows the overlay network to reveal new groups to subsequent joining nodes and make them available to incoming lookups.
Join Protocol

The detailed join algorithm shown in Figure 14 is divided into the following steps:

1. A joining node performs a lookup for group $gid$, which is routed at the top overlay to the supernode whose identifier is $successor(gid(0))$ (line 3).
2. If a group for the resource type exists, a supernode is already created for the resource type and the joining node becomes a member of the second-level overlay (lines 5–7).
3. If a group for the resource type does not exist, the joining node becomes the supernode of a newly created group. The joining node then sets its predecessor and successor pointers accordingly (lines 9–11). In addition, the supernode in step 1 updates its successor pointer to the joining node.
4. Stabilization is used by the new supernode to build a finger table (line 12).
Hierarchical Structured Peer-to-Peer Networks

Figure 15. Leave operation

```
1.   n.leave()
2.   g' = find_group(n);
3.   if (is_super == true) then
4.       predecessor.replace_successor(successor);
5.       successor.replace_predecessor(predecessor);
6.       for each n' ∈ group(n.id) do
7.           n.depart();
8.       n.join()
9.   else
10.   //leave second-level overlay
11.   supernode.node_leave(n);
```

Leave Protocol

When a supernode leaves its group becomes an orphan group if the supernode is the only one in the group. If a new node attempts to join the orphan group, then a collision occurs because the new node cannot locate the orphan group in the top-level overlay. Hence, a new group is created in the top-level overlay where its group identifier is the same as the orphan group. To prevent this type of collisions, the departing supernode notifies its first-level overlay successor and predecessor to update their finger tables. Furthermore, a new supernode needs to be elected for the orphan group to prevent collisions during subsequent node joins.

Figure 15 presents a simple-but-costly leave protocol that reuses our collision-free join operation (Figure 14) to elect new supernodes. In this protocol, the orphan group is disbanded where all its members are forced to rejoin the system. Thus, the node which completes its join operation first becomes the new supernode.

Failures

A more complex case which leads to collisions is when supernodes fail. A supernode failure invalidates other nodes’ successor pointers and finger table. While inaccurate finger table only degrade lookup performance, inaccurate successor pointers leads to collisions. However, avoiding collisions due to supernode failures is a challenging problem. Unlike departures (Section 3.3.2) where supernodes leave the overlay network gracefully, failures can be viewed as supernodes leave the overlay network silently. This means that there is no notification to the overlay network to indicate that any collision avoidance procedures should be triggered. Hence, it is necessary for the system to detect the presence of supernode failures so that any corrective measures can be initiated, e.g. the collision detection-and-resolution
scheme presented in Section 3.2.

**SUMMARY AND OPEN ISSUES**

Efficient lookup is an essential service in peer-to-peer applications. In structured peer-to-peer systems, dynamic joining and leaving of peers and failing of peer nodes change the structural properties of the overlay network. Stabilization, the process of overlay network maintenance is a necessary overhead and impact on the lookup performance. In this chapter, we discuss three main approaches in reducing overlay maintenance overhead, namely, hierarchical DHT, varying frequency of stabilizations and varying number of routing states. We discuss in more detail hierarchical DHT where nodes are organized as multi-level overlay networks. In hierarchical DHT, collisions of groups occur when concurrent node joins result in multiple groups with the same group identifier being created at the top-level overlay. Collisions increase the size of the top-level overlay by a factor c, which increases the lookup path length by only $O(\log c)$ hops, but increases the total number of stabilization messages by $\Omega(c)$ times. To address the collision problem, we present firstly a collision detection-and-resolution scheme and two approaches to merge collision groups, namely, supernode-initiated and node-initiated. Though the effect of collisions can be reduced by collision detection and correction, the message overhead cost is high. A collision avoidance scheme where join and leave operations are collision free is discussed.

The open issues of group collisions in hierarchical DHT include:

1. Current experimental results on both collision detection-and-resolution and avoidance schemes assume that node joins, leaves, and fails occur exclusively (March, 2005; Teo, 2008). However, in practice, these three events are interleaved and are important when network churn rate is high. Thus, in addition to the frequency of top-level overlay’s stabilizations during collision detections (March, 2005), churn also impacts how often second-level nodes should check the status of their supernode during the node-initiated collision resolution approach. An adaptive method similar to (Ghinita, 2006) is a possible direction; however, this has not been studied in detail.

2. When a supernode leaves, the current collision-free leave protocol uses a simple but naïve approach to deal with orphan groups where all the second-level nodes are forced to rejoin a hierarchical DHT. A more efficient approach is required. For example, an efficient distributed election scheme can be used to select a supernode among the second-level nodes, and only the elected supernode joins the top-level overlay.

3. Node failures are unplanned and collisions that arise due to node failure are therefore harder to address. Avoiding collisions due to supernode failures is a challenge. We envisage two possible solutions; both using multiple supernodes. Firstly, each group employs a number of backup supernodes so that the collision-free join protocol is able to resolve the problem of orphan group before redirecting new nodes to the group. Alternatively, each group can have multiple supernodes in the top-level overlay; but this is at the expense of a larger top-level overlay.
REFERENCES


Hierarchical Structured Peer-to-Peer Networks


Hierarchical Structured Peer-to-Peer Networks


Hierarchical Structured Peer-to-Peer Networks


KEY TERMS AND DEFINITIONS

**Chord**: A structured overlay network with nodes organized as a logical ring.

**Churn**: Changes in overlay networks due to dynamic node joins, leaves, or failures.

**Collision of Groups**: An occurrence when two or more groups with the same group identifier occupy the top-level overlay network.

**Distributed Hash Table**: A class of distributed systems where keys are map onto nodes and nodes are organized as a structured overlay network to support scalable lookup service.

**Finger**: An entry in each node’s routing table (finger table) in Chord

**Key-Value Pair**: A tuple consisting of a unique identifier (key) and an object (value) to be stored into DHT.

**Predecessor**: The immediate counter-clockwise neighbor of a node in Chord.

**Successor**: The immediate clockwise neighbor of a node in Chord.

**Supernode**: A gateway node to a second-level hierarchical overlay network.

**Stabilization**: A procedure to keep the routing information in each peer nodes updated.

ENDNOTES

1 Size of the top-level overlay without collision.

2 Routing states with higher importance such as successor pointers in Chord (Stoica, 2001) and leaf sets in Pastry (Rowstron, 2001), are refreshed/corrected more frequently.

3 This is possible due to the $k$-ary model.