DISC-NG: Robust Service Discovery in the Ethereum Global Network

Michał Król*, Onur Ascigil†, Sergi Rene‡, Alberto Sonnino§‡, Matthieu Pigaglio¶,
Ramin Sadre¶, Felix Lange‖, Etienne Rivière¶
*City, University of London, †Lancaster University, ‡University College London, §MystenLabs,
¶UCLouvain, ‖Ethereum Foundation

***Abstract*—The Ethereum Global Network (EGN) hosts a complete ecosystem of decentralized services, including blockchains such as Ethereum *mainnet* but also exchange markets, content delivery networks, and many more. Service discovery is a fundamental mechanism in the EGN, allowing new nodes to look up and connect to participants to one of the services. The current service discovery of the EGN, DISCv4, is not scalable and efficient enough to support the current and future needs of the ecosystem.**

We present DISC-NG, a novel service discovery protocol for the EGN that is scalable, efficient, and secure. DISC-NG leverages the EGN-wide DHT to allow service participation advertisements to meet service discovery requests. DISC-NG compensates the unbalance in service popularity and minimizes the potential for abuse by malicious nodes. We implement DISC-NG in *devp2p*, the network stack used by the majority of clients connecting to the EGN, as well as in a large-scale simulator. DISC-NG can discover services in the EGN faster than DISCv4, while being more robust to malicious nodes. DISC-NG is now in a staging phase and scheduled for deployment in future versions of *devp2p*.

1. Introduction

Following the advent of open blockchains, a large number of decentralized applications emerged to form complete, decentralized service ecosystems. These include storage services (*e.g.* IPFS [50] or Filecoin [43]), exchange markets (*e.g.* Binance [6]), communication systems (*e.g.* Swarm [23]), interoperability networks (*e.g.* Polkadot [54]), in addition to different distributed ledgers at layer 1 (Ethereum mainnet, testnets, and alternatives) and layer 2 (*e.g.* Arbitrum [2], zkSync Era [56]). The literature has so far considered various aspects of decentralized systems including blockchain design and consensus, smart contract platforms, economics, efficient client designs, and the interoperability between different services. In contrast, *service discovery*, a crucial aspect of the security and performance of decentralized ecosystems, received considerably less attention.

Service discovery enables new nodes to join a decentralized service ecosystem and obtain initial contacts with peers already participating in one of its services. Service discovery must not introduce a single point of failure or centralization, yet it is a particularly crucial and sensitive mechanism. It must ensure that malicious participants to the open ecosystem are unable to bias discovery operations against a victim node or service– and that, despite the ability of these adversaries to operate multiple Sybil identities. Of particular importance is the protection against *eclipse* attacks [27], [38], [48], where an adversary would lure its victim(s) into a sub-network formed of only nodes under its control. Similarly, an adversary may run *denial-of-service* attacks [7], [15] against a specific service, preventing other nodes from discovering peers from the associated sub-network.

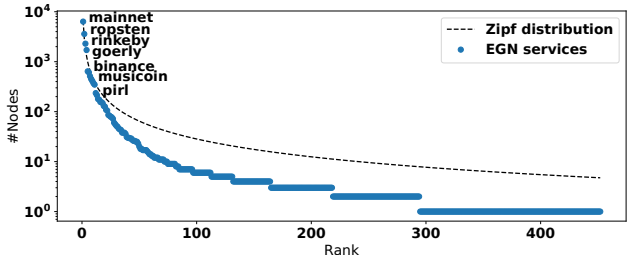


Figure 1: Nodes number distribution of the Ethereum Global Network (EGN) services in September 2023 sorted by decreasing service popularity. A Zipf distribution is given for reference.

With an average of 23,500 live nodes [19], the Ethereum Global Network (EGN) is one of the largest decentralized ecosystems currently in operation. While it is widely known for supporting the Ethereum blockchain (also known as the *mainnet* and supported by about 7,000 nodes), the platform hosts many additional decentralized services. This includes blockchains used for test purposes (*Ropsten*, *Görli*), divergent blockchains resulting from a past fork (*Ethereum classic*), alternative cryptocurrencies (*Pirl*, *Musicoin*), exchange markets (*Binance*), content delivery networks (*Swarm*), or messaging services (*Whisper*). The platform already features almost 500 such services, and their number grows every year [19].

The size distribution of the service-specific sub-networks is very heterogeneous (Figure 1) featuring a *long tail*, with a vast majority of services formed of a few hundred nodes or less. This shows that, in addition to security, performance is of paramount importance for service discovery. The operation must be scalable, and

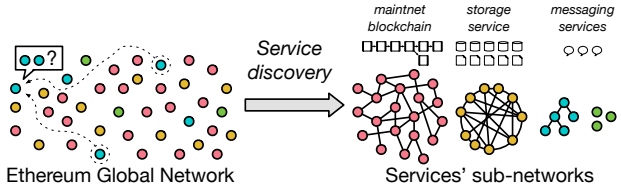


Figure 2: Formation of service-specific sub-networks using the EGN service discovery mechanism.

ensure that the latency and cost of service discovery are not influenced too significantly by the relative popularity of the services. The EGN operates as a unified global network organized as a Distributed Hash Table (DHT) [39], with all nodes participating regardless of their respective services. A new node initially joins this global network and uses service discovery to locate the sub-network formed of peers participating in its service of interest. Service discovery returns a set of peers that are used as entry points to that sub-network, typically supporting a specific overlay network, as illustrated by Figure 2.

The current service discovery mechanism used in the EGN, DISCv4 [18], uses the global DHT to perform *random walks*. A node willing to join a service's sub-network contacts individually a series of nodes collected from random lookups on the DHT, checking service membership with every encountered node until it has collected enough peers participating in its target service. This approach offers good resilience to malicious behaviors but suffers from very poor scalability and performance, in particular for small sub-networks. As more services join the EGN, the inefficiency of the random discovery process introduces a major risk of performance and scalability bottleneck for the entire ecosystem.

Contributions. We present DISC-NG, a novel service discovery mechanism for the EGN. Our protocol enables nodes, members of service sub-networks, to *advertise* their membership to these services in the form of *service advertisements* propagated in the EGN. DISC-NG targets simultaneously (1) robustness, *i.e.* the ability to resist malicious behaviors and Sybil identities, (2) decentralization, (3) efficiency, *i.e.* fast service discovery even for small services, and (4) good load balance over participating nodes. These features rely on two pillar contributions.

First, DISC-NG combines a pseudo-random advertisement placement mechanism that ensures that the density of advertisements for a service increases as DHT lookups get closer to the key associated with that service, and a novel DHT walk mechanism collecting these advertisements. In contrast with the use of direct DHT lookups, these mechanisms protect from common DHT routing attacks [1], [13], [51], [53]. Unlike DISCv4 random walks, however, DISC-NG walks succeed in a predictable number of steps, logarithmic in the network size.

Second, robustness, load balancing, and efficiency for the discovery of smaller sub-networks all rely on a novel *admission protocol*, by which nodes accept or reject incoming service advertisements. The admission protocol ensures that malicious nodes cannot successfully flood the network with advertisements, even when deviating from the protocol or operating Sybils. It also ensures that less popular services get a sufficiently high probability of being represented and found. The core of the admission protocol is a *controlled waiting* mechanism, by which nodes request senders of incoming advertisements to wait and come back after a time to be admitted. A careful design of the functions selecting this waiting time allows to promote diversity of advertisements stored by each node and protects against a vast range of malicious behaviors.

We formally analyze the security (validity and liveness) of DISC-NG and mathematically model its performance. We overcome multiple practical challenges and implement DISC-NG in the code base of *devp2p* [17], the peer-to-peer networking protocols stack used by a majority of the clients connecting to the EGN (including `Go Ethereum` [20], Hyperledger Besu [28], or Nethermind [22]). We evaluate DISC-NG in a 50-server cluster supporting up to 1,000 *devp2p* nodes and with the emulation of world-scale network conditions observed in large decentralized systems. In addition, we employ a simulator whose results are cross-validated with results obtained with the prototype. It allows us to study the behavior of DISC-NG and competing designs with up to 50,000 nodes. Compared to DISCv4, DISC-NG discovers ten times more peers per time slot while achieving a similar or lower probability of being eclipsed by a powerful attacker. Compared to vanilla DHT operations, our system eliminates vulnerability to attacks from resource-constrained attackers and reduces the load on the busiest nodes in the network by two orders of magnitude. DISC-NG is now in a staging phase and scheduled for deployment in future versions of *devp2p*.

Impact. DISC-NG allows small networks to use the large EGN for bootstrapping instead of running their own infrastructure. In contrast to the current DISCv4, our protocol ensures fairness for unpopular services, higher security, fast bootstrap times and better scalability. In the long run, DISC-NG will encourage the development and growth of new applications, promoting decentralization.

2. Background

We detail the operation of the EGN global DHT, which is an evolution of the canonical Kademlia [39] design. Then, we present DISCv4, the current service discovery mechanism operating over this DHT, and its shortcomings.

2.1. The Ethereum Global Network DHT

All nodes in the EGN participate in a global, distributed hash table (DHT). A DHT is a structured overlay network allowing lookup operations, *i.e.* locating a node or group of nodes in charge of a particular *key* in a target *key space* [46], [49]. The EGN uses an evolution of Kademlia [39], a robust and mature DHT design. Every node n in the overlay is assigned a unique identifier (ID) $n.\mathsf{id}$, drawn from the same key space as items, *i.e.* information stored in the DHT. The distance dist between two keys x and y is the logarithm of their bitwise exclusive or (XOR) interpreted as an integer, *i.e.* $\mathsf{dist}(x, y) = \log_2(x \oplus y)$. The Kademlia DHT assigns a key x to the node n with the identifier $n.\mathsf{id}$ that is the *closest* to x according to dist.

Each node n in the overlay maintains a *routing table* $B(n.\mathsf{id})$ centered around n's ID. The routing table is partitioned into m *buckets* $B(n.\mathsf{id}) = [b_0(n.\mathsf{id}), b_1(n.\mathsf{id}), \ldots, b_{m-1}(n.\mathsf{id})]$, where m is the length in bits of nodes' and items' identifiers. Bucket $b_i(n.\mathsf{id})$ contains a list of peers whose IDs share a common prefix of length i with $n.\mathsf{id}$.

The bucket partitioning scheme divides the key space from the point of view of n into disjoint intervals, halving in length every time the bucket's associated prefix includes one more common bit with $n.\mathsf{id}$, as shown by Figure 3 (left). As a result, a node's routing table provides a more detailed (*i.e.* fine-grained) view of the subset of

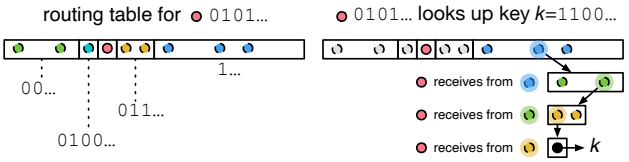


Figure 3: Principle of Kademlia as used in the EGN. Left: routing table with 5 buckets and associated prefixes. Right: iterative lookup process by retrieval of buckets and queries until finding the closest node.

the network with closer node IDs and a less detailed view of nodes with distant IDs. This property is essential for efficiency and enables lookup operations that take a logarithmic number of steps (*hops*) in the number of nodes in the network. It allows a degree of flexibility as each bucket can contain *any* peer sampled from those whose IDs fall within the corresponding interval of the key space.

In Kademlia, a node n_i performing a lookup toward key selects the closest node n_j in its routing table to key and sends it a message. Node n_j returns the closest node to key it knows, and the process repeats until no closer nodes are found. For security, the EGN Kademlia variant hides the precise target key. A node n_i requests an *entire* bucket where key is located from node n_j, and filters them locally before continuing the process (Figure 3, right).

2.2. DISCv4 service discovery

In DISCv4, the service discovery protocol currently used in the EGN, nodes perform *random walks* over the DHT by looking up random keys and performing *handshakes* with all encountered peers. A handshake involves a secure channel establishment between the initiator node and the encountered peer and incurs overhead to both endpoints. If the initiator node discovers during the handshake that the encountered peer is part of the target service's sub-network, it follows this service-specific joining process. The objective of a node is to fill up its *service-level connection slots* with *outbound* connections (*i.e.* connections initiated by the node). The process of node discovery completes when a node fills up all these slots. Nodes also reserve a number of *inbound* connection slots that can be filled with connections initiated by other nodes.

Security and Efficiency. Peer discovery through random walks is reasonably resilient to eclipse attacks. Attackers cannot strategically place Sybil identities in the key space to increase their chance of being discovered by victims, as all locations in the key space have an equal chance of being discovered. On the other hand, the brute-force approach of performing handshakes with all randomly encountered nodes is particularly inefficient. First of all, handshakes with peers not in the target sub-network are wasteful and incur unnecessary overhead. More importantly, for services with low popularity, the number of handshakes can be excessive, *i.e.* on average a node needs an order of a hundred handshakes with other nodes in order to locate a single peer when 1% of the nodes are part of the target service. Because all services are initially unpopular, the upfront cost of building a sub-network can be large, and finding peers can take a long time.

3. System and threat models

We present our system and threat models as well as the target properties of DISC-NG. We summarize the notation used in the paper in Table 1 in Appendix A.

3.1. System model

We assume a network of nodes $N = \{n_0, n_1, ...\}$.[1] At startup time, each node generates a public/secret key pair, which it uses to secure point-to-point communication with its peers. Node n is identified by its *node ID* $n.\mathsf{id}$ (the hash of its public key) and its IP address $n.\mathsf{ip}$.

Multiple nodes may share the same IP address (due to NAT or being hosted by the same physical machine) [38]. However, two nodes cannot share the same ID.

DISC-NG leverages the existing Ethereum DHT but does not rely on its key lookup operations. DISC-NG indexes different sub-networks in the EGN as *services* $S = \{s_0, s_1, ...\}$. A service is represented by an arbitrary string that hashes to a specific key. For brevity, we use s to also refer to the identifier of a service s.

We define the following roles in the system, that all peers play simultaneously.

- **Advertisers** $A(s) = \{a_0, a_1, ...\}$ participate in service s and want to be discovered by their peers. Advertisers $A(s)$ make themselves discoverable by placing advertisements (*i.e.* registering) for service s. An advertisement (ad) maps a service to the advertiser placing the ad. We define the set of advertisers for *all* services as $A = \bigcup_{s \in S} A(s)$. A single node can be an advertiser for multiple services.
- **Discoverers** $D(s) = \{d_0, d_1, ...\}$ attempt to discover advertisers registered under service s, using service lookup operations. We define the set of discoverers for all services as $D = \bigcup_{s \in S} D(s)$.
- **Registrars** $R = \{r_0, r_1, ...\}$ accept ads from advertisers and responds to service queries by discoverers. When asked for a specific service s, a registrar responds with ads containing advertisers that registered for this service. A registrar does not have a specific, assigned service and may store ads for any service.

We assume that the popularity of services in the system is highly heterogeneous, *i.e.* it can follow a power law distribution [32]. Any node can participate in registering and discovering (one or more) services and use the same ID and IP for all its services.

We assume a partially synchronous network model [16], where messages are eventually delivered within a bounded time Δ. We also assume that, at all times, all honest peers (advertisers, discoverers, and registrars) are connected through at least one other honest peer (*e.g.* the honest nodes are not partitioned). Finally, we assume that each honest node is able to receive a bounded amount of traffic per second. We formalize and justify our assumptions in Section 7.

3.2. Threat Model

We assume an open, adversarial environment. We distinguish between honest nodes N^h and malicious nodes N^m with $|N^h| + |N^m| = |N|$. Honest nodes follow

1. $|N|$ is unknown to the participants but is used in our analysis.

the protocol while malicious ones may arbitrarily deviate from it. Malicious nodes can also coordinate their actions. We define sets of malicious A^m, D^m, R^m and honest A^h, D^h, R^h advertisers, discoverers, and registrars respectively. Malicious actors can spawn multiple virtual Sybil nodes within one physical machine, and operate multiple physical machines. We use the number of malicious nodes $|N^m|$ and the number of distinct IP addresses $|n.\mathsf{ip} \in N^m|$ under the control of an attacker as parameters for our evaluation in Section 8.

3.3. Target Properties

Under the considered threat model, DISC-NG achieves the following properties.

Validity. No honest registrar can be tricked into accepting an ad by an advertiser that did not follow the protocol.

Liveness. *(i)* Honest advertisers can register an ad with at least one honest registrar and *(ii)* honest discoverers eventually discover a service that has been advertised by an honest advertiser with an honest registrar.

Decentralization. DISC-NG does not rely on a single trusted node or entity at any point.

Fairness. The system provides efficient lookup and registration operations for all participants regardless of the service they look up or register for. Each advertiser has an equal probability of being discovered by its peers. With nodes complying with the protocol, DISC-NG ensures a balanced load distribution across systems participants and regions of the key space (*i.e.* it avoids *hotspots*).

Efficiency. DISC-NG ensures that the number of nodes contacted and the number of messages exchanged increase logarithmically with the number of system participants $|N|$, for both lookup and register operations. Sending and processing service discovery requests requires only simple operations involving a constant amount of resources. The storage usage for the registrars is limited by a configurable but fixed cap that does not depend on the amount of incoming traffic.

4. Placement of advertisements

We detail the distribution of advertisements in the network and search for service-specific peers.

Challenge. The first challenge in designing a robust service discovery mechanism is to decide on the placement of ads, *i.e. which* registrars should be responsible for storing ads for each service.

A first possibility is to store ads at the closest nodes to the hash of the service ID using traditional DHT operations. Such a solution is *efficient*, as both advertisers and discoverers know how to reach dedicated registrars within a logarithmic amount of steps. Unfortunately, it causes *unequal* load distribution across registrars, especially when the popularity of services varies significantly. Registrars storing popular services would receive a large portion of the registration requests in the network. Finally, this solution is *not secure*, as an attacker could generate and strategically place its Sybil identities, and take control over all the traffic related to a single service.

Alternatively, advertisers could place their ads on random registrars across the entire network. This approach,

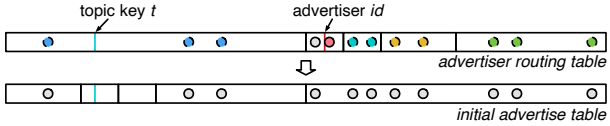


Figure 4: Creating an advertise table from a routing table.

as DISCv4, is *secure* as an attacker would need to take control over the entire network to control a single service. Furthermore, random placement is *fair* and achieves good load balance across registrars regardless of the service popularity distribution. However, and similarly again to DISCv4, random placement is not *efficient*, as it makes it difficult for discoverers to find placed ads, especially for unpopular services.

Distributing ads across registrars. Algorithm 1 presents the DISC-NG advertisement procedure (also shown on Figure 5a). For each service s, its advertisers $A(s)$ continuously maintain up to $K_{register}$ active (*i.e.* unexpired) registrations or ongoing registration attempts in every bucket of an *advertise table* $B(s)$. Ongoing attempts are tracked in a dedicated data structure (Line 1). The advertise table $B(s)$ is similar to the routing table but centered around the service ID rather than the node's ID (Figure 4). It is initialized from the advertiser's routing table (Line 2). At every step, a random registrar in the bucket is chosen (Line 6). The getRandomNode() function remembers already returned nodes and never returns the same one twice during the same ad placement process.

An advertiser $a \in A(s)$ willing to register an ad at a registrar r starts by sending an initial request uniquely containing the ad itself (Line 20). The ad contains the IP of the advertiser $a.\mathsf{ip}$, the service s the ad is for (Line 11), and additional information needed to later contact the advertiser (*e.g.* an application-specific port number). In the remainder of the paper, we omit this additional information in favor of brevity.

Registration may be unsuccessful if the selected registrar is down or refuses to store the ad (Line 28). In this case, the ongoing registration entry is removed allowing a different registrar to be queried. A successful registration follows an admission procedure (Section 5) and places an ad on an advertiser for a fixed amount of time E. The advertisers run Advertise() periodically. Increasing $K_{register}$ makes the advertiser easier to find at the cost of increased communication and storage costs.

Looking up services. Algorithm 2 presents the DISC-NG lookup procedure run by discoverers. Discoverer $r \in R(s)$ runs Lookup(s) periodically and when more peers are requested by its service application. The procedure aims at identifying advertisers $a_1, a_2, ... \in A(s)$ and returning them to the application.

Similarly to advertisers, a discoverer starts with the creation of service-specific *search table* $B(s)$ (Line 2, Figure 4). The lookup starts from the farthest bucket $b_0(s)$ (*i.e.* peers whose ID has the *smallest* common prefix with the service ID) and progresses through all the buckets $b_i(s) \in B(s)$ (Line 4). The discoverer issues a maximum of K_{lookup} queries per bucket.

Making the advertisers $A(s)$ and discoverers $D(s)$ walk towards s's ID in a similar fashion guarantees that the two processes overlap and contact a similar set of

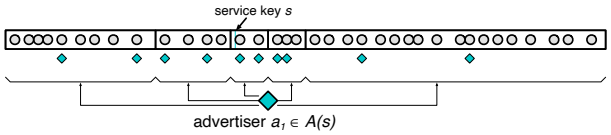
Algorithm 1 Advertisement algorithm run by advertisers.

```
1:  ongoing ← MAP ⟨bucket; LIST⟨registrars⟩⟩
2:  B(s) ← B(self.id)

3:  procedure ADVERTISE(s)
4:      for i in 0, 1, ..., m − 1 do
5:          while ongoing[i].size < K_register do
6:              registrar ← b_i(s).getRandomNode()
7:              if registrar = None then
8:                  BREAK
9:              end if
10:             ongoing[i].add(registrar)
11:             ad.service ← s
12:             ad.ip ← self.ip
13:             SIGN(ad)
14:             async(ADVERTISESINGLE(registrar, ad, i))
15:         end while
16:     end for
17: end procedure

18: procedure ADVERTISESINGLE(registrar, ad, i)
19:     while True do
20:         response ← registrar.Register(ad)
21:         B(s).add(response.neighbors)
22:         if response.status = Confirmed then
23:             SLEEP(E)
24:             break
25:         else if response.status = Wait then
26:             SLEEP(min(E, response.ticket.wait_for))
27:             ad.ticket ← response.ticket
28:         else
29:             break
30:         end if
31:     end while
32:     ongoing[i].remove(registrar)
33: end procedure
```

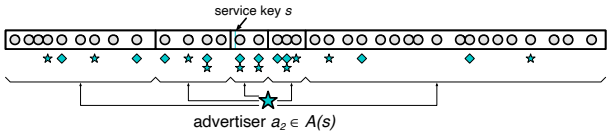
registrars that relay the ads (Figure 5 exemplifies the process for two advertisers $a_1, a_2 \in A(s)$ and one discoverer $d \in D(s)$). At the same time, contacting random registrars in each encountered bucket makes it difficult for an attacker to strategically place malicious registrars in the network. The first bucket $b_0(s)$ covers the largest fraction of the key space as it corresponds to peers with no common prefix to s (*i.e.* 50% of all the registrars). Placing malicious registrars in this fraction of the key space to eclipse a service discovery process would require considerable resources. Subsequent buckets cover smaller fractions of the key space, making it easier for the attacker to place Sybils but also increasing the chance of advertisers already gathering enough ads in previous buckets.

A queried registrar returns at most F_{return} advertiser for s the registrar has in its *ad cache*. A successful search stops when at least F_{lookup} distinct advertisers have been collected (Line 16) or no unqueried registrars remain in any of the buckets (Line 7). Parameters F_{return} and F_{lookup} play an important role in setting a compromise between security and efficiency. A small value of $F_{return} << F_{lookup}$ increases the *diversity* of the source of ads received by the discoverer but increases search time, and requires reaching buckets covering smaller key ranges where eclipse risks are higher. On the other hand, similar values for F_{lookup} and F_{return} reduce overheads but increase the danger of a discoverer receiving ads uniquely from malicious nodes. Finally, low values of F_{lookup} stop the search operation early, before reaching registrars close to the service hash, contributing to a more balanced load distribution.

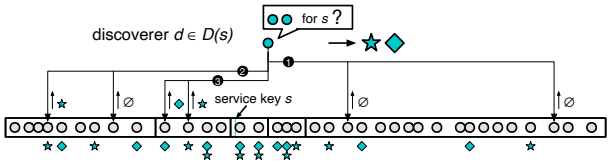
Updating peer tables. The search and advertise tables



(a) Advertiser a_1 pushes ads at two registrars located in each bucket of its advertise table, represented as diamonds.



(b) Advertiser a_2 pushes ads at two registrars located in each bucket of its advertise table, represented as stars.



(c) A discoverer d looks up ads starting from the furthest bucket, and continues until it has collected ads from a sufficient number of different advertisers, two in this example.

Figure 5: Advertisers continuously maintain a fixed number of ads in every bucket. Discoverers use an iterative process to look up ads stored by advertisers, asking registrars in the furthest buckets first, gradually moving towards the service ID. The three figures show how two advertisers push ads to different buckets (5a and 5b), and how a discoverer looks up these ads (5c).

Algorithm 2 Lookup algorithm run by discoverers.

```
1:  procedure LOOKUP(s)
2:      B(s) ← B(self.id)
3:      foundPeers ← SET ⟨peers⟩
4:      for i in 0, 1, ..., m − 1 do
5:          for j in 0, ..., K_lookup − 1 do
6:              peer ← b_i(s).getRandomNode()
7:              if peer = None then
8:                  BREAK
9:              end if
10:             response ← peer.GetAds(s)
11:             for ad in response.ads do
12:                 assert(ad.hasValidSignature())
13:                 foundPeers.add(ad.advertiser)
14:             end for
15:             B(s).add(response.neighbors)
16:             if foundPeers.size ≥ F_lookup then
17:                 return foundPeers
18:             end if
19:         end for
20:     end for
21:     return foundPeers
22: end procedure
```

$B(s)$ are bootstrapped upon their creation using entries from the local node routing table. As the routing table of node n is centered on the peer ID $B(n.\text{id})$ and not on s, the density of peers around s might be low or even null, particularly when n.id and s are distant in the key space. The buckets are thus filled opportunistically while interacting with peers during the search or advertisement process. Registrars, apart from responding to queries, return a list of peers that are later used to populate $B(s)$ (Algorithm 1 Line 21 and Algorithm 2 Line 15).

Malicious registrars could return large numbers of

Algorithm 3 Returning peers by registrars to nodes traversing the DHT.

```
1: procedure GETPEERS(s)
2:     peers ← SET ⟨peers⟩
3:     B(s) ← B(self.id)
4:     for i in 0, 1, ..., m − 1 do
5:         peers.add(b_i(s).getRandomNode())
6:     end for
7:     return peers
8: end procedure
```

malicious nodes in a specific bucket attempting to poison the routing process. To limit this risk, a local node communicating with a registrar r asks it to return a single peer per bucket from r's view of $B(s)$(Algorithm 3). Contacting registrars in consecutive buckets divides the search space by a constant factor, and allows learning new peers from more densely-populated routing tables towards the destination. The procedure mitigates the risk of having malicious peers polluting the table while still learning rare peers in buckets close to s.

5. Admission Protocol

We describe the registration procedure followed by an advertiser when attempting to register an ad at a registrar.

Challenge. Registrars have a limited amount of memory and can store only a finite number of ads. If the registration demand surpasses the supply, each registrar has to decide which ads should be admitted. In an open setting, implementing simple replacement policies such as Least Recently Used (LRU) or Least Frequently Used (LFU) exposes the system to an attacker that bombards registrars with ads and evicts honest ads of honest advertisers.

Admission. Algorithm 4 presents the admission procedure run by registrar r to decide whether to admit ad coming from advertiser a. Registrars store admitted ads in a data structure called an *ad cache*. Each ad stored in the ad cache has an associated expiry time E, after which the ad is automatically removed. The total size of the ad cache is limited by its capacity C. DISC-NG does not impose service/IP/ID-specific limits on the content of the *ad cache* to accommodate for diverse network conditions and application popularity distributions.

Advertiser $a \in A(s)$ may place at most one ad for a specific service s in the ad cache of a given registrar r. Registration requests for ads already in the cache are ignored (Line 2). Based on the content of the ad cache and ad, r calculates an ad-specific waiting time (Line 4). We detail the calculation of the waiting time in Section 6. If r decides that a has to wait (*i.e.* $\mathsf{t_{remaining}} > 0$ in Line 17), r does not store ad but instead issues a *ticket*.

Tickets are digitally signed objects issued by registrars to advertisers to reliably indicate how long an advertiser already waited for admission. Each ticket contains a copy of ad, the ticket creation time $\mathsf{t_{init}}$, the ticket last modification time $\mathsf{t_{mod}}$, and the time that a still has to wait for before being admitted $\mathsf{t_{wait\_for}}$.

Upon reception of a ticket, a waits for the indicated time $\mathsf{t_{wait\_for}}$ and attempts to register again at r. The consecutive registration request must include the last ticket issued by r. Tickets can be used uniquely during a registration window Δ (Line 14). Δ is chosen to accommodate for the maximum delay between the advertiser and the registrar. This prevents an attacker from gathering many tickets, accumulating long waiting times (Section 6), and submitting the tickets all at once to overwhelm the registrar. Advertisers only read the wait time $\mathsf{t_{wait\_for}}$ from the ticket and do not use the creation ($\mathsf{t_{init}}$) or the modification ($\mathsf{t_{mod}}$) timestamps. As a result, DISC-NG does not require clock synchronization between advertisers and registrars.

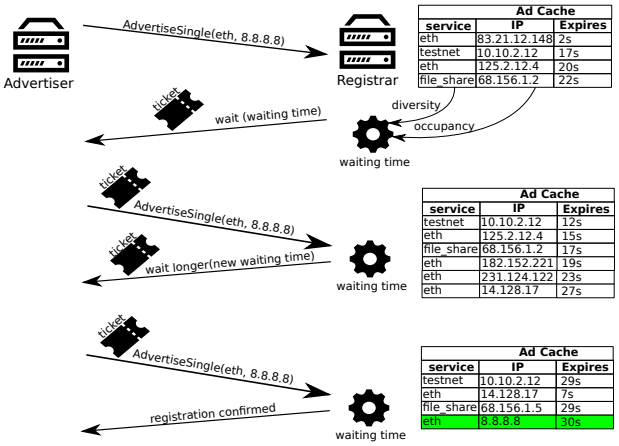


Figure 6: DISC-NG admission protocol. The advertiser sends its ad to the registrar which calculates a waiting time and issues a ticket. The advertiser waits for the indicated time and attempts to register again including the ticket. Due to additional ads admitted to the ad cache in the meantime, the advertiser has to wait again and receive an updated ticket. The advertiser eventually accumulates enough waiting time and registers at the registrar.

Importantly, the waiting time $\mathsf{t_{wait\_for}}$ is not binding. At every registration attempt from a, r calculates a new waiting time $\mathsf{t_{wait\_for}}$, based on the current content of the ad cache. The remaining waiting time $\mathsf{t_{remaining}}$ is calculated as a difference between $\mathsf{t_{wait\_for}}$ and the time a has already waited for, as indicated in the ticket (Line 15). With consecutive registration attempts, a increases its accumulated waiting time and eventually will be admitted (Line 18). If a misses its registration window or does not include its last ticket, it loses all its accumulated waiting time and has to start the admission procedure from scratch.

Discussion. The admission protocol allows registrars to prioritize advertisers that have been waiting for the longest time. At the same time, combining non-binding waiting times with immutable tickets stored and the advertiser side means that the registrars are not required to maintain any state for each ongoing request before admitting an ad into the ad cache. This feature protects against advertisers who do not come back and against DoS attacks that would attempt to exhaust registrars' memory.

6. Waiting Time

The waiting time determines the time advertisers have to wait before being admitted to the ad cache. The function directly shapes the structure of the ad cache, determines its diversity, and performs flow control. Each request is given a waiting time based on the ad itself and the current state of the ad cache, according to Equation (1).

Algorithm 4 Admission algorithm run by registrars.

```
1:  procedure REGISTER(ad, ticket)
2:      assert(ad not in ad_cache)
3:      response.ticket.ad ← ad
4:      t_wait ← CALCULATEWAITINGTIME(ad)
5:      if ticket.empty() then
6:          t_remaining ← t_wait
7:          response.ticket.init_time ← NOW()
8:          response.ticket.mod ← NOW()
9:      else
10:         assert(ticket.hasValidSignature())
11:         assert(ticket.ad = ad)
12:         assert(ad.notInAdCache())
13:         t_scheduled ← ticket.mod + ticket.wait_for
14:         assert(t_scheduled ≤ NOW() ≤ t_scheduled + Δ)
15:         t_remaining ← t_wait − (NOW() − ticket.init)
16:     end if
17:     if t_remaining ≤ 0 then
18:         ad_cache.add(ad)
19:         response.status ← Confirmed
20:     else
21:         response.status ← Wait
22:         response.ticket.wait_for ← MIN(E, t_remaining)
23:         response.ticket.mod ← NOW()
24:         SIGN(response.ticket)
25:     end if
26:     response.neighbors ← GETPEERS(ad.s)
27:     return response
28: end procedure
```

Algorithm 5 Lookup response algorithm run by registrars.

```
1:  procedure LOOKUPRESPONSE(ad)
    assert(ad.hasValidSignature())
2:      response.peers ← ad_cache.getPeers(ad.s)[: F_return]
3:      response.neighbors ← GETPEERS(ad.s)
4:      return response
5:  end procedure
```

$$w(\mathsf{ad}) = \underbrace{E}_{\text{scaling}} \times \underbrace{\frac{1}{(1-\frac{c}{C})^{P_{occ}}}}_{\text{occupancy score}} \times \left(\underbrace{\frac{c(\mathsf{ad}.s)}{c}}_{\text{service similarity}} + \underbrace{score(\mathsf{ad.IP})}_{\text{IP similarity}} + \underbrace{G}_{\text{safety}} \right) \quad (1)$$

In the remainder of this section, we describe the individual components of Equation (1).

Scaling. The waiting time is normalized by the amount of time each ad spent in the cache E (*i.e. expiry time*). It binds the absolute values of the returned waiting time to E and allows us to reason about the number of incoming requests regardless of the time each ad spends in the ad cache.

Occupancy score. The occupancy score progressively increases the waiting time as the ad cache fills up and limits the memory used by a registrar. c is the number of ads already in the ad cache, C is the capacity of the ad cache, and P_{occ} is a protocol parameter. When the number of ads in the cache is low ($c \ll C$), the occupancy score goes to 1. As the ad cache fills up, the score will be amplified by the divisor of the equation. The higher the value of P_{occ}, the faster the increase. With occupancy c close to the capacity C, the occupancy score goes to infinity thus limiting the number of admitted requests.

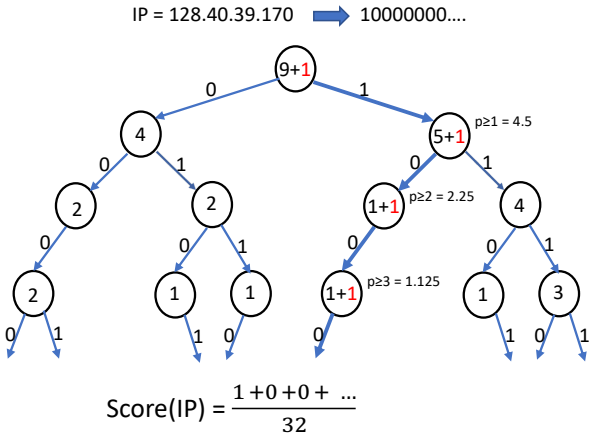


Figure 7: Inserting an IP address into the IP *tree* structure.

Service Similarity. The service similarity determines how similar the incoming request is to the ads already in the ad cache in terms of service ID. $c(\mathsf{ad}.s)$ is the number of ads for service s already in the cache. Requests significantly different from the current content of the cache ($c(s) \approx 0$) receive lower similarity scores resulting in lower overall waiting times. The similarity goes to 1 as the specified service dominates the cache $c(s) \approx c$. Such an approach promotes fairness across services, *i.e.* it is easier for less popular services to get into the cache.

IP Similarity. The similarity score used for services cannot be securely applied to IPs. It is easier for an attacker to generate similar IP addresses (*i.e.* within a single subnet) than to control many diverse ones (with different prefixes). An attacker could thus easily generate multiple addresses that would receive low *similarity scores*. A common solution is to limit the number of IPs coming from the same subnetwork [20], [38]. Unfortunately, it is impossible to reliably set those limits without knowing the network size or NAT configuration of the honest nodes. DISC-NG implements a more versatile approach that directly captures the similarity across different IPs and translates it into a numerical score. We introduce a *binary tree* (Figure 7) that stores IPs used by ads in the cache.

Each tree vertex stores a counter, while the edges represent consecutive 0s or 1s in a binary representation of IPs. Apart from its root, the *tree* consists of 32 levels (33 levels in total) representing bits in the binary representation of IPv4 IP addresses[2]. Algorithm 6 presents the pseudocode for adding an IP address to the *tree*. The counter of every *tree* vertex is initially set to 0. When adding an IP to the *tree*, the address is first converted to its binary representation (Line 4) and follows a path in the *tree* corresponding to consecutive bits (Line 7). Counters of all the visited vertices are increased by 1 (Line 6). As a result, the root counter stores the number of all the IP addresses in the ad cache, its 0 successor stores the number of the IPs starting with 0, root's 1 successor stores the number of the IP addresses starting with 1 and so on.

When the counter of a visited vertex is higher than it should be in a perfectly balanced tree (Line 12), a point is added to the score (Line 13). In the end, the

2. For simplicity, we present the *tree* for IPv4 addresses but its adaptation for IPv6 is straightforward.

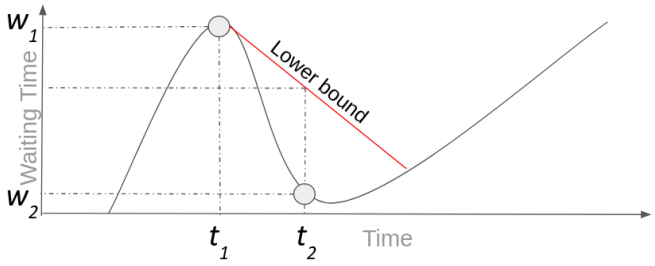


Figure 8: Waiting time lower bound.

similarity score for an IP is normalized by the length of the IP address, and thus the maximum score value (Line 16). The add procedure is used to calculate the IP score every time the waiting time is calculated. However, the vertex counters are increased in Line 6 only when an ad is admitted to the cache. When an ad expires, its IP is removed from the *tree*, and the counters are decreased. The IP similarity ranges from 0 to 1 and returns values closer to 1 for IPs sharing the same prefix and protects against filling the cache by advertisers with a small number of distinct IPs.

Algorithm 6 Adding an address to the IP tree.

```
1:  procedure ADD(tree, IP)
2:      v ← tree.root
3:      score ← 0
4:      bits ← IP.to_binary()
5:      for i in 0, 1, ..., 31 do
6:          v.counter ← v.counter + 1
7:          if bits[i] = 0 then
8:              v ← v.left
9:          else
10:             v ← v.right
11:         end if
12:         if v.counter > tree.root.counter / 2^i then
13:             score ← score + 1
14:         end if
15:     end for
16:     return score / 32
17: end procedure
```

Safety. A resourceful attacker could send ads for random topics (topic similarity ≈ 0) and from diverse IPs (IP similarity ≈ 0). To prevent the cache from overflowing in such cases, the safety parameter G ensures that the waiting time never reaches 0.

Lower Bound. Every change in the ad cache may increase or decrease the waiting times of other pending requests. Therefore, an advertiser receiving waiting time w_1 at time t_1, may get a smaller waiting time w_2 at time t_2 ($t_1 < t_2$) in case the content of the ad cache is different (*e.g.* when an ad for the same service expires between t_1 and t_2). As a result, advertisers are incentivized to frequently send new ticket requests hoping to get a better waiting time and generating unnecessary overhead in the system. To prevent this, DISC-NG ensures that any advertiser already in possession of a ticket, cannot get a better waiting time by sending new ticket requests, as explained next.

When asking for a new waiting time before the previously obtained one elapses, an advertiser loses its already accumulated waiting time (including the previous ticket allows the registrar to ignore the request). This means that asking for a new waiting at time t_2 can lower the overall waiting only if the new waiting time w_2 is smaller than w_1 by more than the time elapsed $t_2 - t_1$: $w_1 - w_2 < t_2 - t_1$. To avoid it, DISC-NG enforces a lower bound on the waiting time. We make sure that an advertiser's waiting time received at t_2 is not smaller than the waiting time at t_1 ($t_1 < t_2$) by more than $t_2 - t_1$ (Figure 8).

Keeping lower-bound information associated with previously received ads goes against the objective of storing no information at the registrar related to pending requests. The unbounded memory overhead induced by this storage could be exploited by an attacker. We observe, however, that storing bounds for every request is not necessary. To illustrate why, let us consider a rewrite of the waiting formula of Equation (1) as a summation:

$$w(\mathsf{ad}) = \frac{E \cdot G}{(1-\frac{c}{C})^{P_{occ}}} + \frac{E \cdot c(\mathsf{ad}.s)}{c(1-\frac{c}{C})^{P_{occ}}} + \frac{score(\mathsf{ad}.\mathsf{IP})}{(1-\frac{c}{C})^{P_{occ}}} \tag{2}$$

In this summation, the total waiting time will respect the lower bound as soon as a corresponding lower bound is enforced for each of the three components. As a result, a registrar only needs to store lower-bound information for every different IP, and every different service already in the ad cache. In contrast with the (unbounded) stream of incoming ads, these two sets have a bounded size.

When service s enters the cache for the first time, $\mathsf{bound}(s)$ is set to 0, and a $timestamp(s)$ is set to the current time. When a ticket request arrives for the same service, we calculate the service waiting time w_s and return the value $w_s = max(w_s, bound(s) - timestamp(s))$. The bound and the timestamp are updated when a new ticket is issued and $w_s > (bound(s) - timestamp(s))$.

We also maintain a lower-bound state for the ticket holders' IP addresses in the IP tree structure: the state for an IP address is maintained at the node, which corresponds to the longest prefix match in the existing tree (without introducing new nodes). We also aggregate the lower bound states of multiple IPs mapping to the same node by applying a $max()$ function.

7. Formal Analysis

We formally argue the properties of DISC-NG. Whenever referring to an *honest* party, we mean a party that follows the protocols specified below.

Definition 1 (Honest advertiser)**.** *An honest advertiser is an advertiser that follows Algorithm 1.*

Definition 2 (Honest registrar)**.** *An honest registrar is a registrar that follows Algorithm 4 when interacting with an advertiser and Algorithm 5 when interacting with a discoverer.*

Definition 3 (Honest discoverer)**.** *An honest discoverer is a discoverer that follows Algorithm 2.*

7.1. Validity

Intuitively, validity ensures that no registrar can be tricked into accepting an ad by an advertiser without a valid ticket.

Definition 4 (Valid ticket)**.** *A valid* ticket *issued by a registrar r over an ad x is a statement uniquely referencing x and a timestamp of the ticket's creation; and a signature by r over those fields.*

Theorem 1 (Validity): No honest registrar r admits an ad x without a valid ticket (Definition 4) referencing x.

Proof. The proof can be found in Appendix B.1. □

7.2. Liveness

We show that an honest discoverer eventually discovers a service that has been advertised by an honest advertiser with an honest registrar. The liveness of DISC-NG relies on assumptions that we initially described in Section 3 and formalize below.

Assumption 1 (Partial synchrony [16])**.** *There exists a known Δ and a global stabilization time (GST) after which all messages sent among honest nodes are delivered within the network delay Δ. All messages sent among honest nodes are eventually delivered.*

We justify this assumption by noting that no existing EGN service operates under a weaker network assumption. Furthermore, the Ethereum *mainnet* blockchain requires a synchrony assumption strictly stronger than Assumption 1 [24].

Assumption 2 (No eclipses)**.** *At all times, all honest nodes (advertisers, discoverers, and registrars) are connected through at least one other honest node.*

The assumption is standard for most peer-to-peer systems [26], [31]. EGN already implements multiple mechanisms preventing partitioning honest nodes at the DHT level [27], [38].

Assumption 3 (Maximum traffic)**.** *The maximum traffic that can be received by a single registrar is $B_{max} < \frac{S(C-1)}{E}(1 + \frac{G}{(1/C)^{P_{occ}}})$, where G and P_{occ} are system parameters (see Table 1), S is the size of the messages, E is the ad expiration time, and C is the maximum capacity of the registrars' cache.*

We motivate Assumption 3 by our choice of systems parameters (Table 1). Using these parameters we find that $B_{max} < 11^{25}$ bps. This upper bound is much larger than the currently achievable speed record of about 10^{14} bps [44]; we hence argue about the practicality of Assumption 3. In other words, we assume that no registrar (even if attacked by an attacker with infinite resources) can receive and process traffic that matches or exceeds B_{max}.

Advertiser liveness. An honest advertiser can eventually register an ad with at least one honest registrar. Intuitively, honest advertisers receive tickets indicating a minimum waiting time before their ad can be registered. The cache of the registrars has a bounded capacity, but ads expire and are removed from the cache after a fixed amount of time. As a result, we argue that it is impractical for an adversary to overwhelm the cache of an honest registrar because doing so would necessitate meeting unmanageable bandwidth demands.

Theorem 2 (Advertiser liveness): An honest advertiser a can eventually register an ad x with an honest registrar r.

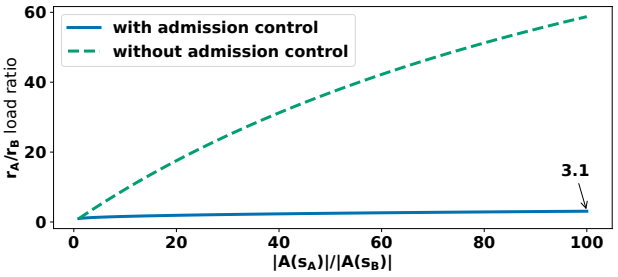


Figure 9: Load ratio between registrars located closest to most popular (r_A) and least popular (r_B) services for $K_{register} = 5, |N| = 25000, |A(s_A)| + |A(s_B)| = 15000$.

Proof. The proof can be found in Appendix B.2. □

Discoverer liveness. An honest discoverer seeking a service eventually connects with an honest advertiser offering this service. Intuitively, an honest advertiser can eventually register an ad for its service with a set of honest registrars, and an honest discoverer eventually discovers these registrars and thus the advertiser offering the service.

Theorem 3 (Discoverer liveness): An honest discoverer d eventually discovers service s advertised by an honest advertiser a with an honest registrar r.

Proof. The proof can be found in Appendix B.3. □

7.3. Decentralization

DISC-NG does not rely on a single trusted entity at any point. The underlying DHT represents a decentralized, permissionless system. Any node can generate a node ID close to the identifier of a particular service. Furthermore, DISC-NG ads placement and discovery procedures (Section 4) contact randomly chosen nodes in every DHT bucket making sure that the data is stored to and obtained from multiple locations.

7.4. Fairness

We argue that the load distribution across the registrars due to registration and lookup processes exhibits a relatively low degree of imbalance. For simplicity, we compare the load on registrar r_A located closest to the most popular service s_A and the load on registrar r_B located closest to the least popular service s_B. We assume the IDs of s_A and s_B to be located on the opposite sides of the DHT key space, the worst-case scenario for load balancing.

Empirical analysis of registration load. Figure 9 presents the registration load ratio between both registrars as a function of increasing imbalance of popularity between the two services (expressed as the ratio of the number of their respective advertisers $|A(s_A)|$ and $|A(s_B)|$), assuming advertisers with IP address diversity. The figure was obtained by mathematically modeling the load caused by advertisers for s_A and s_B and assuming a small uniform background load from other services at both registrars, as described in more detail in Appendix C.2.

When s_A is a hundred times more popular, r_A receives only 3.1 times more requests than r_B with admission control. On the other hand, when all the requests receive a

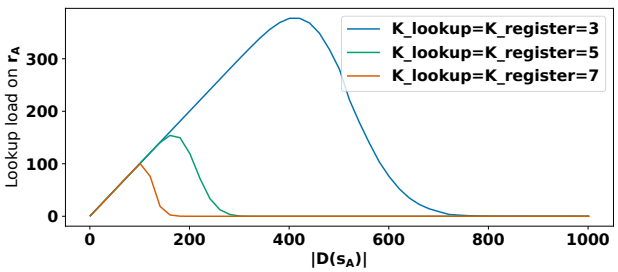


Figure 10: r_A's lookup load as a function of increasing popularity of s_A for $|N|$ =25000, $|D(s_B)|$=100.

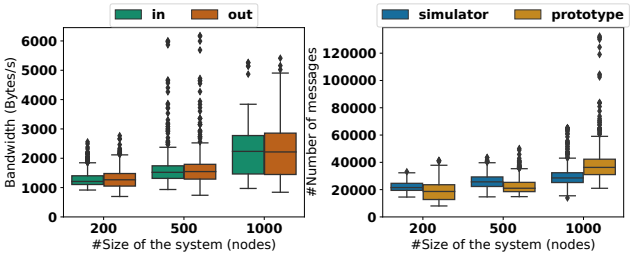


Figure 11: Incoming and Outgoing bandwidth distribution for increasing system sizes in the testbed cluster (left) and corresponding distribution of message counts in the testbed and the simulator (right).

fixed waiting time (*i.e.* no admission control), r_A receives nearly 60 times more requests than r_B.

Initially, r_A receives more requests than r_B. However, as r_A admits ads, the waiting time for the remaining registration attempts increases due to the increasing occupancy c and service similarity component $\frac{c(s_A)}{c}$. The increased waiting time slows the rate of incoming requests and limits the traffic received by r_A.

Empirical analysis of lookup load. Figure 10 presents r_A's load due to lookups as a function of the popularity of s_A. For this figure, we fixed the number of advertisers and discoverers for the unpopular service $|D(s_B)| = |A(s_B)| = 100$ and gradually increased the number of discoverers $D(s_A)$ looking for service s_A as well as the number of its advertisers, assuming $|D(s_A)| = |A(s_A)|$. We include more details in Appendix C.2.

Depending on the $K_{register}$ and K_{lookup} parameters, the maximum load is experienced when the number of advertisers/discoverers $|D(s_A)| = |A(s_A)|$ is relatively small. Eventually, the lookup load goes back to zero when s_A becomes more popular because as the number of advertisers and discoverers increases the probability that the discoverers will finish their search before reaching r_A increases as well. We observe the same behavior for different $K_{register}$ and K_{lookup} values as shown in Figure 10.

A discoverer looking for s_A will start at the furthest bucket $b_0(s_A)$ and progress toward $b_{m-1}(s_A)$ until it finds F_{lookup} ads. Initially, this increases the load on r_A located closest to the s_A. However, the more participants in s_A, the more ads will be placed and discovered in early buckets $b_i(s)|i << m$. As a result, an increasing number of discoverers will terminate their lookup operations (finding F_{lookup} nodes) before reaching r_a.

8. Performance Evaluation

We evaluate DISC-NG in two steps. First, we evaluate a prototype using a testbed cluster of 50 servers supporting up to 1,000 nodes. We use the testbed results to validate the results of a simulator, allowing us to perform tests at a much larger scale with confidence in the results, reaching up to 50,000 nodes. For all the experiments, we assign node IDs and IPs randomly drafted from the EGN [19].

8.1. Testbed and simulator validation

We first describe our evaluation of the performance and cost of the DISC-NG prototype. We implement DISC-NG in *devp2p* [17], the network stack used by the EGN, and integrate it with the *Go Ethereum* (Geth) client [20] as a service application.

We deploy instances of *devp2p* and Geth in a cluster of 50 servers, each equipped with an 18-core Intel Xeon Gold 5220 CPU and 96 GB of RAM. We performed careful load tests and observed that each server could support up to 20 instances of DISC-NG with no visible impact (*i.e.* no contention on CPU resources, no memory shortage, and no increase of application latencies). We consider, therefore, network sizes of 200, 500, and 1,000 nodes.

Network emulation. DISC-NG is intended to operate with WAN latencies. The fast 25-Gbps interconnect linking our servers is not representative of such a setting. We rely, therefore, on network emulation using the Linux tool `tc` to reproduce the characteristics of a planetary-scale deployment. As there is no publicly available data of node-to-node latencies in the EGN, we use a similar dataset [34] collecting the all-pair latencies in IPFS, a large-scale storage network that has a similar scale and decentralization objective as the EGN. Roundtrip latencies range from 8 ms to 91 ms, with an average of 34 ms. We further limit the connection capacity of each node to 20 KBytes/s.

Setup. We use 30 services whose popularities follow a Zipf distribution (with parameter $\alpha = 1$), as perceived in Figure 1. Each node constantly registers for its assigned service and performs 5 lookup operations, aiming to discover $F_{lookup} = 30$ peers. We summarize and justify all the default system parameters in Table 1 in Appendix A.

Simulator validation. We concurrently run the same experiments in a simulator of DISC-NG that we developed using PeerSim [40], a scalable P2P network simulator.

Bandwidth usage. We start by studying the resource usage of DISC-NG. As advertisers periodically push ads to the system, and as discoverers periodically attempt to locate peers for their service, DISC-NG incurs a constant flow of messages. Figure 11 presents the distribution of bandwidth for different sizes of the network. We observe that the incoming and outgoing bandwidth are well balanced across peers (*i.e.* no peer acts as a bottleneck) and that the overall bandwidth budget is very modest, and well below the capacity of our emulated links. As the system grows in size, the median bandwidth cost increases, with outliers that are still well below the typical capacity of an internet link. In the second plot, we compare the distribution of the number of messages in the testbed and those obtained in the simulation. The distributions match

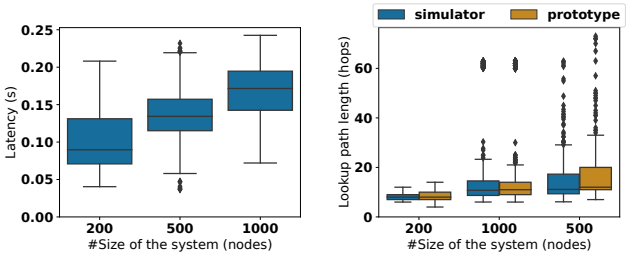


Figure 12: Distribution of latencies for looking up at most 30 advertisers in the network for increasing system sizes (left), and comparison of the corresponding lookup path length in the testbed and the simulator (right).

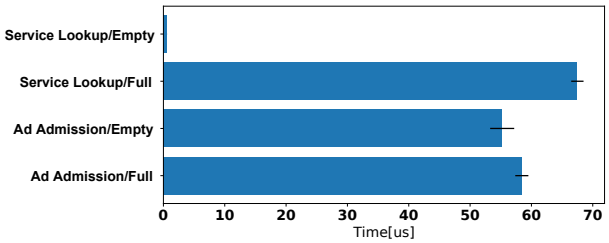


Figure 13: Request processing time.

and allow us to consider the number of messages as a valid indicator of bandwidth usage in the simulations.

Service latencies. We now evaluate the latency of discovery requests, presented in Figure 12. Each discoverer attempts to find up to 30 advertisers in the network. Discovery latency grows moderately with increasing system sizes and remains in the order of a few hundred milliseconds. The distribution of lookup path length aligns with this observation and the lookup path lengths are comparable in the testbed and simulations. It makes this metric a valid indicator of service latency in simulations.

Local processing time. We zoom in on the local processing time required for the different operations performed by discoverers and advertisers (Figure 13). We consider cases with an empty and a full *ad cache*. Lookups performed on an empty *ad cache* are the fastest as they do not require processing the *ad cache* content. Lookups performed on a full *ad cache* require $66\mu s$. The registration admission operation requires $58\mu s$ and is not heavily influenced by the increasing number of stored ads. For all operations, we observe consistent processing time across multiple runs.

Finding advertisers. We analyze the average number of lookup results obtained by each node, in the largest configuration with 1,000 nodes. We focus on the 20 most popular services. Figure 14 groups nodes by their service, as indicated by the colors of the bars (order within one color is random). Services are sorted from highest to lowest popularity, as can be seen by the width of the color zones. The vast majority of discoverers are always able to find the required number of peers. Less popular service nodes (right side of the graph) do not always obtain enough results. This is caused by the lower number of participants for those services (< 30). Importantly, within each service, we observe the same number of obtained results indicating high fairness.

Being found by discoverers. Figure 15 presents the number of times each advertiser is discovered by others.

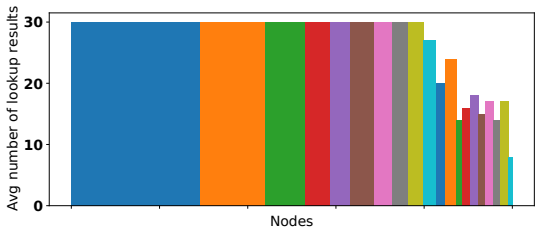


Figure 14: Average number of lookup results obtained by each node.

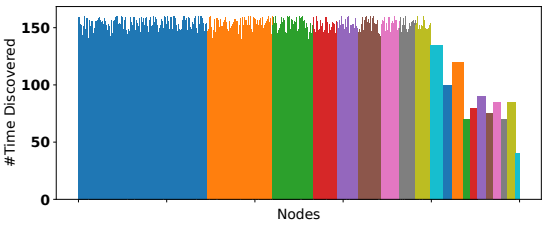


Figure 15: Number of times each node is discovered during lookup.

It allows us to verify that no advertiser is discriminated against in the network. We use the same visual grouping as in Figure 14. Within each service, nodes are discovered with uniform distribution, and we did not observe advertisers not discovered by their peers. The distribution closely matches the one observed in Figure 14.

Waiting times. At first, the equal discovery rate across services with different popularity might be surprising. To investigate it further, for each service, we plot the average total waiting time received by its advertisers in Figure 16. We express the waiting time in ad expiry time E and keep the service colors consistent with the one used in Figure 14 and Figure 15. DISC-NG returns higher waiting times to advertisers from popular services. As a result, those nodes spend less time in the ad caches and more time waiting to be admitted. This mechanism allows advertisements of less popular services to get a fair share of ad cache space and results in an equal discovery rate.

8.2. Simulations

Setup. In addition to DISC-NG, we implement the following three baselines in the simulator.

- **DISCv4** - described in Section 2.2.
- **DHT** - storing ads on 16 closest nodes of the service ID, using vanilla Kademlia DHT lookup operations as used in InterPlanetary File System (IPFS) [35], [50], with LRU as the replacement policy for ads.
- **DHTTicket** - the same as DHT, but using DISC-NG admission control (Section 6) instead of LRU.

We set the number of services $|S| = 300$ with their popularity following a Zipf distribution with exponent 1.0 (Figure 1). We consider a default size of 25,000 nodes, recreating the current EGN topology, that we extend to up to 50,000 nodes. Each simulation takes one hour of simulated time during which each advertiser tries to maintain active (*i.e.* unexpired) registrations and each node performs a single lookup operation uniformly spread across the simulation time. We provide a full list of the default simulation parameter values in Table 1 in Appendix A.

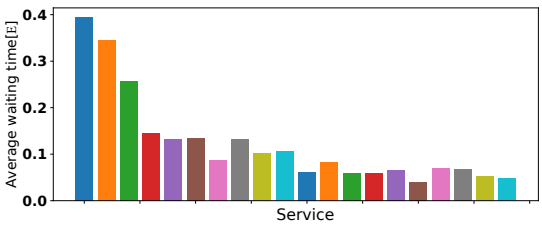


Figure 16: Avg. total waiting time received per service.

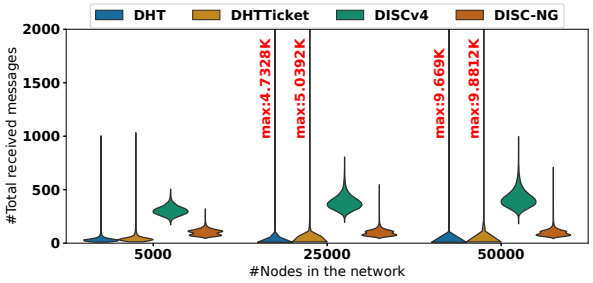


Figure 17: Message overhead for different network sizes.

In the rest of this section, we use violin plots [55] for a compact illustration of both the distribution and the density of data points. We limit the shape of the violins within the range of the observed data and set the widths of each data point in the violin proportional to the count of observations at that data point. We display the maximum observed values (in red) if the range of the data points exceeds the range of the y-axis.

Message overhead. Figure 17 and Figure 18 present the number of messages received per node for the registration and lookup operations. For DISCv4, only lookups generate messages as there is no registration process in the protocol. DHT-based protocols introduce low average overhead but overload nodes close to popular service IDs impacting fairness. The DISCv4 protocol has a better load distribution between nodes since the destination of lookup messages is chosen randomly. However, its average message overhead is the highest among all protocols. DISC-NG introduces average overhead but provides more equal load distribution compared to DHT-based protocols.

Finding Advertisers. Figure 19 presents the number of *discovered advertisers* during a single lookup operation. With a fixed number of services and increasing network size, each service-specific network grows and it is easier to find the required $F_{lookup} = 30$ number of advertisers. However, DISCv4 again suffers from poor performance for all the investigated network sizes due to its random movement in the network. DHTTicket has worse performance than DHT because the high contention when registering in only 16 nodes for the same service, causes high waiting times and limits the number of nodes discovered. DISC-NG has a discovery performance close to DHT.

Figure 20 presents the number of *discovered advertisers* but with an increasing number of services. The random walk of DISCv4 discovers a relatively low number of results per operation (< 5). The performance decreases with an increasing number of services in the network. DHTTicket has a performance between DHT and DISCv4. DISC-NG and DHT solutions discover the required number of 30 advertisers. The rare cases where those protocols do not discover the required amount of peers. are caused by services with ≤ 30 participants.

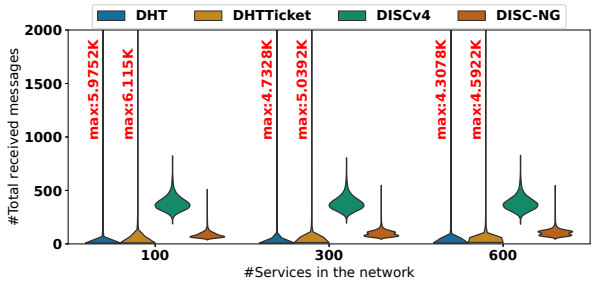


Figure 18: Message overhead for a different number of services during a single advertisement period.

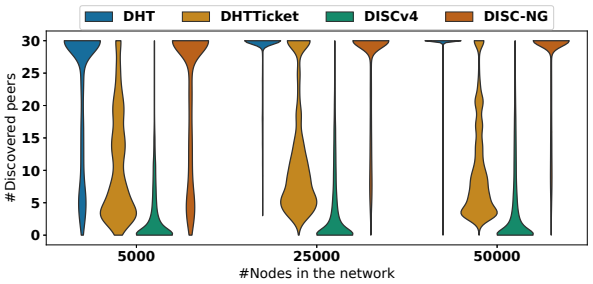


Figure 19: Discovered peers for different network sizes.

Simulating malicious nodes. We evaluate DISC-NG resistance against eclipse attacks. Malicious nodes return uniquely other malicious peers for both the DHT routing and service lookups. Furthermore, malicious advertisers place ads at honest registrars at a rate 10 times higher than honest nodes. By default, $|N^m| = \frac{|N|}{3}$ nodes are malicious. We introduce a *#Attackers reusing IPs* parameter defining the number of attacker nodes reusing the same IP addresses (5 by default). Lower values mean diverse IPs of malicious nodes but increase costs for the attacker. We use attacker IPs having minimum similarity with the honest IPs to obtain an upper bound on the impact of attacks. An honest node is eclipsed when all advertisers obtained in a lookup are malicious.

On top of each violin plot, we specify the *percentage of eclipsed (honest) lookup operations*. We assume that the attacker identifiers are uniformly distributed in the address space. We omit the results for scenarios with non-uniform Sybil ID distributions. In that case, the DHT and DHTTicket suffer from 100% eclipse rates when the attacker places 16 malicious nodes close to a target service ID, compared to a 0% eclipse rate for DISCv4 and DISC-NG. The results below represent the worst-case scenario for DISC-NG and the best-case scenario for protocols we compare against. We target an moderately popular service ($\approx$ 500 nodes participating in the service).

Lookup eclipse resistance. Figure 21 illustrates the percentage of malicious nodes in the lookup results and the lookup eclipse rate. DISC-NG achieves $\approx 0\%$ eclipse rate, even with a high number of malicious nodes and is the most resistant protocol. The admission control mechanism combined with the ad placement strategy, forms an efficient protection even against an attacker controlling 50% of the nodes. Surprisingly, the random approach of

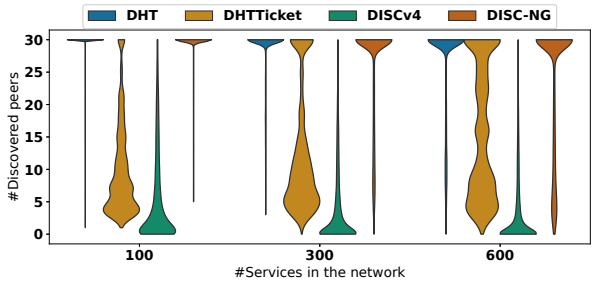


Figure 20: Discovered peers vs. the number of services.

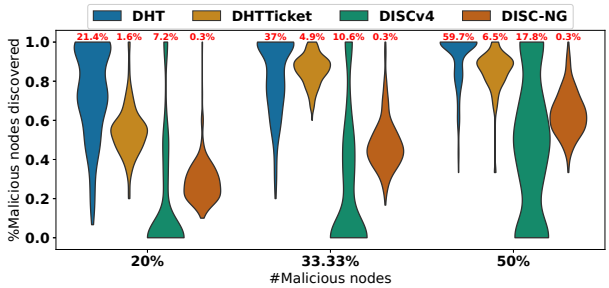


Figure 21: Lookup eclipse rate for a different number of Sybil nodes used in the attack.

DISCv4 performs worse than DISC-NG, reaching up to 17.8% eclipse rate. DISCv4 requires querying much more nodes to obtain the required number of advertisers and therefore has a higher chance of querying malicious nodes. DHT achieves the worst performance, reaching up to 59.7% eclipse rate. Thanks to our admission mechanism, DHTTicket outperforms both DISCv4 and DHT with a $< 6.5\%$ eclipse rate.

In Figure 22, we vary the size of the IP address pool used by the adversary. More IPs used by the attacker makes DISC-NG more likely to get eclipsed as the attacker IP score in the waiting time function is similar to values received by honest nodes. However, even for this worst-case scenario, DISC-NG achieves the eclipse rate $< 0.5\%$. DHT and DISCv4 suffer from significantly higher but remain unaffected by the increasing number of IPs used. Including the admission protocol, allows DHTTicket to outperform the other baseline protocol. However, the regular, DHT-based placement policy is more susceptible to eclipsing compared to DISC-NG.

9. Related Work

A decentralized service discovery system can be organized by directly storing the membership information in a DHT [39], [45], [46], [49]. DHT-based solutions offer fault-tolerant, scalable and efficient ways of finding nodes in large-scale networks. However, it is difficult to guarantee the availability of published service descriptions. If nodes close to a service hash fail, the whole subnetwork becomes undiscoverable. While solutions such as Chord4S [25] reduce this risk, the main drawback remains the vulnerability to Sybil attacks.

Other systems implement service discovery on top of publish-subscribe platforms. However, those solutions are built directly on top of a DHT [4], [9], [47] (and share its weaknesses), introduce high overhead to keep the data

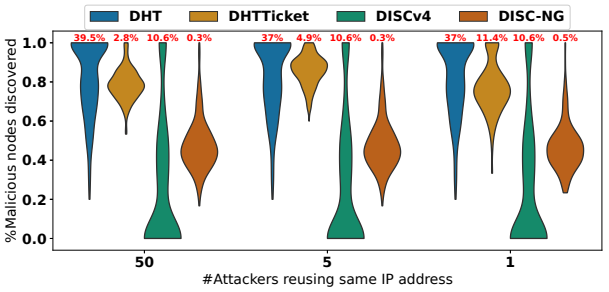


Figure 22: Lookup eclipse rate for a different number of IP addresses used by Sybil nodes.

up to date [52], introduce a single point of failure [12], or require all nodes to be correct (*i.e.* not byzantine) [3].

Recently, multiple service discovery protocols were implemented for the blockchain space [21], [30], [37]. Unfortunately, these solutions are meant to work in small-scale systems [21], or require writing to the blockchain (thus introducing significant monetary and/or computational cost) [30], [37].

Multiple works proposed DHT enhancements to make it more resistant to Sybil attacks. This can be achieved by exploiting social relations between participants operating the nodes [13], [14], introducing some kind of Proof-of-Work [5] or sampling participant identifiers [10]. All these solutions are difficult to implement in current P2P networks, and may have a negative impact on privacy. An extensive number of systems have been proposed for resilient peer sampling in P2P networks [8], [29], [41], [42]. While those systems are useful in some scenarios, they cannot be easily adapted to application-specific peer sampling required by the Ethereum ecosystem.

Relevant to our work, the Ethereum DHT was recently enhanced [27], [38] to make it more resistant to low-resource eclipse attacks at the DHT level. Those solution enable DISC-NG to operate, as it relies on honest participants not being fully eclipsed at the DHT level.

The enforcement of a waiting time for incoming requests to improve fairness and implement rate control has been used for preventing DDoS attacks towards centralized services or domains. In this context, the key interest is to avoid maintaining per-client state at a server while offering priority services to clients that have waited the longest, and to adjust waiting times based on per-domain traffic and congestion. Implementations of this idea include NetFence [36], Lazy Suzan [11] and the work of Kung *et al.* [33]. In contrast with DISC-NG, however, these approaches do not focus on content (topic ads) diversity, but only use enforced waiting for rate control.

10. Conclusions

On the foundational level, DISC-NG is the first practical, secure, and efficient service discovery protocol that can be deployed in large, real-world P2P networks. It combines the efficiency of traditional DHT operations with security inherited from pseudo-random ad placement. Our novel admission protocol, while performing only simple mathematical calculations, protects against a wide range of malicious behaviors, ensures equal load distribution, and promotes diversity in the network. DISC-NG is scheduled

for deployment in future versions of *devp2p*. An interesting future direction is to add Sybil identities detection mechanism [10] and automatically modify systems parameters to operate in a more secure, but more costly, mode (*e.g.* by decreasing the maximum number of ads retrieved from a single registrar).

References

[1] Mahdi Nasrullah Al-Ameen and Matthew Wright. Design and evaluation of Persea, a sybil-resistant DHT. In *9th ACM symposium on Information, computer and communications security*, ASIA CCS, 2014.

[2] Arbitrum. Secure scaling for Ethereum. https://arbitrum.io.

[3] Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. TERA: topic-based event routing for peer-to-peer architectures. In *Inaugural international conference on Distributed event-based systems*, DEBS, 2007.

[4] Ryohei Banno, Susumu Takeuchi, Michiharu Takemoto, Tetsuo Kawano, Takashi Kambayashi, and Masato Matsuo. Designing overlay networks for handling exhaust data in a distributed topic-based pub/sub architecture. *Journal of Information Processing*, 23(2):105–116, 2015.

[5] Ingmar Baumgart and Sebastian Mies. S/Kademlia: A practicable approach towards secure key-based routing. In *International Conference on Parallel and Distributed Systems*, ICPADS, pages 1–8, 2007.

[6] Binance. http://binance.com.

[7] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *14th ACM conference on Computer and communications security*, CCS, pages 92–102, 2007.

[8] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009.

[9] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[10] Thibault Cholez, Isabelle Chrisment, and Olivier Festor. Efficient DHT attack mitigation through peers' ID distribution. In *Workshops and Phd Forum of the IEEE International Symposium on Parallel & Distributed Processing*, IPDPSW. IEEE, 2010.

[11] Jon Crowcroft, Tim Deegan, Christian Kreibich, Richard Mortier, and Nicholas Weaver. Lazy susan: dumb waiting as proof of work. Technical report, University of Cambridge, Computer Laboratory, 2007.

[12] György Dán and Niklas Carlsson. Centralized and distributed protocols for tracker-based dynamic swarm management. *IEEE/ACM Transactions on Networking*, 21(1):297–310, 2012.

[13] George Danezis, Chris Lesniewski-Laas, M Frans Kaashoek, and Ross Anderson. Sybil-resistant DHT routing. In *European Symposium On Research In Computer Security*, ESORICS. Springer, 2005.

[14] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *Annual Network and Distributed System Security Symposium*, NDSS, 2009.

[15] Dan Dumitriu, E Knightly, Aleksandar Kuzmanovic, Ion Stoica, and Willy Zwaenepoel. Denial-of-service resilience in peer-to-peer file sharing systems. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):38–49, 2005.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[17] Ethereum. Devp2p: peer-to-peer networking protocols for ethereum clients. https://github.com/ethereum/devp2p/.

[18] Ethereum. Discv4. https://github.com/ethereum/devp2p/blob/master/discv4.md.

[19] Ethereum. Discv4 dns list. https://github.com/ethereum/discv4-dns-lists.

[20] Go Ethereum. Go ethereum: Official go implementation of the ethereum protocol. https://geth.ethereum.org/.

[21] Carson Farmer, Sander Pick, and Andrew Hill. Decentralized identifiers for peer-to-peer service discovery. In *IFIP Networking Conference*, 2021.

[22] Ethereum foundation. Nethermind: .NET Ethereum client. https://geth.ethereum.org/.

[23] Swarm Foundation. Swarm: distributed data storage and retrieval system. https://www.ethswarm.org.

[24] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EuroCrypt. Springer, 2015.

[25] Qiang He, Jun Yan, Yuanyuan Yang, Ryszard Kowalczyk, and Hai Jin. A decentralized service discovery approach on peer-to-peer network. *IEEE Transactions on Services Computing*, 6:1 – 1, 01 2013.

[26] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium*, 2015.

[27] Sebastian Henningsen, Daniel Teunis, Martin Florian, and Björn Scheuermann. Eclipsing ethereum peers with false friends. In *Workshops of the IEEE European Symposium on Security and Privacy*, EuroS&PW. IEEE, 2019.

[28] Hyperledger. Besu: Ethereum client for both public and private permissioned network use cases. https://geth.ethereum.org/.

[29] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8–es, 2007.

[30] Navin V Keizer, Onur Ascigil, Ioannis Psaras, and George Pavlou. Flock: Fast, lightweight, and scalable allocation for decentralized services on blockchain. In *International Conference on Blockchain and Cryptocurrency*, ICBC. IEEE, 2021.

[31] Lucianna Kiffer, Asad Salman, Dave Levin, Alan Mislove, and Cristina Nita-Rotaru. Under the hood of the Ethereum gossip protocol. In *International Conference on Financial Cryptography and Data Security*, FC, 2021.

[32] Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. Measuring ethereum network peers. In *Proceedings of the Internet Measurement Conference*, IMC, 2018.

[33] Yi-Hsuan Kung, Taeho Lee, Po-Ning Tseng, Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Soo Bum Lee, Yue-Hsun Lin, and Adrian Perrig. A practical system for guaranteed access in the presence of ddos attacks and flash crowds. In *23rd International Conference on Network Protocols*, ICNP. IEEE, 2015.

[34] Probe Lab. Final report: Nat hole punching measurement campaign. https://github.com/plprobelab/network-measurements/blob/master/results/rfm15-nat-hole-punching.md.

[35] Protocol Labs. A kademlia dht implementation on go-libp2p. https://github.com/libp2p/go-libp2p-kad-dht.

[36] Xin Liu, Xiaowei Yang, and Yong Xia. Netfence: preventing internet denial of service from inside out. *ACM SIGCOMM Computer Communication Review*, 40(4):255–266, 2010.

[37] Yacov Manevich, Artem Barger, and Yoav Tock. Endorsement in hyperledger fabric via service discovery. *IBM Journal of Research and Development*, 63(2/3), 2019.

[38] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. *IACR Cryptology ePrint Archive*, 2018(236), 2018.

[39] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*, IPTPS. Springer, 2002.

[40] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *9th International Conference on Peer-to-Peer*, P2P, 2009.

[41] Barlas Oğuz, Venkat Anantharam, and Ilkka Norros. Stable distributed P2P protocols based on random peer sampling. *IEEE/ACM Transactions on Networking*, 23(5):1444–1456, 2014.

[42] Matthieu Pigaglio, Joachim Bruneau-Queyreix, Yérom-David Bromberg, Davide Frey, Etienne Rivière, and Laurent Réveillère. RAPTEE: Leveraging trusted execution environments for byzantine-tolerant peer sampling services. In *42nd International Conference on Distributed Computing Systems*, ICDCS. IEEE, 2022.

[43] Protocol Labs. Filecoin: an open-source cloud storage marketplace, protocol, and incentive layer. https://filecoin.io.

[44] Benjamin J Puttnam, Ruben S Luís, Georg Rademacher, Yoshinari Awaji, and Hideaki Furukawa. 319 tb/s transmission over 3001 km with s, c and l band signals over $>$ 120nm bandwidth in 125 μm wide 4-core fiber. In *Optical fiber communications conference and exhibition*, OFC. IEEE, 2021.

[45] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, aug 2001.

[46] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware, pages 329–350, 2001.

[47] Vinay Setty, Maarten Van Steen, Roman Vitenberg, and Spyros Voulgaris. Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub. In *ACM/IFIP/USENIX 13th International Middleware Conference*, Middleware. Springer, 2012.

[48] A Singh, T-W Ngan, P Druschel, and DS Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *25th IEEE International Conference on Computer Communications*, INFOCOM.

[49] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[50] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. Design and evaluation of IPFS: a storage layer for the decentralized web. In *Annual conference of the ACM Special Interest Group on Data Communication*, SIGCOMM, 2022.

[51] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):1–49, 2011.

[52] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and ETH2.0 networks. *CoRR*, abs/2007.02754, 2020.

[53] Peng Wang, James Tyra, Eric Chan-Tin, Tyson Malchow, Denis Foo Kune, Nicholas Hopper, and Yongdae Kim. Attacking the Kad network. In *4th international conference on Security and privacy in communication networks*, SecureComm, 2008.

[54] Web3 Foundation. Polkadot interoperability network. https://polkadot.network.

[55] Mike Yi. A complete guide to violin plots. https://chartio.com/learn/charts/violin-plot-complete-guide/.

[56] zkSync Era. Scaling the Ethos and technology of Ethereum. https://arbitrum.io.

A. Notation

Table 1 summarizes the paper notation and provides the default values of the system parameters used by DISC-NG.

In consultation with developers of *devp2p*, we assume a cache capacity of $C = 1,000$ entries, the ad expiry time $E = 15$min and an average size of an advertisement equal to $1KB$. The security G and the exponent P_{occ} determine the possibility of the ad cache going above the capacity C. G should be set to a small value to limit its influence on the waiting time (where IP and service similarity scores should play the dominant role). P_{occ} should be large enough to prevent overflowing of the cache and small enough to enable usage of large portions of the cache under normal traffic conditions. We empirically set $G = 10^7$ and $P_{occ} = 10$. These values provide good protection against ad cache overflowing and good usage of cache space under normal conditions. We choose $K_{register}$ and $K_{lookup} = 5$ as a reasonable tradeoff between efficiency, load balance and security. We set $F_{lookup} = 30$, as the most common value used by current EGN applications. We choose $F_{return} = 10$ to ensure that the discoverers receive ads from at least 3 different registrars.

B. Security Proofs

This appendix completes Section 7 by providing the proofs of Theorem 1, Theorem 2, and Theorem 3.

B.1. Validity Proof (Theorem 1)

We prove Theorem 1 by showing that no registrar can be tricked into accepting an ad by an advertiser without a valid ticket.

Lemma 1 (Minimum waiting time)**.** *The minimum waiting time w_c associated with a valid ticket t (Definition 4) issued by a honest registrar r with a current cache size of c is*

$$w_c = \frac{GE}{(1 - c/C)^{P_{occ}}} \tag{3}$$

where G and P_{occ} are constant system parameters (see Table 1), E is the ad expiry time, c is the current size of the r's cache, and C is the maximum capacity of that cache.

Proof. The waiting time w_c is minimal when all the ads held by the registrar r are different from the one referenced by ticket t (*i.e.* $c(service) = 0$) and when these ads originated from a different IP address than t (*i.e.* $score(IP) = 0$). We find Equation (3) by inserting these values into Equation (1). □

Lemma 2 (Ticket requirement)**.** *No honest registrar r admits an ad x without a ticket.*

Proof. Let us assume an honest registrar r admits an ad x without any ticket. Ads admission only happens at Line 18 of Algorithm 4 if $t_{remaining} \leq 0$ (Line 17 of Algorithm 4). Upon attempting to admit x without any ticket (Line 5 of Algorithm 4), r sets $t_{remaining} \leftarrow t_{wait}$ (Line 6 of Algorithm 4); and Algorithm 4 does not modify $t_{remaining}$ later. However, Lemma 1 indicates $t_{wait} \geq 0$ (assuming $G \neq 0$ and $E \neq 0$, which we ensure axiomatically), which implies $t_{remaining} \geq 0$, hence a contradiction. □

Theorem 1 (Validity)**.** *No honest registrar r admits an ad x without a valid ticket (Definition 4) referencing x.*

Proof. Let us assume an honest registrar r admits an ad x without a valid ticket. Ads admission only happens at Line 18 of Algorithm 4 if $t_{remaining} \leq 0$ (Line 17 of Algorithm 4). There are then only two cases: (i) r admits x without any ticket, (ii) r admits x with an invalid ticket.

TABLE 1: Notation and default parameters

Participants	
$n_i \in N$	nodes
$n.\mathsf{IP}$	node's IP address
$n.\mathsf{id}$	node's ID
$n_i \in N^h$	honest nodes
$n_i \in N^m$	malicious nodes
$d_i \in D(s)$	discoverers for service s
$a_i \in A(s)$	advertisers for service s
$r_i \in R$	registrars
$s_i \in S$	services
Attributes	
c	number of ads in the ad cache (occupancy)
$c(s)$	number of ads for service s in the ad cache
$b_i(x) \in B(x)$	i bucket centred around key x, in routing table centred around x
$w(x)$	waiting time of for request x
$\bar{w}(x)$	average waiting time of for request x
$\mathsf{ticket}.t_{init}$	ticket creation time
$\mathsf{ticket}.t_{mod}$	ticket last modification time
$\mathsf{ticket}.t_{wait\_for}$	ticket waiting time
score(IP)	IP similarity score
key	key in the DHT

System parameters		**Default values**
C	capacity of the ad cache	1000
$K_{register}$	number of ads placed per bucket	3
K_{lookup}	parallel requests during lookup	5
E	expiry time (ad lifetime)	15min
F_{lookup}	number of peers to find	30
F_{return}	max number of service-specific peers returned from a single registrar	10
P_{occ}	occupancy exponent	10
G	safety parameter from the similarity score	10^{-7}
Δ	registration window	1s
X_{size}	average request size	1000 B
m	number of buckets advertise/search table	16

Simulation parameters	**Default values**
Simulation time	1 hour
Network size $\lvert N\rvert$	$25,000$
Number of services $\lvert S\rvert$	300
Service popularity Zipf distribution exponent	1.0
Percentage of attackers $\frac{\lvert N^m\rvert}{\lvert N\rvert}$	33%
Number of attackers reusing IP address domain $\frac{\lvert N^m\rvert}{\lvert n.ip \in N^m\rvert}$	5

Lemma 2 ensures x is not admitted without a valid ticket, hence we only need to consider case (ii). Line Algorithm 4 of Algorithm 4 ensures the algorithm aborts if the ticket does not contain a timestamp and a valid signature over x. There are thus no cases, where r admits x with an invalid ticket, hence a contradiction. □

B.2. Advertiser Liveness Proof (Theorem 2)

We argue that an honest advertiser can eventually register an ad with at least one honest registrar. Intuitively, honest advertisers receive tickets indicating a minimum waiting time before their ad can be registered. The cache of the registrars has a bounded capacity, but ads expire and are removed from the cache after a fixed amount of time. As a result, we argue that it is impractical for an adversary to overwhelm the cache of an honest registrar because doing so would necessitate meeting unmanageable bandwidth demands.

Lemma 3 (Registrar availability)**.** *An honest advertiser a eventually runs Algorithm 1 with an honest registrar r.*

Proof. This proof directly follows from Assumption 2. Advertiser a continuously looks for registrars through DHTs lookups (Section 4) and runs Algorithm 1 with every registrar it connects with; it thus eventually runs Algorithm 1 with an honest registrar r. □

Lemma 4 (Registrar's cache availability)**.** *The cache of an honest registrar r is never full.*

Proof. We prove this lemma by contradiction assuming that the cache maintained by registrar r reaches size $c = C - 1$ (where C is the maximum cache capacity). Equation (10) (Section C.1) can be re-written to express the total number of incoming requests $|X|$ as a function of the cache size c, the ads expiry time E, and the average waiting time $\bar{w}$ assigned to the incoming requests $|X|$:

$$|X| = c \times (1 + \frac{\bar{w}}{E}) \tag{4}$$

Lemma 1 shows that the minimum waiting time associated with a valid ticket is given by

$$w = \frac{G \times E}{(1 - c/C)^{P_{occ}}} \tag{5}$$

We derive the minimum number of incoming requests required to maintain the cache full with c requests by combining Equation (4) and Equation (5):

$$|X| = c \times (1 + \frac{G}{(1 - c/C)^{P_{occ}}}) \quad (6)$$

We compute the bandwidth required to maintain the cache $c = C - 1$ by denoting L the size of every request and substituting $c = C - 1$ in Equation (6):

$$B_{C-1} = \frac{L(C-1)}{E}(1 + \frac{G}{(1/C)^{P_{occ}}}) \quad (7)$$

This leads to $B_{C-1} = B_{max}$, which contradicts Assumption 3; hence $c \neq C - 1$. We conclude the proof by noting that the size c of the registrar's cache increases monotonically through the serialized processing of each incoming request; as a result, the size of the cache cannot reach $c = C$ without first reaching $c = C - 1$, which implies $c < C - 1 < C$. □

Lemma 5 (Finite waiting time)**.** *There exists a maximum waiting time w_{max} larger than every waiting time w computed by an honest registrar r; that is, $w \leq w_{max} \in \mathbb{N}$.*

Proof. Honest registrars configure $C \neq 0$ (Table 1), which we can ensure axiomatically. Let us assume $w \to \infty$. This implies $c = C$ (Equation (1)), which contradicts Lemma 4. □

Lemma 6 (Guaranteed advertise after waiting)**.** *An honest registrar r always admits a new ad x upon receiving a valid ticket t (Definition 4) old of at least w_{max} units of time (i.e. the maximum waiting time) within its scheduled window.*

Proof. Upon receiving ticket t, registrar r computes t_{wait} (Line 4 of Algorithm 4). Since t is not empty, r directly proceeds with the checks Lines 10, 12, and 14 of Algorithm 4. All these checks pass since t is valid (Definition 4), the ad x is new (i.e., not yet registered with r), and r receives t within its scheduled window. Since $t_{wait} < w_{max}$ (Lemma 5) and t is old of at least w_{max} units of time, the registrar computes $t_{remaining} < 0$ (Line 15 of Algorithm 4) and admits x (Lines 17 and 18 of Algorithm 4). □

Theorem 2 (Advertiser liveness)**.** *An honest advertiser a can eventually register an ad x with an honest registrar r.*

Proof. Lemma 3 ensures an honest advertiser a eventually runs Algorithm 1 with an honest registrar r. Advertiser a thus attempts to register the ad x with registrar r (Line 20 of Algorithm 1). Since registrar r is honest, it runs Algorithm 4 and always replies to a (Line 27 of Algorithm 4). Two cases are then possible: (i) r directly admits x (Line 18 of Algorithm 4) and confirms it to a (Line 22 of Algorithm 1), or (ii) r replies with a valid ticket (Definition 4) instructing a to wait for w units of time (Line 25 of Algorithm 1). Case (i) is straightforward, r registers x and the proof concludes; we are thus left with case (ii). Advertiser a continuously tries to register x using the registrar's ticket (Lines 19 and 20 of Algorithm 1). After GST and after (at most) w_{max} units of time (Assumption 1 ensures GST eventually arrives and Lemma 5 ensures w_{max} exists), a provides r with a ticket old of at least w_{max} units of time within its scheduled window. Lemma 6 then ensures r registers x. □

B.3. Discoverer Liveness Proof (Theorem 3)

We argue that an honest discoverer seeking a service eventually connects with an honest advertiser offering this service. Intuitively, an honest advertiser can eventually register an ad for its service with a set of honest registrars, and an honest discoverer eventually discovers these registrars and thus the advertiser offering the service.

Lemma 7 (Registrar discovery)**.** *An honest discoverer d eventually runs Algorithm 2 with an honest registrar r.*

Proof. This proof directly follows from Assumption 2. Discoverer d continuously looks for new registrars through DHTs lookups (Section 5) and runs Algorithm 2 with every registrar it connects with. Since d randomly samples the peers it connect with (Algorithm 3), it eventually runs Algorithm 2 with an honest registrar r. □

Lemma 8 (Lookup termination)**.** *Let us assume a set of honest registrars R advertises an ad for service s. An honest discoverer d running Algorithm 2 over the input s eventually connects with a registrar $r \in R$.*

Proof. We prove this lemma by induction through the serialized discovery of honest registrars throughout multiple runs of Algorithm 2. Lemma 7 ensures the recursion base. Let us assume discoverer d is connected with honest registrar r'; we show there exists at least one run of Algorithm 2 where it connects with different honest registrar r''. Discoverer d running Algorithm 2 visits buckets $b_0(s), b_1(s), \ldots$ and queries K_{lookup} registrars per bucket until it finds F_{lookup} peers (Line 17 of Algorithm 2). Let us assume that this is achieved after visiting k buckets. Discoverer d always selects random peers (Algorithm 3) but can be unlucky and only contact malicious registrars (malicious registrars only return malicious ads), and/or only receive malicious ads from honest registrars. The probability for this to happen for all t visited buckets is

$$p^{ecc} = \prod_{i=0}^{t-1} (p_i^{ecc})^{K_{lookup}} \quad (8)$$

where p_i^{ecc} is the probability to only receive malicious ads from a random queried registrar in bucket $b_i(s)$. Remember that discoverer d continuously (re-)runs Algorithm 2 over input s until it eventually connects with a registrar $r \in R$ (Section 5). As a result, we only need to show that there exists at least one run where $p^{ecc} < 1$. To find p_i^{ecc}, we look at the random registrar in bucket $b_i(s)$. Let p_i^m be the probability that the registrar is malicious and $1 - p_i^m$ the probability that it is honest. For the latter case, let p_i^{hm} be the probability that the honest registrar only returns malicious ads. The probability p_i^{ecc} to be eclipsed in that bucket is therefore

$$p_i^{ecc} = p_i^m + (1 - p_i^m) \cdot p_i^{hm} \quad (9)$$

Assumption 2 ensures there exists at least one run of Algorithm 2 where $(1 - p_i^m) > 0$ and $p_i^{hm} > 0$. As a result, discoverer d connects to registrar r''. □

Lemma 9 (Ad discovery)**.** *An honest discoverer d wishing to discover service s eventually connects with an honest registrar r advertising an ad for service s (registered by an honest advertiser a).*

Proof. After GST, Theorem 2 ensures advertiser a eventually registers an ad for service s with the set of registrars R. Lemma 7 ensures discoverer d eventually runs Algorithm 2 with a honest registrar r'. Lemma 8 then ensures d eventually discovers and connects with a registrar $r \in R$; and Assumption 1 ensures this connection eventually happens before the ad expires. As a result, discoverer d connects with a registrar $r \in R$ advertising an ad for service s. □

Theorem 3 (Discoverer liveness)**.** *An honest discoverer d eventually discovers service s advertised by an honest advertiser a with an honest registrar r.*

Proof. After GST, Lemma 9 ensures discoverer d eventually connects with an honest registrar r advertising an ad x for service s, registered by an honest advertiser a. Discoverer d thus learns the address of a and discovers service s. □

C. Extended Analysis

In this appendix, we provide more details on fairness in Section 7 and discuss additional mathematical analysis of our protocol.

C.1. Efficiency

C.1.1. Memory usage is bounded by the capacity of the ad cache. We focus exclusively on registrars, as advertisers and discoverers require a fixed amount of memory for their operations. The amount of ads in the cache is given by

$$c = \frac{|X| \times E}{E + \bar{w}(x)} \tag{10}$$

where $|X|$ is the number of requests constantly trying to get into the table, E is ad lifetime and $\bar{w}(x)$ is the average waiting time received by requests x. In the worst case scenario, when requests x can achieve 0 similarity score for both the service and the IP addresses, the waiting time formula is given by $\bar{w}(x) = G \times E/(1 - \frac{c}{C})^{P_{occ}}$.

The possibility of the cache going above the capacity is determined by G and P_{occ}. G should be set to a small value to limit its influence on the waiting time (where IP and service similarity scores should play the dominant role). P_{occ} should be large enough to prevent overflowing of the cache and small enough to enable usage of large portions of the cache under normal traffic conditions.

Lemma 10. *Ad cache never reaches its capacity C.*

Proof. Let us assume registrar r_j receiving registration requests $x_{ij} \in X_j$ The number of ads in the r's ad cache depends on the total number of incoming requests $|X_j|$ and the average waiting time they receive $\bar{w}(X_j)$. Let us assume that it is possible to reach $c = C$ ads in the ad cache. This would require maintaining $|X_j|$ high enough to reach $c = C - 1$ and then increasing it further, where $|X_j| = \frac{c(E+\bar{w}(X_j))}{E}$. In the worst case scenario, when all then requests $x_{ij} \in X_j$ achieve 0 similarity score for both the service and the IP addresses, the waiting time formula is given by: $\bar{w}(X_j) = \frac{G \times E}{(1-\frac{c}{C})^{P_{occ}}}$, so that $|X_j| = \frac{c(E+\frac{G \times E}{(1-\frac{c}{C})^{P_{occ}}})}{E} = c(1 + \frac{G}{(1-\frac{c}{C})^{P_{occ}}})$. Calculating $|X_j|$ necessary to maintain $c = C - 1$ gives us $|X_j| = (C-1)(1+\frac{G}{(1-\frac{C-1}{C})^{P_{occ}}})$. Substituting the protocol parameters, we receive $|X_j| \approx 10^{25}$.

Every request has to be re-sent to the registrar at least every E (Algorithm 4 line 22) and the size of a request is $X_{\mathsf{size}} = 1000$ B. Therefore, the size of the income traffic requires bandwidth $\mathsf{bandwidth} = 11^{25}$ bps. This value is much larger than the currently achievable speed record of $\approx 10^{14}$ bps [44]. Therefore, in practice, it is impossible to reach $c = C$ ads in the ad cache. □

In consultation with developers of Geth [20], we assume a cache capacity of $C = 1,000$ entries and an average size of an advertisement equal to 1KB. As discussed in Section 7 earlier, we choose $G = 10^{-7}$ and $P_{occ} = 10$; these values provide good protection against cache overflowing and a good usage of cache space under normal conditions.

Pending requests (*i.e.* not in the cache) do not create any state, apart from updating the lower bound, at the registrar (*i.e.* the registrar uniquely calculates the waiting time and returns a signed ticket). The lower bound state created by registrars is bounded by the number of distinct IPs and services in the cache and is thus bounded by its capacity n.

C.1.2. Register and lookup operations finish within $O(log(|N|))$ steps. As detailed in Section 4, registration of ads and service lookup operations allow learning peers from buckets associated with increasingly large prefixes to the service ID destination, guaranteeing these operations to finish within $O(log(|N|))$ steps.

C.2. Fairness

We assume a Zipf distribution of the service popularities in the system and that the service IDs are uniformly distributed in the DHT key space. For simplicity, we uniquely compare the load of registrar r_A – located close to the most popular service s_A and the load of registrar r_B – located close to the least popular service s_B.

We assume the IDs of s_A and s_B to be located on the opposite sides of the DHT key space, the worst-case scenario for load balancing. Both r_A and r_B receive different amounts of traffic for both s_A and s_B, and the same amount of traffic for other services, represented by a fictive service s_Y.

C.2.1. Registration operations achieve equal load distribution. As the closest node to the ID of s_A, r_A receives registration requests from all advertisers $A(s_A)$. As the furthest node from the ID of s_B, it also receives, on average, $\frac{|A(s_B)| \times K_{register}}{|N|/2}$ requests from advertisers for service s_B. Analogically, r_B receives requests from all advertisers $A(s_B)$ and $\frac{|A(s_A)| \times K_{register}}{|N|/2}$ requests for service s_A.

As $|A(s_A)| \gg |A(s_B)|$ (service s_A is much more popular than s_B), the initial number of requests is higher for registrar r_A. However, as its ad cache fills up, r_A will issue higher waiting times making the requests less frequent. Figure 9 presents the registration load ratio between both registrars as a function of increasing popularity between the two services. The load difference experiences sub-linear growth. When s_A is 100 times more popular, r_A receives only 1.6 times more requests than r_B. We also present results without the admission control (*i.e.* all the requests receive a fixed waiting time) for reference.

To obtain Figure 9, we extend Equation (10) by the service similarity score (still assuming complete IP address diversity). The average number of ads c_A and c_B for services s_A and s_B, respectively, in r_A's cache is

$$c_A = \frac{|A(s_A)|}{1 + (G + \frac{c_A}{c}) \cdot (1 - \frac{c}{C})^{-P_{occ}}} \quad (11)$$

$$c_B = \frac{\frac{|A(s_B)| \cdot K_{register}}{|N|/2}}{1 + (G + \frac{c_B}{c}) \cdot (1 - \frac{c}{C})^{-P_{occ}}} \quad (12)$$

$$c_Y = \frac{|A_Y|}{1 + (G + \frac{c_Y}{c}) \cdot (1 - \frac{c}{C})^{-P_{occ}}} \quad (13)$$

$$c = c_A + c_B + c_Y \quad (14)$$

where $|A_Y|$ is a "background load" representing the registration requests of the other services in the system. For Figure 9, we set $|A_Y|$=100. Again, the above system of non-linear equations must be solved numerically for c_A and c_B.

C.2.2. Lookup operations achieve equal load distribution. Let us assume again that registrar r_A is the closest node to the ID of service s_A. All the discoverers $D(s_A)$ looking for s_A will go towards this node during their lookup operations, so the number of requests is expected to grow as $|D(s_A)|$ grows. At the same time, the more advertisers s_A has, the more ads will be placed in other buckets further away from s_A and therefore r_A. Recall that discoverers stop their lookup operations after collecting F_{lookup} peers. We set $F_{lookup} = 30$, a value commonly used by applications in the Ethereum ecosystem. As the number of ads in the network grows, more discoverers are likely to stop before reaching r_A. Figure 10 presents r_A's load for increasing values of $|D(s_A)|$, assuming $|D(s_A)| = |A(s_A)|$ for simplicity. Depending on the $K_{register}$ and K_{lookup} parameters, the maximum load is experienced when the number of s_A-advertisers/discoverers is relatively small. It goes back to 0 when the service becomes popular in the network. We choose $K_{register}$ and $K_{lookup} = 5$ as a reasonable tradeoff between efficiency, load balance and security.

In the following, we describe how Figure 10 is obtained. As described in Section 4, a discoverer looking for service s_A will start at the furthest bucket and progress toward the closest registrar to that service's ID until it has received F_{lookup} responses. Let bucket 0 be the furthest bucket containing $|N|/2$ registrars, bucket 1 the closer bucket containing $|N|/4$ registrars etc. In each bucket i, the discoverer will query $\max(K_{lookup}, \frac{|N|}{2^{i+1}})$ registrars. We need to calculate the probability that a discoverer will reach the last bucket, i.e., the closest registrar to the service ID. In the following, we calculate the distribution $p_{0..t}(Resp = V)$ of the number of responses V that a discoverer will have received after traversing buckets 0 to t.

Let $p_{rr,i}(Resp = V)$ be the probability that the discoverer will receive V responses from a registrar rr in bucket i. We have

$$p_{rr,i}(Resp = V) = \sum_{g=0}^{|A(s_A)|} p(rr \text{ received } g \text{ registrations} \wedge rr \text{ has } \min(V, F_{return}) \text{ ads})$$

where F_{return} is the maximum number of responses a registrar will return. The probability that the registrar received g registrations is the probability that g out of the $|A(s_A)|$ advertisers chose the registrar. For bucket i, it is given by the binomial distribution:

$$p(rr \text{ received } g \text{ registrations}) = \binom{|A(s_A)|}{g} \left(\frac{K_{register}}{|N|/2^{i+1}}\right)^g \left(1 - \frac{K_{register}}{|N|/2^{i+1}}\right)^{|A(s_A)|-g}$$

Given g registrations, the number of ads that the registrar returns can be calculated using Equation (11) by substituting $|A(s_A)|$ with g. Combining both results, we can calculate the joint probability $p_{rr,i}(Resp = V)$.

Calculating the exact probability $p_i(Resp = V)$ that a discoverer obtains V responses in bucket i and the probability $p_{0..t}(Resp = V)$ to obtain V responses after visiting buckets 0 to t is numerically intensive. For Figure 10, we use Monte-Carlo simulation to approximate $p_{0..t}(Resp = V)$ from $p_{rr,i}(Resp = V)$. For each bucket $0 \leq i \leq t$, the simulation randomly draws K_{lookup} samples from the distribution $p_{rr,i}(Resp = V)$, in this way simulating the querying of K_{lookup} registrars per bucket. This approximation assumes that the distribution of responses for the individual registrars in a bucket are independent, which is mostly correct for large $|N|$. The simulation is repeated 100,000 times for the result shown in the figure. The probability to reach the last registrar is then $p_{0..u}(Resp < F_{lookup})$, where u is the number of buckets.

C.3. Security

C.3.1. DISC-NG achieves high resistance against DoS attacks. DISC-NG implements an admission control mechanism protecting the ad cache from being overwhelmed by an attacker with a limited number of IP addresses. Including a non-deterministic component in the ad placement mechanism reduces the efficiency of DoS attacks targeting a specific service or a part of the key space. Preventing honest nodes from using the system requires involving resources significantly surpassing the combined resources (see above) of the honest participants and incurs a preventive resource/monetary cost. Importantly, all the malicious ads are removed after ad lifetime E. An attacker thus has to constantly use their resource to perform the attack and the system quickly recovers once the attack stops.

C.3.2. DISC-NG achieves high resistance against eclipse attacks. We assume an attacker performing all the malicious activities listed in Section 3 using Sybil nodes. A lookup operation is considered eclipsed if all

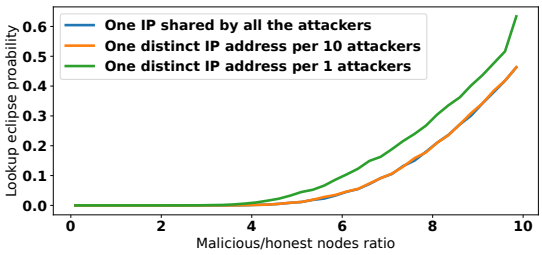


Figure 23: *Lookup eclipse probability*.

the peers received by the discoverer consist of malicious nodes. A discoverer can receive malicious peers from honest registrars (if malicious advertisers were able to place their ads) and malicious registrars (always returning the maximum amount of F_{return} malicious peers). The probability of being eclipsed by a random node in a bucket thus depends on the probability of encountering a malicious registrar (determined uniquely by the number of Sybil identities) and the probability of an honest registrar returning uniquely malicious peers (determined also by the number of IP addresses under the attacker's control). Figure 23 illustrates the probability of a lookup operation being eclipsed as a function of the increasing ratio between malicious and honest nodes. For all tested setups, the eclipse probability is close to 0 when the attacker uses less than 4 times the amount of honest nodes participating in a service[3]. An attacker can eclipse 60% of the lookups only when using 10 nodes per 1 honest node and providing a distinct IP address for all of them. However, such an attack would introduce a significant resource/monetary cost to the attacker.

In the following, we explain how the lookup eclipse probability shown in Figure 23 can be calculated. A honest discoverer looking for service s will visit buckets $b_0(s), b_1(s), \ldots$ and will query K_{lookup} registrars per bucket until it has found F_{lookup} peers. Let us assume that this is achieved after visiting t buckets. The discoverer can be unlucky and only contact malicious registrars (assuming that malicious registrars will only return malicious ads), and/or only receive malicious ads from honest registrars. The probability for this to happen for all t visited buckets is

$$p^{ecc} = \prod_{i=0}^{t-1} (p_i^{ecc})^{K_{lookup}} \tag{15}$$

where p_i^{ecc} is the probability to only receive malicious ads from a random queried registrar in bucket $b_i(s)$.

To find p_i^{ecc}, we look at the random registrar in bucket $b_i(s)$. Let p_i^m be the probability that the registrar is malicious and $1 - p_i^m$ the probability that it is not malicious. For the latter case, let p_i^{hm} be the probability that the honest registrar will only return malicious ads. The probability p_i^{ecc} to be eclipsed in that bucket is therefore

$$p_i^{ecc} = p_i^m + (1 - p_i^m) \cdot p_i^{hm} \tag{16}$$

The probability p_i^m that an individual registrar in bucket $b_i(s)$ is malicious depends on what source the discoverer used to add nodes to its search table for that bucket. If the discoverer itself is located in bucket $b_u(s)$, it can be expected that, on average, the buckets $b_0(s)$ to $b_u(s)$ are fully populated by the discoverer from its own routing table and that the routing table is also able to provide 17 nodes uniformly distributed over the key space for the remaining buckets. Since the buckets get smaller and smaller toward s, we can expect that we will get 8 nodes for bucket $b_{u+1}(s)$, 4 nodes for bucket $b_{u+2}(s)$, 2 nodes for bucket $b_{u+3}(s)$, and one node for bucket $b_{u+4}(s)$ from the routing table. The probability for each of these nodes to be malicious is $\frac{|N^m|}{|N|}$. However, this means that in order to achieve $K_{lookup} = 5$ lookups per bucket, the discoverer also has to resort to nodes returned by the queried registrars starting from bucket $b_{u+2}(s)$. If the queried registrar is honest, it will return nodes that are malicious with probability $\frac{|N^m|}{|N|}$. However, if the registrar is malicious, we can assume that all the nodes it returns are malicious, too. Consequently, the probability that a node in bucket $b_{u+2}(s), b_{u+3}(s), \ldots$ is malicious will be greater than $\frac{|N^m|}{|N|}$ and will increase with i.

To calculate p_i^{hm}, we need to know the distribution of honest and malicious ads in a honest registrar. If the registrar contains c^h honest and c^m malicious ads, the probability to randomly select only malicious ads from those $c^h + c^m$ ads is $\prod_{j=0}^{c^h+c^m} \frac{c^m - j}{c^h + c^m - j}$. Calculating the distribution of c^h and c^m analytically is difficult due to the complexity of the waiting time calculation. Instead, we approximate p_i^{hm} using the averages of c^h and c^m. On average, a registrar in bucket $b_i(s)$ will receive $q^h = \frac{|N^h| \cdot K_{register}}{|N|/2^{i+1}}$ honest and $q^m = \frac{|N^m| \cdot K_{register}^m}{|N|/2^{i+1}}$ malicious registration requests. The averages of c^h and c^m can then be obtained by solving

$$c^h = \frac{q^h}{1 + (G + \frac{c^h}{c} + score(IP)^h) \cdot (1 - \frac{c}{C})^{-P_{occ}}} \tag{17}$$

$$c^m = \frac{q^m}{1 + (G + \frac{c^m}{c} + score(IP)^m) \cdot (1 - \frac{c}{C})^{-P_{occ}}} \tag{18}$$

$$c = c^h + c^m \tag{19}$$

where the average IP scores for honest ads $score(IP)^h$ and malicious ads $score(IP)^m$ depends on the numbers of honest ads and malicious ads in the table and on the number of different IP addresses used by the advertisers. We show in Appendix C.3.3 how to calculate $score(IP)^h$ and $score(IP)^m$.

We solve equations (17)–(19) numerically to obtain p_i^{hm} and use Monte-Carlo simulation to calculate p_i^m. Figure 23 shows the resulting eclipse probability p^{ecc} for different scenarios and the default parameter values. For this figure, we assume $u = 2$, which is the average expected bucket for a random discoverer.

C.3.3. IP Score. We explain how to calculate averages for the IP scores. To simplify the understanding we will start with simple situations and complete them step by step. Furthermore, for the more complex situations, we only show the calculation where the number of entries in the cache with identical IP addresses is a power of 2. Note that n is used here for the number of random entries in the IP tree, as we will explain below in more detail.

3. For comparison, a regular DHT lookup operation can be eclipsed by using a fixed amount of 20 Sybil nodes (Section 8)

Situation 1: There are already n random addresses in the tree. What score will a new random IP address get?

We assume completely random addresses. That means it can happen with a certain probability that some of the random addresses are identical.

First level of the tree: With probability 0.5, the new address goes to the 0-branch and the probability that branch contains more than $\frac{n}{2}$ of the entries is $P(\#0 \geq \lfloor \frac{n}{2} \rfloor + 1)$. Also with probability 0.5, the new address goes to the 1-branch and the probability that branch contains more than $\frac{n}{2}$ of the entries is $P(\#1 \geq \lfloor \frac{n}{2} \rfloor + 1)$. Since the binomial distribution is symmetric, $P\left(\#0 \geq \lfloor \frac{n}{2} \rfloor + 1\right) = P(\#1 \geq \lfloor \frac{n}{2} \rfloor + 1)$. The average score for the first level is:

$$\text{NatScore}_{\text{level}}(n) = \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} \frac{\binom{n}{i}}{2^n} = 1 - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} \frac{\binom{n}{i}}{2^n}$$

We assume that the levels are independent and that we will have, on average, $\frac{n}{2^{i-1}}$ entries in each subtree of level i. Using the above equation on each level gives

$$\text{Score}_{\text{random}}(n) = \frac{1}{32} \sum_{i=1}^{32} \text{NatScore}_{\text{level}}\left(\frac{n}{2^{i-1}}\right)$$

Situation 2: There are alread $n + k$ addresses in the tree, however n are random and k are identical. What score will a new random IP address get?

We assume again completely random addresses. That means it can happen with a certain probability that one of the random addresses is identical to another random address or even to the k identical ones.

Let's first define the probability that at least q out of m fair coin tosses are head:

$$p(m, q) = \sum_{i=q}^{m} \frac{\binom{m}{i}}{2^m}$$

First level ("level 0") of the tree: With probability 0.5, the new IP address is in the same branch as the k identical addresses. The probability that more than half of the entries are in this branch knowing that the k identical are in this branch is $p\left(n, \lfloor \frac{n+k}{2} \rfloor + 1 - k\right)$.

With probability 0.5, the new IP address is in the branch that only contains random addresses. The probability to have the majority of the entries in this branch knowing that k are definitely not in this branch is $p\left(n, \lfloor \frac{n+k}{2} \rfloor + 1\right)$.

In total, the score for the first level is $\frac{1}{2} p\left(n, \lfloor \frac{n+k}{2} \rfloor + 1 - k\right) + \frac{1}{2} p\left(n, \lfloor \frac{n+k}{2} \rfloor + 1\right)$.

On the second level, we have four branches, of which one contains with certainty the k identical addresses, leading to the score $\frac{1}{4} p\left(\frac{n}{2}, \lfloor \frac{n+k}{4} \rfloor + 1 - k\right) + \frac{3}{4} p\left(\frac{n}{2}, \lfloor \frac{n+k}{4} \rfloor + 1\right)$.

And so on for the other levels. In total, the average score is:

$$\text{Score}_{\text{random}}(n, k) = \frac{1}{32} \sum_{i=1}^{32} \left[\frac{1}{2^i} p\left(\frac{n}{2^{i-1}}, \left\lfloor \frac{n+k}{2^i} \right\rfloor + 1 - k\right) + \left(1 - \frac{1}{2^i}\right) \cdot p\left(\frac{n}{2^{i-1}}, \left\lfloor \frac{n+k}{2^i} \right\rfloor + 1\right) \right]$$

Situation 3: There are already $n + k$ addresses in the tree, however n are random and k are identical (also random). What score will an IP address get that is identical to the k identical entries?

The situation is similar to situation 2. However, we know that we always stay on the branch with the k identical entries. The score is:

$$\text{Score}_{\text{identical}}(n, k) = \frac{1}{32} \sum_{i=1}^{32} p\left(\frac{n}{2^{i-1}}, \left\lfloor \frac{n+k}{2^i} \right\rfloor + 1 - k\right)$$

Situation 4: There are already $n + 2^f \cdot k$ addresses in the tree, however n are random and there are 2^f "groups" of k entries with identical addresses. Those 2^f addresses are distributed perfectly over the tree. What score will a new random IP address get?

Since the 2^f addresses are distributed perfectly over the tree, the new random address will see exactly $2^{f-1} \cdot k$ of them in its branch at the first level, $2^{f-2} \cdot k$ in its branch at the second level etc. After level a, the subtrees start to behave like in situation 2.

The score for the level $1 \leq j \leq f$ is $p\left(\frac{n}{2^{j-1}}, \left\lfloor \frac{n+2^f k}{2^j} \right\rfloor + 1 - 2^{f-j} k\right)$.

After level f, we can apply the score of situation 2 to each subtree:

$$score(IP)^h = \frac{1}{32} \sum_{j=1}^{f} p\left(\frac{n}{2^{j-1}}, \left\lfloor \frac{n+2^f k}{2^j} \right\rfloor + 1 - 2^{f-j} k\right) + \frac{1}{32} \sum_{i=f+1}^{32} \left[\frac{1}{2^{i-f}} p\left(\frac{n}{2^{i-1}}, \left\lfloor \frac{n+2^f k}{2^i} \right\rfloor + 1 - k\right) + \left(1 - \frac{1}{2^{i-f}}\right) \cdot p\left(\frac{n}{2^{i-1}}, \left\lfloor \frac{n+2^f k}{2^i} \right\rfloor + 1\right) \right]$$

Situation 5: Like situation 4 (i.e., $n+2^f \cdot k$ addresses), but the new address is not random; it is one of the 2^f addresses.

The score for levels 1 to f is the same as in situation 4. However, for the levels $> f$, the "new" address will always stay on a branch that contains the k entries with the same address. For those levels, we can use the result from situation 3.

$$score(IP)^m = \frac{1}{32} \sum_{j=1}^{f} p\left(\frac{n}{2^{j-1}}, \left\lfloor \frac{n+2^f k}{2^j} \right\rfloor + 1 - 2^{f-j} k\right) + \frac{1}{32} \sum_{i=f+1}^{32} p\left(\frac{n}{2^{i-1}}, \left\lfloor \frac{n+2^f k}{2^i} \right\rfloor + 1 - k\right)$$