HOURS: Achieving DoS Resilience in an Open Service Hierarchy

Hao Yang, Haiyun Luo, Yi Yang, Songwu Lu, Lixia Zhang
Computer Science Department, University of California, Los Angeles
E-mails: {hyang, hluo, yangyi, slu, lixia}@cs.ucla.edu

Abstract

Hierarchical systems have been widely used to provide scalable distributed services in the Internet. Unfortunately, such a service hierarchy is vulnerable to DoS attacks. This paper presents HOURS that achieves DoS resilience in an open service hierarchy. HOURS ensures high degree of service accessibility for each surviving node by: 1) augmenting the service hierarchy with hierarchical overlay networks with rich connectivity; 2) making the connectivity of each overlay highly unpredictable; and 3) recovering the overlay when its normal operations are disrupted. We analyze an HOURS-protected open service hierarchy, and demonstrate its high degree of resilience to even large-scale, topology-aware DoS attacks.

1. Introduction

Hierarchical systems have been widely used to provide scalable distributed services in the Internet. Such a system usually organizes the servers into a tree-like topology, and forwards the user queries along prescribed top-down paths. The hierarchy is accessible to any user over the global Internet, and its topology is publicly known. We call such systems *open service hierarchies*. Examples are DNS [16], LDAP [24], PKI [7] to name a few. Due to the popularity and importance of these services (e.g., domain name resolution, certification), it is desirable that they be highly available even under the emerging Internet Denial-of-Service (DoS) attacks[1][17][26].

Two requirements arise in order to achieve high degree of service availability. First, the server that holds the answer to a user query should be properly functioning. Second, the service should be *accessible* in that the query can be forwarded to the server holding the answer. While individual servers can be protected through replication or anycast techniques, we focus on the resilience of *service accessibility* under DoS attacks in this paper.

1 We use the term *DoS attack* in its general form, which also includes the Distributed DoS (DDoS) attacks.

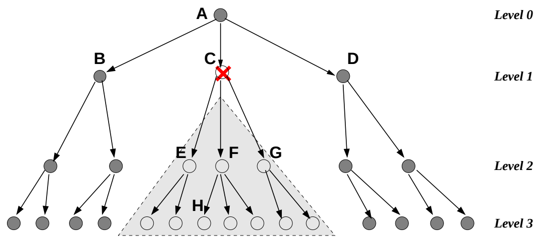

Figure 1. **DoS attacks against a single node may throttle the accessibility of the entire subtree.**

The service accessibility is vulnerable to DoS attacks in an open service hierarchy due to its low connectivity. In the example shown in Figure 1, the DoS attacks against any single node (e.g., C) may create *domino* effects and throttle the accessibility of all its descendants (e.g., E, F, G and H). Specifically, due to the failure of the high-level node C, user queries cannot reach node H that holds the answer, even if node H itself is well-protected and functioning properly. Since the hierarchy topology is publicly known, the attacker can selectively attack the *weakest* node/link along the top-down path to deny the service of the target node.

The goal of this work is to ensure high degree of service accessibility for each *surviving* node, regardless of the failures of other nodes that are directly under DoS attacks. In other words, we seek to confine the damage of DoS attacks in an open service hierarchy. This problem is challenging due to two reasons. First, the attacker can exploit the hierarchy topology information to launch topology-aware, and potentially large-scale DoS attacks. Second, the open nature of the service prohibits existing DoS solutions that are based on user authentication, such as SOS [11].

Our approach is to exploit Hierarchical Overlays Using Randomized Structure (HOURS) and establish *rich yet unpredictable* connectivity in the hierarchy. The base design of HOURS explores two ideas: *hierarchical overlays* and *randomized overlays*, to achieve DoS resilience. In this design, each node guides its children to form an overlay network. Within an overlay, each node keeps a few *random* pointers to other sibling nodes as well as their children. This way, HOURS augments the service hierarchy with hierarchical overlays and enriches the connectivity. When certain nodes are under DoS attack, the queries are forwarded across the overlays to bypass the attacked nodes.

The above base design is highly resilient to random attacks in which the attacker randomly selects victim nodes. However, it cannot address topology-aware and large-scale attacks, in which the attacker attacks large numbers of adjacent nodes in an overlay. The enhanced design of HOURS defeats such attacks by a novel forwarding mechanism and an active recovery mechanism.

We analyze the DoS resilience of an HOURS-protected open service hierarchy in terms of both service accessibility and forwarding efficiency. The results show that HOURS is highly resilient and achieves graceful performance degradation when the scale of the DoS attacks increases. These results are also verified by extensive simulations.

HOURS preserves the original service hierarchy and is backward-compatible with the current system implementation, as opposed to completely replacing the hierarchy with a flat overlay [4]. HOURS is also incrementally deployable. Any part of the hierarchy can immediately benefit from the deployment of HOURS with significantly improved (local) connectivity and enhanced resilience against DoS attacks.

The rest of the paper is organized as follows. Section 2 defines our network model. Section 3 and Section 4 present the base design and the enhanced design of HOURS, respectively. Section 5 analyzes the system resilience against DoS attacks, and Section 6 presents simulation evaluations. Section 7 discusses several design issues. Section 8 reviews the related work. Finally, Section 9 concludes this paper.

2. Models

We consider an *open service hierarchy*, a hierarchical system that provides large-scale lookup service in the Internet such as DNS [16], LDAP [24], PKI [7]. While it is hard to model all implementation details of various systems, we capture their essential features in a simplified model, on which our design and analysis are based. In this model, an open service hierarchy is characterized as follows:

- *Topology*: It consists of a large number of nodes that are organized into a hierarchical tree structure.
- *Naming*: There is a unified naming space. Each node manages a unique portion of the space, and may delegate a subset of its portion to its children nodes.
- *Usage*: The users access the lookup service via queries, which are forwarded top-down over the hierarchy to the nodes that hold the answer/data.
- *Openness*: The service is open to any user over the Internet, and the hierarchy topology is publicly known [2].

For ease of presentation we assume that the service hierarchy exhibits an exact tree structure, and each node corresponds to one physical server. In Section 7 we will see that our design can be extended to accommodate replicated servers and more complicated topologies such as a mesh. We assume that the tree may have an arbitrary number of levels, and there may be an arbitrary number of nodes in each level. Each node in the tree may join, leave, fail at any time despite infrequently.

Lastly we define two terms as follows.

- *Sibling*: Two nodes in a tree are siblings if and only if they share the same parent node.
- *Nephew*: A nephew node is a child of a sibling node.

3. Base Design of HOURS

The base design of HOURS explores two ideas, hierarchical overlays and randomized overlays, to establish rich yet unpredictable connectivity in the hierarchy.

3.1. Hierarchical Overlays

On top of the original service hierarchy, HOURS organizes nodes into multiple overlay networks, as shown in Figure 2. Each node participates in one overlay only, formed together with all its siblings. Within an overlay, each node maintains a routing table that has: 1) a few sibling pointers; and 2) q nephew pointers to the children of the neighboring sibling. For example, in the level-1 overlay shown in Figure 2, node D keeps three sibling pointers to node C, I and J, respectively. D also keeps one nephew pointer to G, a child of C that is D's neighbor (q=1 in this case). We will describe how each node picks up these pointers shortly.

The sibling and nephew pointers in the hierarchical overlays enrich the connectivity among nodes both horizontally (i.e., in the same level) and vertically (i.e., across adjacent levels). When the top-down forwarding fails due to DoS attacks on the prescribed path, such overlay pointers can be used to forward the query and bypass the attacked nodes, as described in Section 3.3.

HOURS preserves yet augments the original hierarchy, thus achieving several desirable properties. First, without overhauling the infrastructure base, HOURS is compatible with the current system implementation and incrementally deployable. Second, HOURS preserves the delegated management and allows for each parent to enforce proper admission control. This is important in preventing attackers from joining the hierarchy and launching DoS attacks from *inside* the system, as we will further elaborate in Section 5.

2 Although a normal user does not maintain such topology information due to overhead concern, the attackers may intentionally collect this information to launch topology-aware attacks.

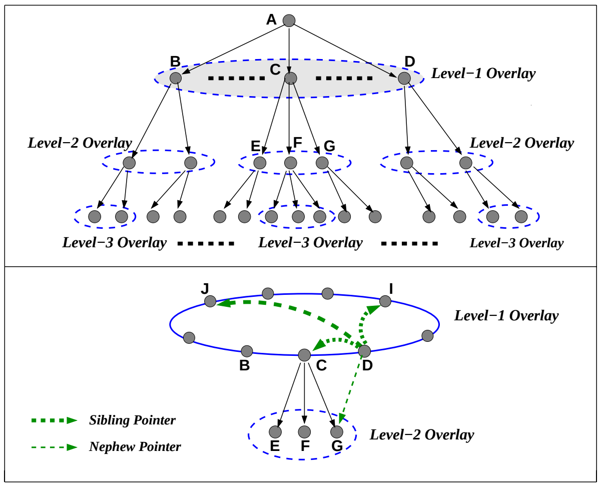

Figure 2. The base design of HOURS. *Upper*: HOURS augments the original hierarchy with hierarchical overlay networks. *Bottom*: One example of the overlays, in which each node (e.g., D) keeps a few random pointers to its siblings and nephews.

3.2. Randomized Overlays

The construction of each overlay is inspired by a small-world paradigm [12]. Each node, with the assistance from its parent node, generates its routing table (i.e., sibling and nephew pointers) using a *randomized* algorithm shown in Algorithm 1.

Each node is assigned an identifier (*ID*) that is randomly chosen from a circular identifier space, e.g., by applying a hash function such as SHA-1 on its name. For the purpose of probability calculation during routing table generation, each node is also given an *index* by its parent. The parent node sorts the IDs of all its children, assigns index 0 to an arbitrary node, and traverses the identifier circle clockwise while incrementing the index one by one. We use node i to refer to the node with index i. Note that the ID of a node is determined by its name and the publicly known hash function, while the node index depends on the complete membership information of an overlay.

The sibling pointers are generated based on the node indices. Consider two arbitrary nodes, say i and j, in an overlay of N nodes. The index distance from node i to node j is defined as $d_x(i,j)$=$(j-i) \mod N$, i.e., clockwise difference between their indices. Accordingly, node i keeps a pointer to node j with a probability of $\frac{1}{d_x(i,j)}$. This way, each node keeps a pointer to its clockwise sibling for sure, as well as a few random shortcuts to other siblings. Finally, each node contacts its clockwise sibling, and establishes q nephew pointers to q random children of that sibling.

The above operations require a node to know: 1) the size of the overlay, N; and 2) the addresses of the nodes to which it decides to generate pointers. Note that these two pieces of information are *readily available* at the parent node that manages and assigns indices to all its children. In essence, the routing table generation in HOURS is fully distributed in that each node builds its routing table independently, but under the centralized guidance of its parent node [3].

Algorithm 1 Pseudo code for routing table generation

```
1: Obtain the overlay size $N$ and its own index $i$ from the parent
2: $U = \phi$
3: for $m = 1$ to $N-1$ do
4:     With probability of $\frac{1}{(m-i) \mod N}$, $U = U + \{m\}$
5: end for
6: Query the parent about the addresses of all nodes with indices in $U$
7: Create routing table with the returned sibling pointers
```

3.3. Query Forwarding

In HOURS, a query is forwarded along the top-down tree path whenever possible, a process called *hierarchical forwarding*. When hierarchical forwarding fails at an intermediate node due to DoS attacks, the query is forwarded across overlays using sibling and nephew pointers as a detour, a process called *overlay forwarding*. After the attacked node(s) are bypassed, the query will be routed back to the top-down path, from which the hierarchical forwarding resumes. Thus the entire query forwarding may be a mixture of hierarchical forwarding and overlay forwarding.

We use $[v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_l]$ to denote the top-down tree path for a query, where v_i denotes a level-i intermediate node in the service hierarchy, and v_l denotes the destination node that holds the answer/data. Upon the failure of an intermediate node v_{i-1}, overlay forwarding is activated and the forwarding path becomes

$$[\cdots v_{i-2} \rightarrow S_{i-1} \rightarrow S_i(v_i) \rightarrow v_{i+1} \cdots]$$

where S_i denotes the overlay forwarding path inside the overlay that v_i belongs to. S_i starts with an entrance node e_i and ends at the exit node x_i. For a slight abuse of notations, S_i may also denote a level-i overlay network, and its meaning will be clear from the context.

The query typically starts with the root node v_0. When v_0 is out of service due to DoS attacks, query forwarding can start with any node in the l overlay networks along the top-down tree path, i.e., $\{S_1, S_2, \cdots, S_l\}$. We will discuss such bootstrapping issues in Section 7.

Once a query enters an overlay at the entrance node e_i, it is forwarded toward v_i, even if v_i may be down. Each intermediate node uses its sibling pointers to forward the query

3 As a result, technically a new node cannot join the overlay when its parent fails. However, this may not be a problem in practice since the management of the service hierarchy requires a new node to register itself with and be admitted by the parent in the first place.

Algorithm 2 Pseudo code for query forwarding

```
1: if (query destination is a descendant node) then
2:    $V \leftarrow$ the child along the top-down tree path
3:    if ($V$ is alive) then
4:       Forward the query to $V$
5:    else
6:       Forward the query to an alive child
7:    end if
8: else
9:    if (OD-node is not the clockwise neighbor in the overlay) then
10:      Forward the query using the best sibling pointer
11:   else
12:      Forward the query using the best nephew pointer
13:   end if
14: end if
```

in a greedy manner (Line 10 in Algorithm 2), until the query hits an exit node. Specifically, it forwards the query to the sibling in its routing table that is closest to node v_i in the *ID* space. For this reason we also call v_i the *overlay-destination node*, or **OD-node** in short.

The exit node can be either the OD-node v_i itself if it is alive, or v_i's counter-clockwise neighbor if v_i is out of service. If the query reaches the OD-node v_i, hierarchical forwarding resumes[4], i.e., $[\cdots S_i(v_i) \rightarrow v_{i+1} \cdots]$. Otherwise, the query reaches v_i's counter-clockwise neighbor, and then is forwarded to the next level (i+1) overlay using a nephew pointer, i.e., $[\cdots S_i \rightarrow S_{i+1} \cdots]$. To speed up overlay forwarding, the query is forwarded to the nephew that is closest, in the *ID* space, to the next level OD-node v_{i+1} (Line 12 in Algorithm 2).

3.4. Effectiveness of the Base Design

The performance of the randomized overlay, in terms of both routing table size and number of overlay forwarding hops, is shown in Theorem 1 (proof in technical report [23]).

Theorem 1 *With high probability, each node keeps a routing table with $O(\log N)$ entries, and each query is forwarded in $O(\log N)$ steps.*

As analyzed in Section 5, the base design is highly resilient to random attacks in which the attacker randomly selects victim nodes. However, using the original hierarchy topology, the attacker can easily infer the membership and thus neighboring relationship in each overlay. He can then precisely locate the OD-node v_i and its counter-clockwise neighbor, and shut them down simultaneously to break both the top-down tree path and the overlay path. We defeat such *topology-aware* attacks in the enhanced design.

4 Unless the immediate next-hop v_{i+1} fails too, in which case the query is sent to v_{i+1}'s counter-clockwise neighbor in the next level overlay, i.e., $[\cdots S_i(v_i) \rightarrow S_{i+1} \cdots]$, and overlay forwarding continues.

4. Enhanced Design of HOURS

In this section we present three mechanisms to enhance the base design and improve its resilience against large-scale *neighbor attacks*, in which a topology-aware attacker attacks a large number of neighboring nodes in an overlay. Before we describe the details, we outline the differences between base design and enhanced design as follows.

	Base Design	*Enhanced Design*
Sibling Pointer	*O(log N)*	*O(k log N)*
Nephew Pointer	*q*	*O(q k log N)*
Clockwise Neighbor	*1*	*k*
Counter−clockwise Neighbor	*0*	*1*
Overlay Forwarding	Clockwise	Clockwise or Counter−clockwise
Active Recovery	No	Yes

4.1. Increasing the Redundancy

The vulnerability of the base design to neighbor attacks comes from the fact that the pointers to a node v_i's children are maintained only at v_i's counter-clockwise neighbor. Below we progressively present three steps to increase the connectivity and address such vulnerability.

Our first step is to increase the redundancy of nephew pointers by a factor of k. That is, the k counter-clockwise neighbors of v_i will each maintain q nephew pointers to its children[5]. Thus the attacker has to shut down all k counter-clockwise neighbors of an OD-node v_i to disrupt the overlay forwarding. While this increases the difficulty to launch attacks, it provides only limited defense (k nodes).

Inspired by the random sibling pointers, our second step is to make nephew pointers *randomized* as well. For ease of implementation we let the nodes store q nephew pointers for each sibling in their routing tables. That is, when a node has established its random sibling pointers, it contacts these siblings and store q nephew pointers from each of them.

Our final step is to increase the redundancy of sibling pointers by a factor of k. Note that these three steps can actually be implemented together by letting a node i keep a pointer to its sibling node j with a probability of $\min(1, \frac{k}{d_x(i,j)})$. This way, each node keeps k pointers to its k clockwise neighbors for sure, and the number of sibling pointers increases by k times on average, compared to the

5 In another word, each node maintains q nephew pointers for each of its k clockwise neighbors.

base design case. Moreover, q nephew pointers are maintained for each sibling pointer in the routing table. As shown later, these redundant connectivity can greatly improve the resilience of the service hierarchy against topology-aware and large-scale DoS attacks.

4.2. Backward Query Forwarding

If any of the k counter-clockwise neighbors of an OD-node v_i is alive, it can serve as the exit node because it has q nephew pointers to v_i's children. However, if all these k nodes fail due to large-scale neighbor attacks, the query will stop at the node with index i-k-1 $(\text{mod } N_i)$. We use a backward forwarding mechanism to route the query backward step-by-step, until it hits a node that has nephew pointers to v_i's children. The complete overlay forwarding in the enhanced design is shown in Algorithm 3. Note that in order to facilitate backward forwarding, each node also maintains a pointer to its counter-clockwise neighbor.

The following theorem and corollary quantify the performance of the backward query forwarding (proof in technical report [23]).

Theorem 2 *For an arbitrary node with an index of i and an arbitrary distance of d, with high probability, there exists a node in the interval $[i-2d, i-d]$ that maintains nephew pointers to node i's children.*

Corollary 1 *With high probability, a query travels at most k steps backward before it hits an exit node.*

Algorithm 3 Pseudo code for enhanced overlay forwarding

```
Require: {OD-node, Mode(either forward or backward)}
 1: if (OD-node is in the routing table) then
 2:     //forward the query to OD-node or its children
 3:     Locate the corresponding entry
 4:     next_hop ← the sibling pointer in the entry
 5:     if (next_hop fails) then
 6:         next_hop ← a nephew pointer in the entry
 7:     end if
 8: else
 9:     //forward the query in the overlay
10:     if (Mode=forward) then
11:         next_hop ← the sibling pointer closest to OD-node
12:         if (itself is closer to OD-node than next_hop) then
13:             //change the mode when greedy forwarding fails
14:             Set Mode to backward in the query
15:             next_hop ← the counter-clockwise closest neighbor
16:         end if
17:     else
18:         next_hop ← the counter-clockwise closest neighbor
19:     end if
20: end if
21: Forward the query to next_hop
```

4.3. Active Recovery

The clockwise greedy forwarding requires each node to track its alive clockwise neighbor. The backward forwarding also requires a node to track its alive counter-clockwise neighbor. Conventional neighborhood recovery techniques [22][20] can maintain these pointers under limited consecutive node failures, provided that there is no gap in the ring. However, such a gap is possible under large-scale neighbor attacks. Thus we present a mechanism, called *active recovery*, to repair the neighbor pointers from a potential gap.

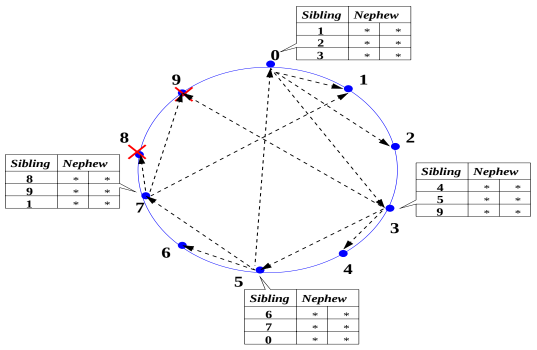

Figure 3. **Active recovery of the overlay**

Specifically, each node periodically probes its counter-clockwise neighbor. When that neighbor fails, it waits for another alive counter-clockwise neighbor (up to k counter-clockwise neighbors maintain pointers to it) to contact it and recover the neighbor pointer. If it has not been contacted after one probing period, it infers that massive node failure happens and starts the active recovery process. The node sends out a *Repair* message destined to itself. For example, if all nodes with index $[s-f, s]$ fail where $f \geq k$, it is node $s+1$ that sends out the *Repair* message. When a node receives this *Repair* message, it checks its routing table. If node $s+1$ is not in the routing table, it treats the message as a normal query and forwards it using the greedy algorithm. Otherwise, it forwards the message using the *second* best choice. Whenever a node cannot forward the *Repair* message based on the above rules, it creates a new routing entry for node $s+1$. We can see that node $s-f-1$ eventually receives this message, and adds node $s+1$ into its routing table. This way, all alive nodes still maintain a ring structure without any gap in the overlay.

We illustrate the active recovery mechanism using an example ($k=2$) shown in Figure 3. When node 8 and node 9 fail at the same time, a gap appearing between node 7 and node 0 breaks the connectivity of the overlay. Node 0 notices the failure of node 9 after one probe period, and sends a *Repair* message, destined to itself (node 0), to node 3 after one probing period. Node 3 treats this message as a normal query and forwards it to node 5. Because node 5 already keeps node 0 in its routing table, it forwards the message using the second best choice in its table: node 7. As node 7 receives the repair message, it creates an entry for node 0 since both forwarding rules do not apply. Finally node 0 fills in the sibling section with the pointer in the *Repair* message. We can see that the gap between node 7 and node 0 is now bridged.

5. Resilience Analysis

In this section, we present a formal analysis on the DoS resilience of an HOURS-protected open service hierarchy. We focus on the query forwarding performance under DoS attacks. Our analysis seeks to answer two questions: whether a query can be forwarded to its destination, and in how many hops. To this end, we use two metrics to quantify the DoS resilience. The *delivery ratio* is used to evaluate the service accessibility, and the *number of forwarding hops* is used to evaluate the forwarding efficiency. Specifically, given a node v, the delivery ratio is defined as the probability that v receives a query for which it holds the answer. The number of forwarding hops is defined as the total number of hops that a query traverses before it reaches the destination.

We first consider DoS attacks launched by attackers outside the hierarchy. We assume that the attackers can completely shut down a certain number of nodes. We also assume that the attackers are aware of the complete topology of the service hierarchy, and the members of each overlay. Since the hash function that maps the name of a node into its *ID* is well-known. the attackers can easily infer the position of the nodes in the identifier circle, and hence the neighboring relationship, in each overlay. However, we assume that the attackers cannot exactly infer the random sibling pointers kept by each node[6]. We also analyze the impact of attackers that are inside the system in Section 5.3.

We use the same notation as in Section 3.3. In addition, the number of nodes in a level-i overlay S_i is denoted as N_i, that is, $N_i = |S_i|$. We use S_{ai} $(\subseteq S_i)$ to denote the set of nodes that are attacked in overlay S_i. We set $N_{ai} = |S_{ai}|$, and $\alpha_i = \frac{N_{ai}}{N_i} \leq 1$ as the attack density.

5.1. DoS Attacks on Hierarchical Forwarding

In order to deny the service provided by a node v_l, the attackers have to first tear down the hierarchical forwarding path, i.e., $[v_0 \rightarrow v_1 \cdots \rightarrow v_l]$. Without HOURS, DoS attacks on any single node along the hierarchical forwarding path results in *zero* delivery ratio. With HOURS, even if all intermediate nodes are attacked simultaneously, the delivery ratio is still 100% since queries can always arrive at v_l using overlay forwarding.

However, overlay forwarding takes longer paths than hierarchical forwarding. Since query forwarding in HOURS is a mixture of hierarchical forwarding and overlay forwarding, the number of forwarding hops in the general case is:

$$\sum_{i=1}^{l} F(i), \text{ where} \begin{cases} F(i) \sim O(\log N_i) & \text{if } v_i \text{ or } v_{i-1} \text{ fails} \\ F(i) = 1 & \text{otherwise} \end{cases}$$

5.2. DoS Attacks on Overlay Forwarding

Since attacking the hierarchical forwarding path by itself is not effective against an HOURS-protected hierarchy, the attackers have to attack the overlay forwarding simultaneously. Overlay forwarding is composed of two phases. One is *intra-overlay forwarding*, i.e., the path from the entrance node e_i to the exit node x_i inside an overlay S_i. The other is *inter-overlay forwarding*, i.e., the single hop from the exit node x_i to any node in the next-level overlay S_{i+1}.

When inter-overlay forwarding is under attack, because the exit node x_i maintains at least q nephew pointers to q nodes in S_{i+1}, the inter-overlay forwarding fails with probability α_{i+1}^q. A reasonably large q, say 10, can make the failure probability of inter-overlay forwarding negligible. Thus the overall delivery ratio can be approximated as $\prod_{i=1}^{l} P_i$, where P_i denotes the probability that the intra-overlay forwarding succeeds in S_i. Below we study the scenarios when intra-overlay forwarding is under attack.

Given an attack density α_i, the attackers select $N_{ai} = \alpha_i N_i$ victims in order to maximize the damage of the attacks, i.e., minimizing P_i. Since the attackers do not know the random connectivity of overlay S_i, they may simply attack N_{ai} randomly-chosen nodes, namely *random attack*. On the other hand, the probability that a node v serves as an exit node for v_i decreases monotonically as their distance $d_x(v, v_i)$, i.e., the clockwise difference of their indexes, increases. The attackers can focus their attack on these counter-clockwise neighbors of v_i to maximize the probability that all potential exit nodes for v_i are shut down, namely *neighbor attack*[7]. In fact, neighbor attack is the optimal strategy for the DoS attackers, given a fixed amount of power characterized by the attack density α_i.

Under either attack, a query is forwarded according to the greedy algorithm to the alive counter-clockwise neighbor of v_i, denoted by u_i. If u_i is not an exit node, i.e., it does not have the sibling pointer to v_i or nephew pointers to S_{i+1}[8], the query will then be forwarded step-by-step in the counter-clockwise direction until it reaches an exit node.

6 Although a parent node guides its children in forming their overlay, it does not record the connectivity in the overlay.

7 Note that attacking v_i's clockwise neighbors does not affect queries that are forwarded toward v_i.

8 This means the neighboring relationship between u_i and v_i has not been established. It happens in the duration of the attacks when the system has not been recovered from node failures yet.

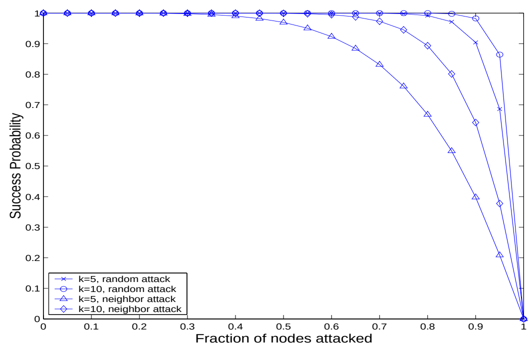

Figure 4. Large-scale attacks on overlay forwarding

Therefore, under random attack, we have

$$P_i = 1 - \alpha_i^k \prod_{j=k+1}^{N_i-1} (1 - \frac{k}{j} + \frac{k\alpha_i}{j}) \tag{1}$$

Under neighbor attack, we have

$$P_i = 1 - \prod_{j=\alpha_i N_i+1}^{N_i-1} (1 - min(1, \frac{k}{j})) \tag{2}$$

Figure 4 plots the relationship between P_i and the attack density α_i in an overlay of $N_i = 200$ nodes, under both random and neighbor attacks with different numbers of redundant neighbor pointers (k). The random attack has almost negligible impact on the service accessibility until more than 80% of the nodes are attacked simultaneously. The optimal neighbor attack causes more damage than the random attack, but the attackers still need to shut down more than 80% of the nodes to halve the service accessibility when $k = 5$. If we increase k to 10, even though 90% nodes are under attack, we can still achieve a delivery ratio as high as 64%.

The figure shows that HOURS can provide high degree of resilience even in the presence of *large-scale* (high α_i), *topology-aware* (neighbor attack) DoS attacks. More importantly, the service accessibility degrades *gracefully* as more and more nodes are out of service due to attacks or failures. This property clearly differentiates HOURS from other structured overlay designs such as Chord [22], in which the overlay connectivity can be easily inferred once its membership is known. With Chord, the service availability will be throttled down from 100% to zero, once the attackers precisely identify and simultaneously shut down the $O(\log N_i)$ nodes that maintain pointers to v_i. Similar conclusion can be drawn for other structured overlay designs such as CAN [19], Pastry [20] and Viceroy [13].

The following theorems (proof in technical report [23]) also characterize the DoS resilience of HOURS in terms of the number of forwarding hops, under random and neighbor attacks respectively. Again, the forwarding efficiency degrades gracefully as the attacker's power (α_i) increases.

Theorem 3 *Let $F(i)$ be the number of overlay forwarding hops in overlay S_i. Under random attack, $F(i) = O(\frac{1}{1-log(1-\alpha_i)} \log N_i)$.*

Theorem 4 *$F(i)$ is defined same as above. Under neighbor attack, $F(i) = O(\log N_i) + O(N_{ai})$.*

5.3. DoS Attacks from Inside the Hierarchy

The attackers who can compromise and control a certain number of admitted nodes inside the service hierarchy can cause more damage than merely shutting down nodes. For example, a compromised node may drop all queries routed through itself, leading to zero service accessibility throughout its subtree. Such damage of a compromised nodes on its descendants is out of the scope of HOURS.

However, HOURS ensures that a compromised node cannot poison the routing tables of other nodes. Thus the only way that a compromised node may damage nodes outside its subtree is to mis-route or drop their queries. The attackers may intentionally introduce routing loops in the overlay, the consequence of which is equivalent to a DoS attack on the nodes involved in the loops and thus follows our analysis above. For query dropping along overlay forwarding path, the following theorem (proof in technical report [23]) shows that the damage is determined by the distance from the compromised node to the targeted victim.

Theorem 5 *A compromised node can decrease the service accessibility of a victim sibling and its descendants by $\frac{1}{d+1}$, where d is the index distance from the compromised node to the victim sibling.*

Besides compromising admitted nodes in the system, the attacker may join the hierarchy as a normal node does. However, the random hash function (e.g., SHA-1) used to map a node's name to its ID ensures that the attacker cannot arbitrarily choose its *ID*. Thus the damage is still limited by Theorem 5. The attacker may accumulate a large number of IDs to increase their chance, known as Sybil [5] attacks. However, when each parent node enforces proper admission control, such attacks can be effectively limited in the first place, as one of the motivations for HOURS to preserve the service hierarchy.

6. Simulation Evaluation

This section evaluates both base design (Section 3) and enhanced design (Section 4) of HOURS using simulations.

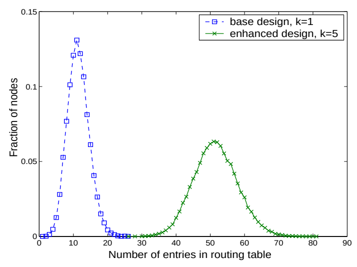

Figure 5. **Routing table size**

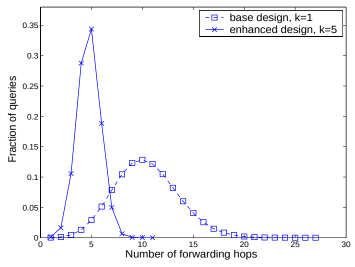

Figure 6. **Forwarding path length**

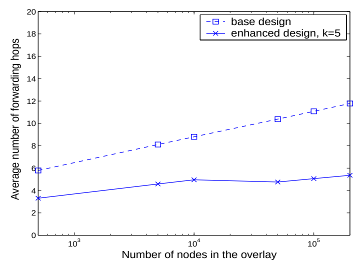

Figure 7. **Scalability**

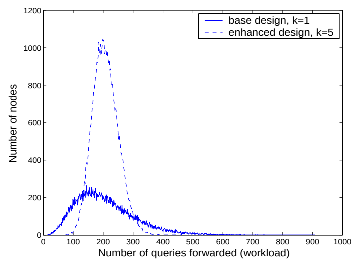

Figure 8. **Workload distribution**

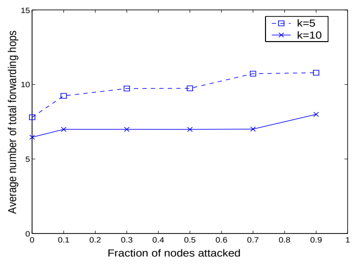

Figure 9. **Impact of random attacks**

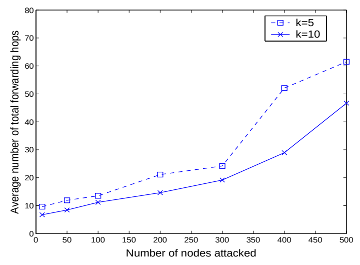

Figure 10. **Impact of neighbor attacks**

6.1. Scalable and Efficient Overlay Forwarding

We first evaluate the performance of a single randomized overlay in terms of *routing table size*, *forwarding path length*, *scalability to large overlay size*, and *load-balancing*. In these simulations, we consider a randomized overlay formed by N nodes, each of which is given a name randomly selected from a large name space.

Figure 5 plots the distribution of routing table size when N=50,000. The unit is one entry, corresponding to 1 pointers in base design, and $q+1$ pointers in enhanced design, respectively. We can see that in the base design case, a routing table has only 13.5 entries on average, which is consistent with our analysis of Theorem 1. In the enhanced design case with $k=5$, the average routing table size increases by 5 times yet still follows a similar distribution.

The distribution of forwarding path length is shown in Figure 6 (N=50,000). In each simulation run, we feed 1 million queries with random chosen source and destination nodes into the overlay. The figure demonstrates the efficiency of overlay forwarding. With the base design, on average a query is forwarded 10.4 hops before it hits the destination. In the enhanced design case, due to the improved connectivity, the average forwarding path length drops to 4.8, and 90% of the queries are forwarded in less than 7 hops.

We show the scalability of the randomized overlay in Figure 7 by varying the overlay size from 500 to $2,000,000$. The Y-axis is the average path length. As expected, in the base design case, the path length increases logarithmically with the overlay size. In fact, it approximates $\ln N$. The enhanced design shows even better performance in that the forwarding path length increases sub-logarithmically as the overlay size grows.

Lastly, we study the load-balancing feature of the randomized overlay. We define the workload of a node as the number of queries that it forwards in a simulation run. The simulation results (N=50,000) are shown in Figure 8, in which X-axis is the workload, and Y-axis is the number of nodes that have undertaken a corresponding amount of workload. We can see that in the base design case, a few nodes may have received unfair share of the forwarding workload. Due to the randomized nature of the overlay, some nodes may appear in more routing tables than the other nodes, i.e., having more inbound links, and thus have larger chances to be selected to forward the queries. The load-balancing feature is greatly improved in the enhanced design case due to the enriched connectivity. As each node keeps a larger routing table, it has more choices in selecting the next hop, thus alleviating the impact of varying inbound degrees.

6.2. DoS Resilience in an Open Hierarchy

Now we evaluate the DoS resilience of an HOURS-protected open service hierarchy using two metrics: *delivery ratio* and *number of forwarding hops*, as defined in Section 5. We build a four-level hierarchy with 1000 nodes at level 1. The attacker has special interest in throttling the service provided by all descendants of a specific level-1 node, say node T. Node T has $50,000$ children at level 2, each of which may also have several children at level 3. It is almost impossible for the attacker to attack all of them at the same time. Thus we simulate a DoS attack against node T and its siblings. We arbitrarily select a level-3 descendant of node T, say node D, and focus our evaluation on the service accessibility of node D.

We use the enhanced design to construct and maintain the overlay structure. The simulator also implements two specific strategies of the attacker, namely *random* and *neighbor* attacks, as described in Section 5.2. To study the impact of large-scale DoS attack, we vary the attack density, defined as the number of failed nodes over the total number of nodes. In each simulation run, we feed 1 million queries into the hierarchy with node D as the destination and collect the performance metrics.

The average number of forwarding hops under random attacks is shown in Figure 9. We do not show delivery ratio because it is always 100% in all simulated cases. We can see that HOURS offers high degree of resilience against random attacks. When node T is attacked but none of its siblings is, with $k = 5$ a query is forwarded in 7.8 hops on average. When 70% of node T's siblings are under attacks, the average number of forwarding hops only increases to 10.7. This number even drops to 7 when we increase the overlay connectivity with $k = 10$.

We plot the average number of forwarding hops under neighbor attacks in Figure 10. Again delivery ratio is omitted as it is always 100% in simulated cases. Consistent with our previous analysis, the neighbor attacks cause more damage than the random attacks. When the attacker can attack 300 neighbors of node T, the average forwarding path takes 24.2 hops when $k = 5$, and 19.1 hops when $k = 10$. In the extreme case when the attacker attacks 500 neighbors of node T, i.e., half of the nodes in this overlay, the average length of forwarding paths becomes 61.4 hops when $k = 5$, and 46.6 hops when $k = 10$. With no surprise, the majority of these hops are spent on counter-clockwise step-by-step forwarding to find a nephew pointer. The results here are quite conservative as we simulate an attacker with very strong power. If the attacker has only limited resources to attack 100 nodes, the average number of forwarding hops is only 13.5 when $k = 5$, and 11.2 when $k = 10$.

In summary, the above simulation results confirm that HOURS can achieve high degree of resilience even in the presence of large-scale, topology-aware DoS attacks.

7. Discussion

In this section we comment on several design issues.

Query Bootstrapping and Caching A query has to enter the hierarchy in the first place, so that HOURS can forward it properly. In order to bootstrap her queries, a client may cache the root node or a few frequently visited level-1 nodes. In case all of them are out of service, the client may still be able to bootstrap the queries by exploiting the cache. Because any node in the overlays along the top-down tree path may serve as the starting point, a query can be bootstrapped whenever such a node has been cached.

Overlay Maintenance Nodes may join, leave, or fail at any time in the service hierarchy. To handle such dynamics, each node periodically re-generates its routing table using Algorithm 1. Since the dynamics in a typical service hierarchy is at most moderate, the update period can be set reasonably large, say half a day. Between consequent updates, the routing states may deviate from the ideal distribution. The DoS attacks against a parent node may even defer the routing table update of its children. However, HOURS can achieve graceful performance degradation even in such abnormal cases, as analyzed in Theorem 3.

Server Replication HOURS works in concert with replicated servers. A pointer to a node that is replicated at multiple servers actually stores the addresses of all these servers. When a query is forwarded using this pointer, it is actually forwarded to any server that is alive. Clearly, server replication can greatly strengthen the system resilience under DoS attacks.

Hierarchy with Mesh Topology Although we present the HOURS design in the context of a tree hierarchy, it is also applicable to service hierarchy with more complex topology such as mesh. HOURS does not prohibit a node with multiple parent nodes from joining multiple overlays. In fact, the mesh topology further increases the connectivity among peering overlays, thus the DoS resilience.

Unbalanced Hierarchy In reality, the topology of a service hierarchy may be highly unbalanced (e.g., DNS [16]). Due to its scalable design, HOURS works well with the hierarchy portion that forms large overlays. In fact, the larger the overlay size, the higher degree the DoS resilience provided by HOURS. However, in a small-sized overlay (e.g., with tens of nodes), the achievable DoS resilience is limited. One possible approach is to aggregate multiple small-size overlays into a large one. But the resulting architecture may deviate from the original service hierarchy. We plan to study this issue in the future.

8. Related Work

DoS attacks have recently attracted intensive attention, and various solutions have been proposed to detect and prevent DoS attacks at network routers [17][26][9][18][21][8]. An overlay-based solution, SOS [11], does not involve network routers, but protects only sites that are accessible to authenticated users. HOURS targets an open service hierarchy that is accessible to arbitrary users, and does not assume any support from the network routers.

Caching is a well-known technique that may alleviate the damage of DoS attacks to certain extent. However, caching provides only an opportunistic query resolution, and its effectiveness highly depends on the query patterns [2][10]. On the contrary, HOURS assures to forward arbitrary queries with high probability.

The idea of using redundant connectivity to improve the resilience of a hierarchy was also adopted in the context of multicast [1]. However, the service hierarchy of our interest has a *unicast* communication model that results in a very different design. The redundant links in [1] are randomly built between any two nodes and used to efficiently flood the multicast tree. In contrast, HOURS establishes hierarchical overlays, and explores the sibling and nephew pointers to forward a query to a single destination node.

A number of overlay techniques [22][19][20][13] have been proposed in the context of peer-to-peer (P2P) networking. See [15] for a general framework of such DHT-based designs. HOURS overlay differs from existing P2P designs in two aspects. First, most P2P designs to date focus on network performance (scalability, efficiency, latency, etc.), while DoS resilience is our primary design goal. Second, the P2P designs are fully distributed and sophisticated. In contrast, HOURS takes advantage of the readily available centralized membership information of each overlay to simplify its design.

The Symphony P2P protocol [14] is probably the most relevant work to HOURS. The base design of our randomized overlay is similar to Symphony, as both are inspired by the small world paradigm [12]. However, we propose several mechanisms in the enhanced design to improve its DoS resilience. Together with the hierarchical overlay architecture, we can achieve high degree of DoS resilience in an open service hierarchy. We also present a thorough analysis to quantify the effectiveness of our design.

9. Conclusion

In this paper we propose HOURS that achieves DoS resilience in an open service hierarchy. HOURS preserves the original hierarchical structure, and augments it with hierarchical overlay networks. When certain nodes are under DoS attacks, user queries are routed across the overlays to bypass the failed nodes and reach the destination. HOURS also exploits several simple mechanisms to enhance each overlay and defeat large-scale topology-aware DoS attacks. The effectiveness of HOURS is confirmed through both analysis and simulations.

HOURS is compatible with the current hierarchical system implementation and incrementally deployable. It works in concert with proactive solutions, such as server replication, that enhance the DoS resilience of individual nodes. Together they create multi-fence against DoS attacks towards building a highly resilient open service hierarchy.

References

[1] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient Multicast using Overlays. In *SIGMETRICS*, 2003.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM*, 1999.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for structured peer-to-peer overlay networks. In *OSDI*, 2002.

[4] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS Using a Peer-to-peer Lookup Service. In *IPTPS*, 2002.

[5] R. Douceur. The Sybil Attack. In *IPTPS*, 2002.

[6] D. Eastlake. Domain Name System Security Extensions. *RFC 2535*, 1999.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. *RFC 2693*, 1999.

[8] A. Habib, M. Hefeeda, and B. Bhargava. Detecting Service Violations and DoS Attacks. In *NDSS*, 2003.

[9] J. Ioannidis, S. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *NDSS*, 2002.

[10] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. In *IMW*, 2001.

[11] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *SIGCOMM*, 2002.

[12] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *STOC*, 2000.

[13] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *PODC*, 2002.

[14] G. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *USENIX USITS*, 2003.

[15] G. Manku. Routing Networks for Distributed Hash Tables. In *PODC*, 2003.

[16] P. Mockapetris. Domain Names - Concepts and Facilities. *RFC 1034*, 1987.

[17] D. Moore, G. Voelker, and S.Savage. Inferring Internet Denial-of-Service Activity. In *USENIX Security Symposium*, 2001.

[18] K. Park, and H. Lee. A Proactive Approach to Distributed DoS Attack Prevention Using Route-based Packet Filtering. In *SIGCOMM*, 2001.

[19] S. Ratnasamy, R. Karp, P. Francis, M. Handley, and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, 2001.

[20] A. Rowstron, and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, 2001.

[21] S. Savage, D. Watherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *SIGCOMM*, 2000.

[22] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[23] H. Yang, H. Luo, Y. Yang, S. Lu, and L. Zhang. HOURS: Achieving DoS Resilience in an Open Service Hierarchy. WING Technical Report, CS Dept, UCLA, 2003.

[24] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *RFC 1777*, 1995.

[25] H. Zhang, A. Goel, and R. Govindan. Using the Small-World Model to Improve Freenet Performance. In *INFOCOM*, 2002.

[26] http://www.caida.org/projects/dns-analysis/oct02dos.xml