DISCv5: Robust Service Discovery in the Ethereum Ecosystem

Anonymous submission #48

***Abstract*—The Ethereum ecosystem hosts a steadily increasing number of decentralized applications, including but not limited to the blockchain of the same name. Nodes joining the ecosystem connect to a global peer-to-peer network in which they have to find their application-specific peers. The peer discovery process is crucial for the efficiency and security of the entire ecosystem. Unfortunately, DISCv4, the current discovery protocol of the Ethereum global network scales poorly with increasing numbers of applications and nodes.**

We present DISCv5, a novel service discovery protocol for the Ethereum global network. DISCv5 combines pseudo-random and deterministic placement of advertisements to achieve both efficiency and security. It implements a new admission control protocol that protects against a wide range of malicious behaviors expected in an open environment. We implement DISCv5 in the official Go Ethereum client (Geth) and in a dedicated scalable simulator. In comparison with DISCv4, DISCv5 performs lookup operations using three orders of magnitude less messages, and discovers ten time more peers per time slot while achieving a similar or lower eclipse probability. DISCv5 is scheduled for deployment in future versions of the Ethereum platform.

1. Introduction

With an average of 23,500 constantly live nodes [1], the Ethereum ecosystem is one of the largest decentralized platforms currently in operation. While it is widely known for supporting the blockchain of the same name (also known as the *mainnet* and hosted by about 7,000 nodes), the platform is also home to a number of additional decentralized applications. 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 applications (*Whisper*). The platform already features almost 500 such applications, and their number grows every year [1]. The size distribution of the application-specific sub-networks varies significantly (Figure 1) featuring a *long tail*, with a vast majority of applications formed of a few hundred nodes or less.

All nodes in the Ethereum platform participate in a *global* peer-to-peer (P2P) network operating a distributed hash table (DHT) [29]. In addition to joining the global network, every node connects to at least one *sub-network* formed of peers participating in its application(s) of interest. In this paper, we focus on the *service discovery* mechanism, by which a node participating in the global P2P network discovers this application sub-network. 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.

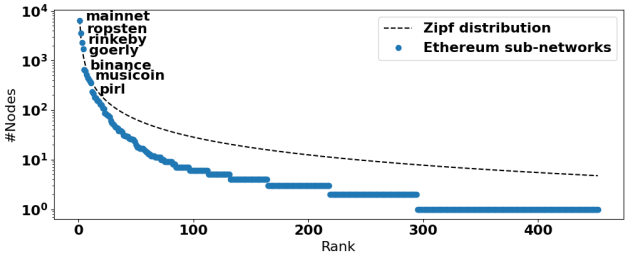

Figure 1: Distribution of the number of nodes of Ethereum's sub-networks in May 2022, corresponding to different applications and sorted by decreasing popularity. A Zipf distribution is given for reference.

Service discovery is a particularly sensitive mechanism in the Ethereum platform. It must ensure that malicious participants to this open network are unable to bias its execution against a victim node or sub-network–and that, despite the ability of these adversaries to operate multiple Sybil identities. Of particular importance is the protection against *eclipse* attacks, 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 against a specific application, preventing other nodes from discovering peers from the associated sub-network. On the other hand, the mechanism must remain efficient and scalable. It is not desirable, for reasons of scalability and robustness, that it relies solely on fixed (sets of) dedicated registrar nodes maintaining the membership of each application. Registrar nodes for popular applications could quickly become overwhelmed, and they would be an easy target for attackers. In addition, the need to provision dedicated registrar nodes would be a hindrance to the emergence of new applications and (initially) small sub-networks.

The current service discovery mechanism used in the Ethereum platform, named DISCv4 [2], leverages the global DHT for service discovery using a simple but robust *random walk* approach. A node willing to join an application's sub-network simply contacts individually a series of nodes collected from random lookups on the DHT, repeatedly checking application membership until it has collected enough peers participating in this target application. This approach offers good resilience to malicious behaviors but it suffers from very poor scalability and performance, in particular for small sub-networks.

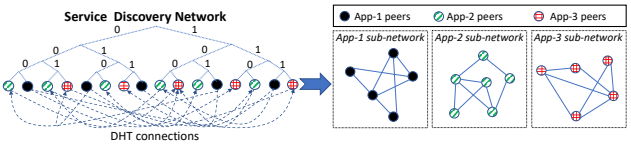

Figure 2: Formation of application-specific sub-networks using a universal service discovery network.

Random walks have, indeed, a decreasing likelihood of encountering peers from the target sub-network, increasing the number of messages required, join latency, and resource consumption at all the encountered peers. As more applications join the DHT, the inefficiency of the random discovery process becomes a bottleneck for the entire ecosystem. Alternative solutions for service discovery were proposed, but were meant for small-scale networks [19], [39], centralised [11], based on unrealistic assumptions [15], [16], or insecure [4], [5], [9], [9], [35].

Contributions. We present in this paper DISCv5, a novel service discovery mechanism for Ethereum. DISCv5 targets a balance between robustness, i.e., the ability to resist malicious behaviors and Sybil identities, efficiency, i.e., fast service discovery even for small applications, and good load-balance over participating nodes.

After providing background knowledge and reviewing the current service discovery mechanism of Ethereum in Section 2, and detailing our system and threat models in Section 3, we detail DISCv5 as follows.

In Section 4, we present how DISCv5 enables nodes, members of application sub-networks, to *advertise* their membership to these applications in the form of *service advertisements*. Any node can act as a *registrar* and store advertisements for any topic. In contrast with the direct use of DHT lookups, however, in DISCv5 service advertisements propagate to a pseudo-random subset of all nodes in the global network. The density of advertisements for an application increases as a service lookup approaches the key of a specific application in the DHT.

Robustness, load balancing, and efficiency for the discovery of smaller sub-networks all rely on a novel *admission protocol*, by which registrars accept or reject incoming service advertisements (Section 5). It ensures that advertisers cannot effectively flood advertisements at a registrar, even when deviating from the protocol or operating Sybils, and that less popular topics get a sufficiently high probability to be represented and found.

At the core of our admission protocol lies the need for advertisers to respect a waiting time imposed by the registrars before being able to successfully register their advertisement. We discuss the design and dynamics of the waiting function in Section 6. The function allows limiting the amount of resources used by each node, promotes diversity of topic advertisements stored by each node, and protects against a vast range of malicious behaviors.

We summarize the results of the analysis of DISCv5 security in Section 7, and provide more thorough development of this analysis in Appendix A.

We overcome multiple practical challenges, implement DISCv5 in the code base of the *Go Ethereum* (Geth) client [3] as well as in a simulator (Section 8). Compared to DISCv4, DISCv5 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 load on the most busy node in the network by two orders of magnitude. DISCv5 is scheduled for deployment in future versions of the Ethereum platform.

We finally review related work in Section 9 and draw conclusions in Section 10.

2. Background

We start by detailing the operation of Ethereum's global DHT, which is an evolution of the canonical Kademlia [29]. Then, we present DISCv4, the current service discovery mechanism of Ethereum operating over this DHT, and discuss its shortcomings.

2.1. The Ethereum global DHT

All nodes in the Ethereum ecosystem 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* [34], [36]. Ethereum uses an evolution of Kademlia [29], a robust and mature DHT design. Every node in the overlay is assigned a unique identifier (*node ID*), drawn from the same key space as items, i.e., information stored in the DHT. The Kademlia DHT assigns a specific key i to the k *closest* (according to a distance metric) nodes to i in that space, where k is a system parameter. The distance d between two keys x and y is a logarithm of their bitwise exclusive or (XOR) interpreted as an integer, *i.e.*, $d(x, y) = log_2(x \oplus y)$ (*i.e.*, the length of the differing suffix in bits).

Each node X in the overlay maintains a *routing table* storing a set of peers, each with its *node ID* and reachability information, *i.e.*, IP address and port numbers. The routing table is partitioned into m *buckets*, numbered zero through $m - 1$, where m is a global system parameter. Bucket i contains a list of up to k peers, whose IDs share a common prefix of length i with X.

The bucket partitioning scheme divides the key space from the point of view of X into disjoint intervals, halving in length every time the bucket's associated prefix includes one more common bit with X's ID. As a result, a node's routing table provides a more detailed (*i.e.*, fine-grained) view of the subset of 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, in addition, 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.

Lookups for a key rely on α concurrent *routing* operations, with the goal of identifying the k closest nodes to that key. Original DHT designs [29], [34], [36] would use a recursive routing mechanism and communicate the target key to each encountered node. This poses a risk, as these intermediate nodes could leverage the target information to establish Sybils in proximity of the target key and return these to the querier. A security measure in Ethereum's Kademlia implementation is, therefore, to use an iterative (i.e., querier-controlled) routing process and hide the precise targeted key. The querier retrieves *entire* buckets from intermediate nodes, and filters them locally before continuing the process. Other counter measures integrated following researchers assessments of Kademlia's security in Ethereum [20], [28] include for instance limiting the number of nodes from the same /24 subnet in the same bucket or routing table, or strengthening the seeding process by which a routing table is initialized.

2.2. DISCv4 service discovery

In DISCv4, Ethereum's current service discovery protocol, 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 some overhead to both endpoints. If the initiator node discovers during the handshake that the encountered peer is part of the target application's subnetwork, it establishes an application-level connection. The objective of a node is to fill up its *application-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 their 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 subnetwork are wasteful and incur unnecessary overhead. More importantly, for applications with low popularity, the number of handshakes can be excessive, *i.e.*, on the order of thousands when 0.1% of the nodes are running the target application. Because all the applications are initially unpopular, the upfront cost of building a subnetwork can be large and finding peers can take a long time.

3. System and threat models

In this section, we present our network and threat models as well as the target properties of DISCv5.

3.1. System model

We assume a network of N nodes.[1] At startup time, each node generates a public/secret key pair, which it uses to secure point-to-point communication with its peers. Nodes are identified by their *node ID*, which is simply the hash of their public key. Multiple nodes may share the same IP address (due to NAT or being hosted by the same physical machine) [28]. However, two nodes cannot share the same ID.

As for DISCv4, DISCv5 leverages the existing Ethereum DHT, but does not rely on its key lookup operations. DISCv5 indexes different applications in the Ethereum eco-system as *topics*. A topic is an identifier for an application, or *service*, provided by a node. Topics are arbitrary strings that hash to a specific key.

All DHT nodes fulfill the following roles:

- A **registrar** accepts advertisements (*ads*) and responds to topic queries. When asked for a specific topic, a registrar responds with nodes that registered ads for this topic. A registrar may accept and store ads for any topic.
- An **advertiser** registers for a specific service topic and wants to be discovered by its peers. Advertisers make themselves discoverable by placing ads for that topic. Nodes are advertisers for every service they participate in.
- A **searcher** attempts to discover nodes registered under a specific topic, using service lookup operations.

A node providing a certain service topic is said to *register* itself when it submits an ad to a registrar. Depending on the needs of the application, a node can advertise multiple topics or no topics at all. We assume that the popularity of topics in the system is highly heterogeneous, i.e., it can follow a power law distribution [23]. Anyone can participate in registering and searching for (one or more) topics and use the same ID and IP for all its topics.

3.2. Threat Model

DISCv5 is designed to operate in an open, adversarial environment. We assume the presence of malicious actors in the network that may arbitrarily deviate from the protocol and coordinate their actions. Malicious actors can spawn multiple virtual nodes within one physical machine, and operate multiple physical machines. As a result, they can control a number of Sybil node IDs.

The security of the service discovery mechanism is fundamentally dependent on the security guarantees provided by the underlying DHT implementation, and in particular on its ability to avoid eclipse attacks against a specific node. Indeed, even if DISCv5 does not rely on DHT lookup operations, information in the DHT routing table is used to initialize and maintain the data structures used by the service discovery layer. We assume, therefore, that no honest node is fully eclipsed by the malicious ones, *i.e.*, each honest DHT node has *at least* one honest peer. As previously mentioned, Ethereum already implements multiple mechanisms preventing eclipse attacks at the DHT level [20], [28].

Maintaining nodes in the DHT requires infrastructure resources (public IP addresses) and we assume that the resources under the control of an attacker are limited. Specifically, we assume that it is easier for an attacker to generate similar IP addresses (*i.e.* within a single subnet) than it is to control many diverse IP addresses (with different prefixes). We use the number of IP addresses and IDs under the control of an attacker as parameters for our evaluation in Section 8.

Through a literature review [10], [20] we collected a list of malicious behaviors that an attacker can use to disrupt a DHT-based service discovery protocol:

- **Malicious DHT Peer**: DHT routing relies on asking peers for other nodes, closer to a specific target. When responding to such requests, an attacker only returns other malicious nodes, in order to hijack the process of DHT traversal by honest nodes.
- **Malicious Registrar**: When queried for a specific topic, an attacker returns a maximum amount of malicious nodes.[2]

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

2. Alternatively, a malicious registrar could simply refuse to respond. However, such behavior is less effective than returning malicious peers.

- **Spamming Advertisers**: An attacker bombards honest registrars with malicious ads. The attack can target single or multiple topics. The attack may cause an honest registrar to refuse honest ads due to lack of resources (*e.g.*, storage and CPU power). This registrar will, furthermore, later return maliciously-inserted ads of the spammer when queried by honest searchers.
- **Generic Spammers**: An attacker bombards an honest registrar with topic queries or random traffic hoping to exhaust bandwidth or CPU power.

Adversaries can strategically target specific nodes or regions in the DHT key space (*i.e.* by generating Sybil Node IDs within the desired region) when implementing these attacks. We assume the presence of attackers that can combine some or all of the behaviors above, and coordinate their actions to maximize their effect.

3.3. Target Properties

Under the considered threat model, DISCv5 achieves the following properties.

Efficiency. DISCv5 ensures that the number of nodes contacted and the number of messages exchanged increase logarithmically with the number of system participants, 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.

Fairness. DISCv5 ensures a balanced load distribution across systems participants and regions of the key space (*i.e.*, it avoids *hotspots*). The system provides efficient lookup and registration operations for all participants regardless of the topic they look up or register for. Each advertiser has a similar probability of being discovered by its peers.

Security. DISCv5 focuses specifically on preventing *Denial of Service* (nodes are unable/slow to discover their application-specific peers) and *Eclipse* (nodes discover uniquely malicious nodes) attacks. While completely eliminating such attacks in an open network is technically impossible, DISCv5 provides high probabilistic security and increases the monetary cost of committing them.

4. Placement of advertisements

In the following, we detail the distribution of advertisements in the network and discuss the process of searching for, and establishing application-specific connections. We start by discussing related challenges and explaining the data structures used by DISCv5.

4.1. Challenges

A first challenge in designing a robust service discovery mechanisms is to decide on the placement of advertisements (ads). In other words, the mechanism must decide which registrars in the network should be responsible for storing ads for each topic.

A first possibility would be to deterministically choose a group of nodes based on the topic itself, using DHT lookups. Such a solution makes it *efficient* to place and retrieve ads, as both advertisers and searchers know how to reach registrars within a logarithmic amount of steps. Unfortunately, such an approach causes *unequal* load distribution across registrars, especially when the popularity of topics varies significantly. Registrars storing popular topics (*i.e.* DHT nodes that are the closest to the hash of the popular topics) would, indeed, 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 topic.

Alternatively, advertisers could place their ads on random registrars across the entire network. This approach, as DISCv4, is *secure* and difficult to attack as an attacker would need to take control over the entire network to control a single topic. Furthermore, random placement is *fair* and achieves good load balance across registrars regardless of the topic popularity distribution. However, and similarly again to DISCv4, random placement is not *efficient*, as it makes it difficult for searchers to find placed ads, especially for unpopular topics.

DISCv5 implements an alternative method of placing ads, that combines both the deterministic and pseudo-random approaches. Advertisers start the registration process from nodes located far away from the topic hash, and traverse the DHT towards nodes close to the topic hash. On their way, advertisers place ads on encountered registrars. This way, as a registration process gets closer to the topic hash, the higher the density of ads gets. The searchers mimic the process and ask encountered registrars for topic-specific ads. The lookup process stops when enough ads are found.

4.2. Data structures

DISCv5 uses two types of tables to handle topic-specific registrations and service lookups, the *advertise* and *search* tables. The structure of these tables is similar but their use and lifecycles differ.

Common tables structure. Both tables are organized as a collection of *buckets* similarly to the Kademlia routing table. Buckets contain peers sharing a common prefix, and can be sorted by these prefixes (*e.g.*, peers with ID starting with 0, 1, 2, and so on). Unlike the Kademlia routing table, however, advertise and search tables are centered on a target topic ID rather than current node's ID, and prefixes represent different distances from this topic ID.

Advertise Table. An advertiser maintains an *advertise table* for each of the topics it belongs to and advertises. These tables exist for the whole duration of the participation to the corresponding service. The table keeps track of ongoing registration attempts with different registrars. Each bucket aims to contain *at least* $K_{register}$ potential registrars (Figure 3). Peers above the number of $K_{register}$ are kept as backup peers.

Search Table. A specific search table is created for every new service discovery request. This table is used to discover a number of ads through a iterative and parallel exploration process, after which the table is discarded.

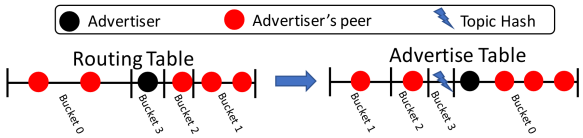

Figure 3: Creation of an advertise table from a routing table.

Construction and update of the tables. Both tables are bootstrapped upon their creation (*i.e.*, joining a new application for an advertise table and initiating a new discovery process for a search table) using entries from the local node routing table, for a specific topic ID. As the routing table is centered on the peer ID and not on this topic ID, the density of peers around the latter might be low or even null, particularly when two IDs are distant in the key space. The buckets are filled opportunistically while interacting with peers during the advertisement, resp. lookup, processes. There is, therefore, no specific DHT lookup for filling in the entries of these tables.

A local node n communicating with peer p in bucket b_i of a table T asks p to return peers from p's routing table that match the prefixes of all buckets $b_j \,|\, j \geq i$ in T. For instance, when communicating with peer p in bucket b_5 of a search table T, n may receive candidate peers for buckets b_5, b_6, and b_8.

The collection of peers in the tables implements a compromise between the risk of having malicious contacted peers pollute the table or a specific bucket with a majority of malicious peers, and the need to learn "rare" peers in buckets close to the topic ID. To limit this risk, local node n asks for a *single* peer for each bucket $b_j | j \geq i$ from a given contact p. Contacting nodes in buckets of increasing prefix length divides the search space by a constant factor, and allows learning new peers from more densely-populated routing tables towards the destination. The opportunistic table update procedure allows, when necessary, to reach the closest nodes to the topic ID within $O(\log(N))$ communication steps.

4.3. Distributing ads across registrars

An advertiser associating itself with a topic first creates a topic-specific advertise table as detailed above. The objective of the ad placement process is to continuously maintain up to $K_{register}$ active (*i.e.*, unexpired) registrations or ongoing registration attempts in every bucket of that table. Buckets located close to the topic hash cover less hash space than buckets located further away and, in turn, contain fewer than $K_{register}$ potential registrars; in this case an attempt of registering an ad is made with all of them.

Placing a fixed amount of ads per bucket make registrars close to a topic hash more likely to receive registrations for that specific topic. Increasing $K_{register}$ makes the advertiser easier to find at the cost of increased communication and storage costs.

The advertiser keeps track of pending or completed attempts to perform registrations in every bucket. A registration may be unsuccessful if the selected registrar is down or refuses to store the ad. In this case, the corresponding peer is removed from the bucket and a backup peer selected as a replacement, if one is available. A successful registration follows an admission procedure that we will detail in Section 5 and places an ad on an advertiser for a fixed amount of time a.

4.4. Looking up services

A service lookup in DISCv5 aims at identifying N_{lookup} advertisers for a specific topic ID and return them to the application. The service lookup is a process that is distinct from the underlying DHT's lookup protocol. A service lookup starts with the creation of an ephemeral, topic-specific *search table* as described in Section 4.2.

A searcher performing a service lookup progresses through the table starting from the farthest bucket (*i.e.*, peers whose ID has the *smallest* common prefix with the topic ID). The search process progresses using maximum K_{lookup} parallel requests, and issuing requests towards up to K_{lookup} peers per bucket. For instance, the searcher may start by querying K_{lookup} peers from bucket b_0. When one of these query returns, the searcher queries a peer from bucket b_1, and so on. A node that has been already queried is marked as such in the table, and is not queried again.

Queries return new candidates registrars for the search table, as explained in Section 4.2, as well as N_{return} topic-specific advertisers with a valid registered ad. A successful search stops when at least N_{lookup} different peers have been collected.[3] The search is unsuccessful when either no unmarked nodes remain in any of the buckets, or when the two last queries did not allow discovering new, unused registrar nodes through opportunistic discovery. In this case, the whole process restarts.

Analysis. The first bucket (b_0) covers the largest fraction of the key space as it corresponds to peers with no common prefix to the topic ID (*i.e.*, it matches 50% of all the registrars). For an attacker, placing malicious registrars in this fraction of the key space in order to eclipse a service discovery process would require considerable resources. Subsequent buckets cover smaller (halved for each increased prefix length) fractions of the key space, increasing the attacker opportunities to place Sybils but also increasing the success rate of the discovery process.

Parameters N_{return} and N_{lookup} play an important role in the compromise between security and efficiency. A small value of $N_{return} << N_{lookup}$ increases the *diversity* of the source of ads received by the searcher but increases search time, and requires reaching buckets covering smaller key ranges where eclipse risks are higher. On the other hand, values of a similar order of magnitude for N_{lookup} and N_{return} reduce overheads but increase the danger of a searcher receiving ads uniquely from malicious nodes. Finally, low values of N_{lookup} stop the search operation early, before reaching registrars close to the topic hash, contributing to a more equal load spread. We discuss the selection of optimal values for all of the protocol parameters in Section 7.

5. Admission Protocol

We now describe the registration procedure followed by an advertiser when attempting to register an advertisement (ad) at a specific registrar.

3. In case more than N_{lookup} peers were found, a random subset of N_{lookup} peers is drawn.

5.1. 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 Most recently used (MRU) exposes the system to an attacker that bombards registrars with dummy ads and evicts honest ones.

DISCv5 solves this problem by using a lightweight *waiting-time-based admission mechanism* for access control. When an advertiser sends an ad placement request to a registrar, the registrar will calculate the amount of time the advertiser needs to wait before being admitted. This *waiting time* is calculated based on the diversity of the request (*i.e.* how different is the request from ads already in the registry) and the space left in the registry. We explain the waiting time calculations in Section 6. The advertiser must wait for a waiting time reported by the registrar before it is allowed to contact the registrar again and have its ad actually admitted.

5.2. Data Structures

Advertisement. When advertisers send a registration request, they send an *advertisement*. The ad contains the IP address of the advertiser, the ID of the advertiser, the topic the ad is for and additional information needed to later contact the advertiser (*e.g.* an application-specific port number). In the remaining part of the paper, we omit the additional information for brevity.

Ad cache. Registrars store received ads locally in a data structure called an *ad cache*. Each ad stored in the ad cache has an associated lifetime a, after which the ad is automatically removed. The total size of the ad cache is limited by n. DISCv5 does not impose topic/IP/ID-specific limits on the content of the *ad cache* to accommodate for diverse network conditions and application popularity distributions. A single advertiser may place at most one ad for a specific topic in the ad cache. Registration requests for ads already in the cache are ignored. An advertiser may, however, attempt to place ads for multiple topics at the same registrar.

Ticket. Tickets are immutable objects issued by registrars to advertisers when receiving a registration request. They are digitally signed by the issuing registrar. Each ticket contains:

- Ad - a copy of a registration request (as described above).
- Initial timestamp - the local time at the registrar when the ad was received for the first time, t_{init}
- Waiting time - the waiting time calculated by the advertiser for the ad, $t_{waiting}$.

5.3. Registration Procedure

An advertiser willing to register an ad at a registrar starts by sending an initial request uniquely containing the ad itself. Based on the content of the ad cache and the incoming registration, the registrar calculates an ad-specific

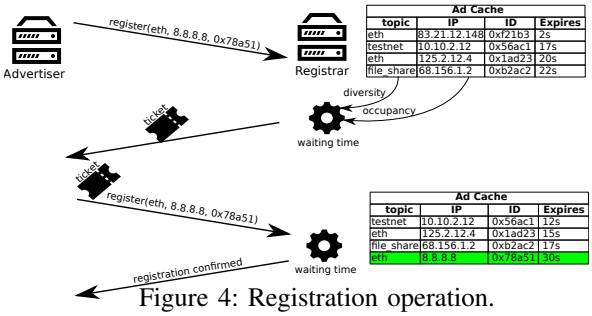

Figure 4: Registration operation.

waiting time (as detailed in Section 6). The advertiser then issues a ticket including the calculated waiting time and the time of receiving the initial request and sends it back to the advertiser.

The advertiser waits for the indicated time and attempts to register again. The consecutive registration request must include the last ticket issued by the registrar. Tickets can be used uniquely during a registration window:

$$t_{window} = [t_{init} + t_{waiting}, t_{init} + t_{waiting} + \delta t_{window}] \quad (1)$$

All registration requests outside the registration window are ignored by the advertiser. The registrar calculates the registration windows based on the content of the ticket. δt_{window} should be chosen to accommodate for the maximum delay between the advertiser and the registrar. The mechanism prevents an attacker from gathering many tickets, accumulating long waiting times (see below) and submitting the tickets all at once to overwhelm the registrar.

The advertiser calculates a new waiting time, based on the current content of the ad cache, every time it receives a registration request (with or without a ticket). The waiting time in the ticket is used only to calculate the registration windows and prevent advertisers from trying to register continuously. The ticket also allows the calculation of an *accumulated* waiting time:

$$t_{cumulative} = now - t_{init} \quad (2)$$

An ad is admitted if, and only if, the accumulated waiting time is equal to or larger than the calculated waiting time $t_{cumulative} \geq w$[4]. For registration requests without a ticket $t_{cumulative} = 0$. An advertiser that misses its registration window (as specified by the most recent ticket), loses all its cumulative waiting time and must attempt to re-register without a ticket. Once an ad is admitted, the registrar confirms the registration to the advertiser.

If the cumulative time is not sufficient, $t_{accumulated} < t_{waiting}$, the registrar issues a new ticket and the advertiser repeats the whole procedure. With consecutive registration attempts, advertisers increase their cumulative waiting time $t_{cumulative}$ and eventually will be admitted.

The inclusion of issue-time allows the registrars to prioritize advertisers that have been waiting for the longest time, as we explain later. The tickets are immutable (*i.e.*, tampering with the ticket is detectable by the registrars that originally issued the ticket thanks to the digital signature). When a registrar issues a new ticket (in case registration is not immediately successful) to an advertiser,

4. Note that a registration request without a ticket may be admitted directly when the calculated waiting time equals $w = 0$.

the registrar simply copies the issue-time from the last issued ticket and uses that as the issue-time of the new ticket. This means that the registrars are not required to maintain any state for each ongoing ticket request given that they can verify the authenticity of the ticket in the incoming registration requests.

An advertiser gives up and stops the registration process with a registrar when it has received r unsuccessful registration attempts (i.e., after being issued r tickets without being admitted). In this case, the advertiser removes the registrar from its advertise table and selects a new node located in the same bucket and attempts a new registration procedure. The mechanism prevents malicious registrars from stalling honest advertisers. Similarly, after the expiration of a previously placed ad, the registrar is removed from the advertise table and the process is restarted with a new node picked from the local route table.

6. Waiting Time

The waiting time function is used to calculate the total 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. It also protects against attacks, where a malicious actor tries to dominate the ad cache and exhaust the resources of a registrar.

Each request is given a waiting time based on the IP address of the registrar, the topic of the request, and the current state of the ad cache. The waiting time function is divided into two parts: an *occupancy score* (ranging from 0 to ∞) and a *similarity score* (ranging from 0 to 2) and is normalized by the amount of time each ad spent in the cache a (*i.e. ad lifetime*). a determines the absolute values of the returned waiting time. The final result is a product of all three: $w = a \times$ *occupancy score* $\times$ *similarity score*.

The *occupancy score* is based uniquely on the number of ads already in the cache. Its role is to progressively increase the waiting time as the ad cache fills up and to limit the memory used by a registrar. The *occupancy score* is defined as $\frac{1}{(1-\frac{d}{n})^{P_{occ}}}$ where d is the number of ads in the cache, n is the capacity of the cache, and P_{occ} is a protocol parameter. When the number of ads in the cache is low ($d \ll n$), 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 a current occupancy d close to the capacity of the cache n, the *occupancy score* goes to infinity thus limiting the number of admitted requests. We analyze the behavior of the waiting function and choose optimal system parameter values in Section 7.

The role of the *similarity score* is to determine how similar the incoming request is to the ads already in the ad cache in terms of IP address and topic. Requests significantly different from the current content of the cache receive lower similarity scores resulting in lower overall waiting times. Such an approach promotes fairness across topics (*i.e.*, it is easier for less popular topics to get into the cache) and protects against attempts to fill the ad cache by a small number of advertisers (as identified by their IP addresses). The similarity score is defined as the sum of the similarity score for the IP, the similarity score for the topic of the request, and a system parameter b: *similarity* $= b +$ *similarity(topic)* $+$ *similarity(IP)*.

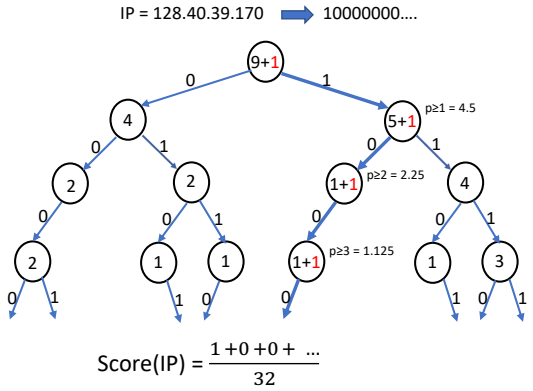

Figure 5: Inserting an IP address into the IP *tree* structure.

b ensures that the waiting time never reaches 0 even when requests get 0 values for IP and topic similarity score. Together with P_{occ}, it shapes the behavior of the waiting functions. We choose values for those parameters in Section 7.

The similarity score for topics is given by *similarity(topic)* $= \frac{d(topic)}{d}$. Here, $d(topic)$ is the number of ads for the specified topic already in the cache and d is the total number of ads in the cache. The score goes to 1 as the specified topic dominates the cache $d(topic) \approx d$.

A simple similarity score used for topics cannot be securely applied to IP addresses. An attacker may be able to generate a large number of different addresses sharing the same prefix (*e.g.* using a single /24 IPv4 network) that, while similar, would receive low *similarity scores*. A common solution (*e.g.*, as adopted by the Go Ethereum client [3], [28]) is to limit the number of IP addresses coming from the same (*e.g.* /24 IPv4 address) network. However, it is impossible to reliably set those limits without knowledge about the network size or NAT configuration of the honest nodes. Instead, we propose a more versatile approach that directly captures the similarity level across different IPs and translates it into a numerical score.

We introduce a binary *tree*, as shown on Figure 5, that stores IP addresses used in the existing registrations in the ad cache. Each vertex stores a counter, while the edges represent consecutive 0s or 1s in a binary representation of IP addresses. For simplicity, we present the *tree* for IPv4 addresses but its adaptation for IPv6 is straightforward.

Apart from its root, the *tree* consists of 32 levels (33 levels in total) representing bits in the binary representation of IPv4 IP addresses. The root level is depicted as level 0, the level of its successor as level 1 and so on. The counter of every *tree* node is initially set to 0. When adding an IP to the *tree*, the address is first converted to its binary representation and follows a path in the *tree* corresponding to consecutive bits. Counters of all the visited nodes are increased by 1. Removing an IP from the *tree* follows the analogical procedure but decreases all the counters on the path. 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 IP addresses starting with 0, root's 1 successor stores the number of the IP

addresses starting with 1 and so on.

After each addition of an address to the *tree*, an IP score is calculated. The score is a sum of *penalty points* obtained on visited nodes.

$$score(IP) = \sum_{i=1}^{32} penalty(p_{\geq i})$$

where $p_{\geq i}$ is the number of IP addresses in the cache sharing a prefix with IP with a length of at least i. A penalty point is given at $p_{\geq i}$ if the IP address to be added makes the tree more unbalanced than the tree currently is:

$$penalty(p_{\geq i}) = \begin{cases} 0, & \text{if } p_{\geq i} \leq \frac{p_0}{2^i} \\ 1, & \text{otherwise} \end{cases} \quad (3)$$

The counter values are taken *before* the increment caused by adding the address[5]. Finally, the similarity score for an IP is normalized by the length of the IP address (and thus the maximum possible number of penalty points): $similarity(IP) = \frac{score(IP)}{32}$

Similar to the topic score, the IP similarity score ranges from 0 to 1 and returns values closer to 1 for IP addresses sharing the same prefix (the longer the shared prefix, the higher the score).

The final formula for the waiting time function can be represented with the following formula:

$$w(IP,\ topic) = a(b + \frac{d(topic)}{d} + \frac{score(IP)}{32})\frac{1}{(1-\frac{d}{n})^{P_{occ}}} \quad (4)$$

6.1. Lower Bound

With the waiting time formula, every change in the ads stored in the ad cache may increase or decrease the waiting times of other pending requests. Therefore, an advertiser receiving waiting time $w(t_1)$ at time t_1, may get a smaller waiting time $w(t_2)$ 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 topic expires between t_1 and t_2). As a result, advertisers are incentivized to keep checking the waiting times as frequently as possible hoping for a better one and generating unnecessary overhead in the system. To prevent this behavior, we design a mechanism ensuring that any node already in possession of a ticket with a determined waiting time, cannot get a better waiting time (including the new waiting time and the time passed between the first ticket request and the subsequent) by sending new ticket requests. One solution to this problem is to take into account all the expiration times when calculating the waiting time. However, it is computationally expensive (*e.g.* $O(d)$ per request) and unfeasible in practice as it opens a possibility of denial of service attacks.

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 $w(t_2)$ is smaller than $w(t_1)$ by more than the time elapsed $t_2 - t_1$:

5. The first added address will thus always have a score of 0

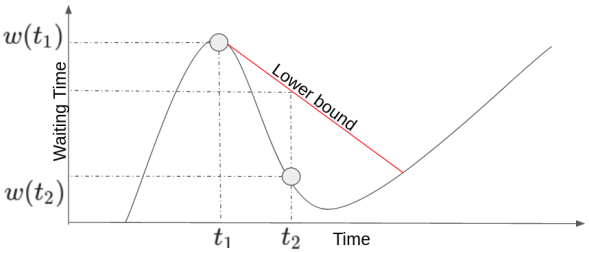

Figure 6: Waiting time lower bound.

$w(t_1) - w(t_2) < t_2 - t_1$. To ensure this is not the case, our protocol enforces a lower bound on the waiting time, *i.e.*, 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 6). However, holding such a bound for every request (*i.e.*, every combination of IP/topic) would cause significant memory overhead ($O(|IPs| \times |topics|) \gg O(d)$) and would present an easy way for an attacker to create additional state at the registrar.

To store the lower bound more efficiently, we rewrite the waiting formula as a sum:

$$w(IP,\ topic) = \frac{a \cdot b}{(1-\frac{d}{n})^{P_{occ}}} + \frac{a \cdot d(topic)}{d(1-\frac{d}{n})^{P_{occ}}} + \frac{score(IP)}{32(1-\frac{d}{n})^{P_{occ}}} \quad (5)$$

Ensuring that the lower bound is enforced for each of the three components, implies that the total waiting will respect the lower bound as well. At the same time, it only requires storing the lower bound for every IP/topic in the table and not all their combinations. This approach reduces the memory overhead to $O(d(IPs) + d(topics)) = O(d)$.

DISCv5 keeps a lower bound for each IP and topic in the ad cache. When a specific topic enters the cache for the first time, bound(topic) is set to 0 and a timestamp(topic) is set to the current time. When a ticket request arrives for the same topic, we calculate the topic waiting time w_{topic} and return the value, $w_{topic} = max(w_{topic}, bound(topic) - timestamp(topic))$. The bound and the timestamp are updated when a new ticket is issued and $w_{topic} > (bound(topic) - timestamp(topic))$.

We also maintain 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. Analysis

In this section, we analyze the properties of DISCv5 and choose optimal values for system parameters. Due to space constraints, we include an extended version of the analysis in Appendix A.

7.1. Efficiency

Memory usage is bounded by the capacity of the ad cache. We focus exclusively on registrars, as advertisers

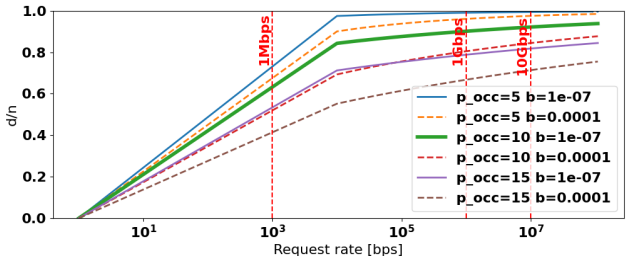

Figure 7: Ad cache space usage for different request rates.

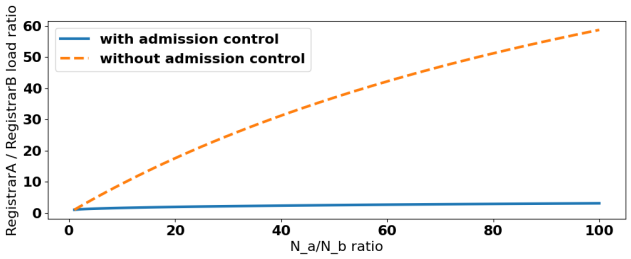

Figure 8: Load ratio between registrars located close to the most popular (*registrar A*) and the least popular (*registrar A*) topics. Parameters: $K_{register} = 5, N = 25000, N_a + N_b = 15000, N_x = 100$.

and searchers require a fixed amount of memory for their operations. The amount of ads in the cache is given by $d = xa/(a + w(x))$, where x is the number of requests constantly trying to get into the table, a is ad lifetime and $w(x)$ is the average waiting time received by requests x. In the worst case scenario, when requests x are able to achieve 0 similarity score for both the topic and the IP addresses, the waiting time formula is given by: $w(x) = ba/(1 - \frac{d}{n})^{P_{occ}}$.

The possibility of the cache going above the capacity is determined by the constant b and the exponent P_{occ}. b should be set to a small value to limit its influence on the waiting time (where IP and topic 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.

In consultation with developers of Geth [3], we assume a cache capacity of $n = 1,000$ entries and an average size of an advertisement equal to 1KB. Figure 7 presents the cache usage for different rates of incoming requests. We choose $b = 10^{-7}$ and $P_{occ} = 10$; these values provide a 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 topics in the cache and is thus bounded by its capacity $O(n)$.

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 topic ID destination, guaranteeing these operations to finish within $O(log(N))$ steps.

7.2. Fairness

We assume a Zipf distribution of the topics popularities in the system and that topic IDs are uniformly distributed in the DHT key space. For simplicity, we uniquely compare the load of *registrar A* – located close to the most popular *topic A* and the load of *registrar B* – located close to the least popular *topic B*. The ID of *topic A* is followed by N_a nodes and that of *topic B* is followed by N_b nodes in the key space.

We assume *topic A* and *topic B* are located on the opposite sides of the DHT key space, the worst-case scenario for load balancing. Both *registrar A* and *registrar B* receive different amounts of traffic for both *topics A* and B, and the same amount of traffic for other topics, represented by *topic X*, with *topic X* $\neq$ *topic A* $\wedge$ *topic X* $\neq$ *topic B*.

Registration operations achieve equal load distribution. As the closest node to the *topic A* ID, *registrar A* receives registration requests from all the N_a advertisers. As the furthest node from the *topic B* ID, it also receives requests from $\frac{N_b K_{register}}{N/2}$ *topic B*-advertisers. Analogically, *registrar B* receives requests from N_b *topic B*-advertisers and $\frac{N_a K_{register}}{N/2}$ requests for *topic A*.

As $N_a \gg N_b$, the initial number of requests is higher for *registrar A*. However, as its ad cache fills up, *registrar A* will issue higher waiting times making the requests less frequent. Figure 8 presents the registration load ratio between both registrars as a function of increasing popularity between the two topics. The load difference experiences sub-linear growth. When *topic A* is 100 times more popular, *registrar A* receives only 1.6 times more requests than *registrar B*. We also present results without the admission control (*i.e.* all the requests receive a fixed waiting time) for reference.

Lookup operations achieve equal load distribution. Let us assume again that *registrar A* is the closest node to *topic A*'s ID. All the *topic A* searchers N_a will go towards this node during their lookup operations, so the number of requests is expected to grow as N_a grows. At the same time, the more participants in *topic A*, the more ads will be placed in other buckets further away from the *registrar A*. Recall that searchers stop their lookup operations after collecting N_{lookup} peers. We set $N_{lookup} = 30$, a value commonly used by applications in the Ethereum ecosystem. As the number of ads in the network grows, more searchers are likely to stop before reaching *registrar A*. Figure 9 presents *Registrar A*'s load for increasing values of N_a. Depending on the $K_{register}$ and K_{lookup} parameters, the maximum load is experienced when the number of *topic A*-advertisers/searchers is relatively small. It goes back to 0 when the topic becomes popular in the network. We choose $K_{register}$ and $K_{lookup} = 5$ as a reasonable tradeoff between efficiency, load balance and security.

7.3. Security

DISCv5 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 the peers received by

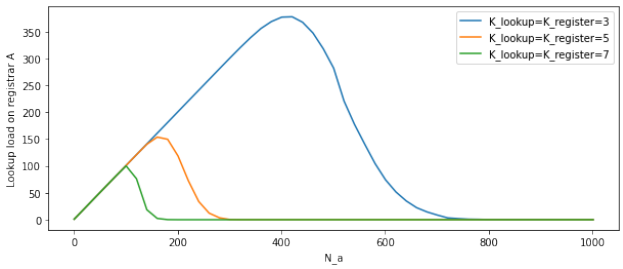

Figure 9: *Registrar A*'s lookup load as a function increasing popularity of *topic A*. Parameters: $N_{return} = 10, N = 25000, N_b = 100, N_x = 500$.

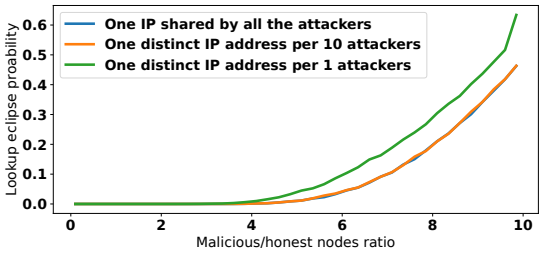

Figure 10: *Lookup eclipse probability*.

the searcher consist of malicious nodes. A searcher 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 N_{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 10 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 topic[6]. 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.

DISCv5 achieves high resistance against DoS attacks.. DISCv5 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 topic 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 a. An attacker thus has to constantly use their resource to perform the attack and the system quickly recovers once the attack stops.

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

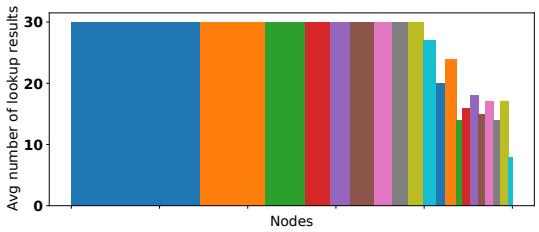

Figure 11: Average number of lookup results obtained by each node.

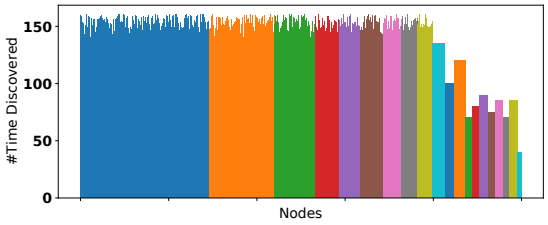

Figure 12: Number of times each node is discovered during lookup.

8. Performance Evaluation

We evaluate DISCv5 first with two complementary means: an actual implementation and large-scale simulations.

8.1. Prototype

We implemented DISCv5 in the *Go Ethereum* (Geth) client [3] and made it fully compliant with the Ethereum P2P protocol suite. We perform our initial tests on a single testbed with 24 cores AMD Ryzen Threadripper 3960X CPU and 256GB of RAM. We launch 1,000 nodes using IDs/IP addressed randomly drafted from the Ethereum live network [1] and 20 topics following the distribution perceived in Figure 1. Each node constantly register for the assigned topic and performs 5 lookup operations, aiming to discover $N_{lookup} = 30$ peers.

We first analyze the average number of lookup results obtained by each node. Figure 11 groups nodes by the topic they participate in, as indicated by the colors of the bars (order within one color is random). Topics are sorted from highest to lowest popularity, as can be seen by the width of the color zones. The vast majority of nodes is always able to find the required number of peers. Less popular topic nodes (right side of the graph) do not always obtain enough results. This is caused by the lower number of participants for those topics (< 30). Importantly, within each topic, we observe the same number of obtained results indicating high fairness. Every node in the network is able to efficiently find its peers.

Figure 12 presents the number of time each node is discovered by others. It allows us to verify that no node is discriminated in the network. We use the same visual grouping as in Figure 11. Within each topic, nodes are discovered a very similar amount of times as peers from the same topic, and we did not observe nodes that would not be discovered by their peers. The distribution closely matches the one observed in Figure 11.

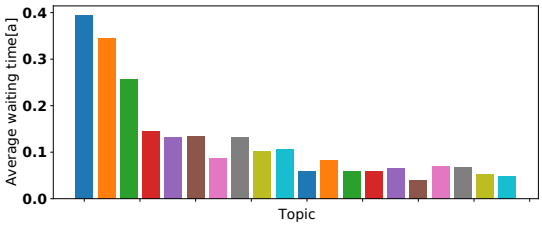

Figure 13: Average total waiting times received per topic.

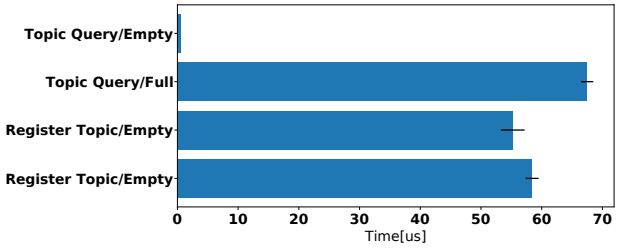

Figure 14: Request processing time.

The results indicate the DISCv5 lookup process is a sound basis to spawn a healthy, application-specific p2p network regardless of the application popularity.

At first, the equal discovery rate across topics with different popularity might be surprising. To investigate it further, for each topic, we plot the average total waiting time received by its advertisers. We express the waiting time in ad lifetimes a and keep the topic colors consistent with the one used in Figure 11 and Figure 12. The waiting time function returns higher waiting times to nodes from popular applications. As a result, those nodes will spend less time in the ad caches and more time waiting to be admitted. This mechanism allows advertisements of less popular topics to get a fair share of ad cache space and results in an equal discovery rate presented above.

Finally, we investigate the processing time required by each type of request across 1000 runs (Figure 14). We consider cases with an empty and full ad cache. Topic query performed on an empty ad cache is the fastest operation as it does not require processing the ad cache content. Topic queries performed on a full ad cache requires $66\mu s$. The register topic operation requires $58\mu s$ on average and is not heavily influenced by the increasing number of the stored ads. For all operations, we observe consistent processing time across multiple runs.

8.2. Simulations

For large scale experiments and comparison with other protocols, we built a prototype of DISCv5 in PeerSim [30], a scalable P2P network simulator.

We implement the following approaches as baselines. **1) DISCv4** described in Section 2.2. **2) DHT** is a traditional key/value store implementation using the vanilla Kademlia DHT lookup operations. In this approach, peers store their ads under desired topics at the 20 closest (by DHT distance metric) nodes to the corresponding topic ID. In DHT, registrars accept all incoming ad registrations right away (*i.e.* with no waiting time) and use a simple Last Recently Used (LRU) as a replacement policy. This solution is currently used in InterPlanetary File System (IPFS) [] - the largest storage network [25]. **3) DHTTicket** is an extension of DHT. Unlike DHT, this approach uses DISCv5's admission control mechanism (Section 6) regulating the storage of incoming registrations.

Metrics. We report the following performance metrics.

- **Message Overhead** - the total number of messages received by each node. Higher values mean larger strain put on each node.
- **Discovered Peers** - the number of application-specific peers discovered by the searchers during lookup operations. This metric is a measure of the topic search performance by the peers.
- **Discovered By** - the number of searchers each advertiser was discovered by. This metric allows us to verify whether each peer is being discovered and how the number of discoveries differs across peers.

We verify the impact of the following parameters:

- **Network Size** - the total number of nodes that participate in the discovery. We set the default value to 25,000, as similar number of live nodes were reported by the official Ethereum crawler [1] (Figure 1). We assign each node an IP address and an ID from the crawled ENS records [6].
- **Topics** - the number of distinct applications. We set the default number of topics to 300, which is similar to the observed value by the Ethereum crawler, and also experiment with values 100 and 600. To obtain the number of peers participating in each topic, we use a Zipf distribution with exponent 1.0 (obtained empirically by fitting the crawler data in Figure 1).
- **%Malicious nodes** - the percentage of attackers targeting the peers in the discovery system related to the total number of nodes participating in the topic. We experiment with values ranging from 20% to 50%, using 33.3% as default value.
- **#Attackers reusing IPs** - the number of attacker nodes reusing the same IP addresses in the attack. Lower values incur increased cost for the attacker. We set its value between 50 and 1, using 5 for the default value. In our experiments, we use hypothetical attacker IP addresses that have minimum similarity with the set of real IP addresses used by IP addresses (obtained from the crawled ENS records) in order to obtain an upper-bound on the impact of attacks.

Each simulation takes one hour of simulated time during which each advertiser tries to maintain active (*i.e.* unexpired) registrations and each nodes each perform a single lookup operation uniformly spread across the simulation time. We set the *registration table* capacity to 500 to align with the requirements from the official Ethereum DHT implementation and lifetime a to 15 min.

We set all the DHT-related parameters to the ones currently used by Ethereum DHT.Based on formal analysis (see Section 7), we set other DISCv5 parameters such as the number of registrations placed per bucket $K_{register} = 5$, the occupancy exponent $P_{occ} = 10$, the number of peers to contact per bucket during a topic lookup $K_{lookup} = 5$, the maximum number of ads a single registrar returns in a lookup $N_{return} = 10$. The total number of requested ads peer lookup (*i.e.* N_{lookup}) is we set to 30 based on similar requirements of existing applications using Geth.

8.2.1. Evaluation Results. In the rest of this section, we use violin plots [38] 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 exceed the range of the y axis.

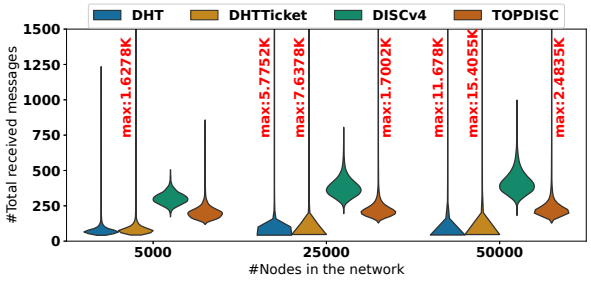

Figure 15: Message overhead for different network size during a single advertisement period.

Network performance. Figure 15 and Figure 16 present *Message Overhead* for the registration and lookup processes. For DISCv4, only lookups generate messages as there is no registration process in the protocol. DHT-based protocols introduce low overhead but overload nodes close to popular topic IDs impacting fairness. The DISCv4 protocol has a better load distribution between nodes since the destination of lookup messages is chosen randomly. However, its message overhead is the highest among all protocols. DISCv5 introduces average overhead but provides more equal load distribution compared to DHT-based protocols.

Figure 17 presents the number of *Discovered Peers* during a single lookup operation for an increasing number of topics. The random walk of DISCv4 discovers relatively low number of results per operation (< 5). The performance decreases with an increasing number of topics in the network. The targeted approach of DISCv5 and the DHT-based solutions discovers the required number of 30 peers. In rare cases, those protocols do not discover the required amount of peers. It is caused by unpopular topics the total number of participants close to 30.

Figure 18 presents *Discovered Peers* during a single lookup operation with an increasing network size. With a fixed number of topics, each application-specific network grows and it is easier to find the required amount of nodes. However, DISCv4 again suffers from poor performance for all the investigated network sizes due to its random movement in the network.

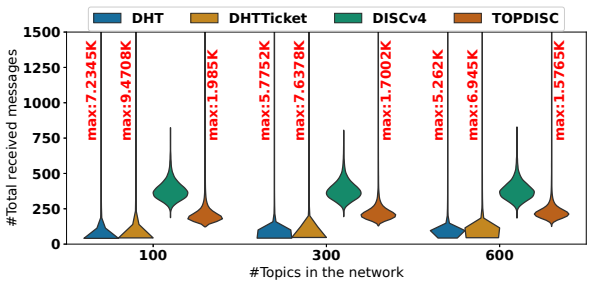

Figure 16: Message overhead for different number of topics during a single advertisement period.

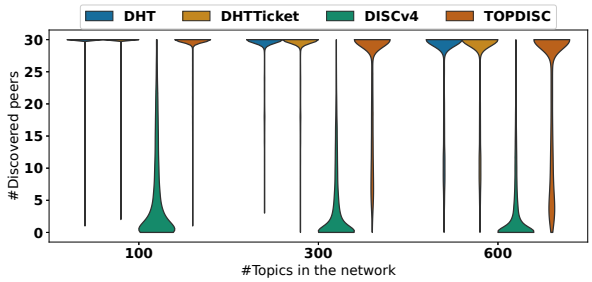

Figure 17: Discovered peers for diff. number of topics.

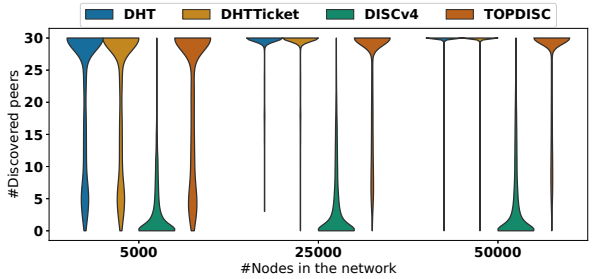

Figure 18: Discovered peers for different network size.

Eclipse attack resistance. We evaluate DISCv5 resistance against eclipse attacks where an attacker simultaneously behaves as a malicious DHT peer, a malicious registrar and a spamming advertiser (see Section 3.2). We consider a lookup eclipse when peers obtained are malicious. We assume a single entity controlling with limited pool of IP addresses coordinating all the malicious nodes. Malicious nodes register ads for the targeted topic at a rate 10 times higher than that of honest nodes. Malicious nodes return other malicious peers in response to both DHT routing and topic queries.

On top of each violin plot, we specify the *percentage of eclipsed (benign) nodes after a lookup operation*. 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 20 malicious nodes close to a target topic. At the same time, DISCv5 is shown to achieve lower eclipse rates under a non-uniform Sybil ID distribution (Section 7). The results below represent the worst-case scenario for DISCv5 and the best-case scenario for protocols we compare against. We evaluate the eclipse attacks targeting an average popular topic (approx. 500 nodes participating in the topic).

Figure 19 presents the percentage of malicious nodes discovered during a single lookup. The number of malicious node is expressed as a percentage of the total number of system participants. DISCv5 achieves close to a 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 form an efficient protection even against attacker controlling 50% of the nodes. Surprisingly, the random approach of DISCv4 performs worse than DISCv5, reaching up to 17.8% eclipse rate. DISCv4 requires querying much

more nodes to obtain the required number of peers, and therefore suffers from higher chance of querying malicious nodes. DHT has significantly the worse performance, reaching up to 59.7% eclipse rate. Thanks to our admission mechanism, DHTTicket outperforms both DISCv4 and DHT resulting in up to 6.5% eclipse rate.

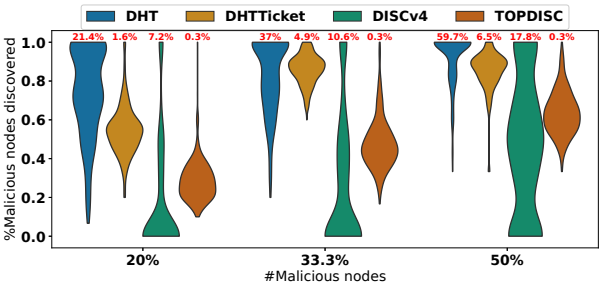

Figure 19: Lookup eclipse rate for a different number of Sybil nodes used in the attack.

Figure 20 illustrates the percentage of malicious nodes in the lookup results and the lookup eclipse rate. We vary the size of the IP address pool used by the adversary. Value 1 means that each malicious node has a unique IP address (resulting in high monetary cost for the attacker). An increasing number of IPs used by the attacker makes DISCv5 more likely to get eclipsed as the the attacker IP score in the waiting function will be similar values received by honest nodes. Even for the worst case scenario, DISCv5 achieves the eclipse rate close to 0% to 0.5%. DHT and DISCv4 suffer from significantly higher but remain unaffected by the increasing number of the IP addresses 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 DISCv5.

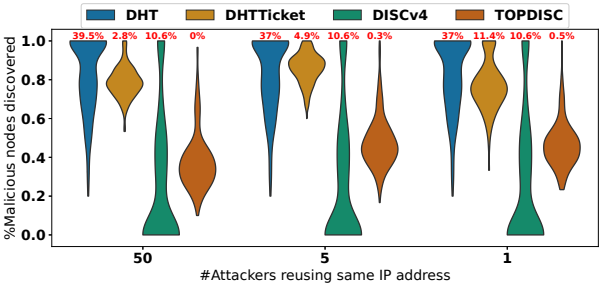

Figure 20: Lookup eclipse rate for a different number of IP addresses used by Sybil nodes.

9. Related Work

A decentralized service discovery system can be organized by directly storing the membership information in a DHT [29], [33], [34], [36]. 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 [18] 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 [5], [9], [35] (and share its weaknesses), introduce significant overhead to keep the data up to date [37], introduce a single point of failure [14], or require all nodes to be correct (*i.e.* not byzantine) [4].

Recently, multiple service discovery protocols were implemented for the blockchain space [17], [22], [27]. Unfortunately, these solutions are meant to work in small-scale systems [17], or require writing to the blockchain (thus introducing significant monetary and/or computational cost) [22], [27].

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 [15], [16], introducing some kind of Proof-of-Work [7] or sampling participant identifiers [12]. 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], [21], [31], [32]. 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 [20], [28] to make it more resistant to low-resource eclipse attacks at the DHT level. Those solution enable DISCv5 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 [26], Lazy Suzan [13] and the work of Kung *et al.* [24]. In contrast with DISCv5, 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 DISCv5 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 behaviours, ensures equal load distribution and promotes diversity in the network. DISCv5 is scheduled for deployment in future versions of the Ethereum platform. An interesting future direction is to add Sybil identities detection mechanism [12] 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).

Open science: All the code will be released open source, as well as datasets and scripts allowing to reproduce our experiments.

Ethics/Prevention of harm: We not introduce novel potential for harm beyond what has been published in the past [20], [28].

References

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

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

[3] Go ethereum. official go implementation of the ethereum protocol. https://geth.ethereum.org/.

[4] 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.

[5] 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.

[6] Davi Pedro Bauer. Ethereum name service. In *Getting Started with Ethereum*, pages 103–106. Springer, 2022.

[7] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–8, 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] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002.

[10] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[11] S. Cheshire and M. Krochmal. Dns-based service discovery. RFC 6763, RFC Editor, February 2013. http://www.rfc-editor.org/rfc/rfc6763.txt.

[12] Thibault Cholez, Isabelle Chrisment, and Olivier Festor. Efficient dht attack mitigation through peers' id distribution. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[13] 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.

[14] 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.

[15] George Danezis, Chris Lesniewski-Laas, M Frans Kaashoek, and Ross Anderson. Sybil-resistant dht routing. In *European Symposium On Research In Computer Security*, pages 305–318. Springer, 2005.

[16] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*, pages 1–15. San Diego, CA, 2009.

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

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

[19] Sumi Helal. Standards for service discovery and delivery. *IEEE Pervasive computing*, 1(3):95–100, 2002.

[20] Sebastian Henningsen, Daniel Teunis, Martin Florian, and Björn Scheuermann. Eclipsing ethereum peers with false friends. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 300–309. IEEE, 2019.

[21] 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.

[22] 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.

[23] 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 2018*, pages 91–104, 2018.

[24] 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 *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, pages 212–223. IEEE, 2015.

[25] Protocol Labs. go-libp2p-kad-dht.

[26] 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.

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

[28] 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.

[29] 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*, pages 53–65. Springer, 2002.

[30] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer*, P2P, 2009.

[31] 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.

[32] Matthieu Pigaglio, Joachim Bruneau-Queyreix, David Bromberg, Davide Frey, Etienne Rivière, and Laurent Réveillère. Raptee: Leveraging trusted execution environments for byzantine-tolerant peer sampling services. *arXiv preprint arXiv:2203.04258*, 2022.

[33] 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.

[34] 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.

[35] 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 International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 271–291. Springer, 2012.

[36] 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.

[37] 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.

[38] Mike Yi. A complete guide to violin plots.

[39] Liang-Jie Zhang and Qun Zhou. Aggregate uddi searches with business explorer for web services. *IBM Developer Works*, 2002.

Appendix: Extended Analysis

In this appendix, we provide more details on the analysis described in Section 7.

1. Efficiency

Due to the admission procedure, an advertisement for a specific topic will stay only $\frac{a}{a+w}$ of the time in the table of a registrar, where a is the ad lifetime in the table (when admitted) and w is the average waiting time. Assuming the "worst case" scenario where a request achieves 0 similarity score for both the topic and its IP address, the waiting time is $w = \frac{b \cdot a}{(1-\frac{d}{n})^{P_{occ}}}$, where n is the table capacity and b and P_{occ} are constants defined in Section 6. If a registrar receives x such requests for that topic, the average number d of ads in the table is therefore

$$d = \frac{x}{1 + b \cdot (1 - \frac{d}{n})^{-P_{occ}}} \tag{6}$$

To obtain the results shown in Figure 7, Equation 6 must be numerically solved for d. With increasing x, d will asymptotically approach the table capacity n.

2. Fairness

2.1. Load distribution of registration operations. We consider two topics A and B that are located on the opposite sides of the DHT hashspace. The registrar closest to topic A will receive ad registration requests from all N_a advertisers of topic A. On the other hand, from the viewpoint of the N_b advertisers of topic B, the same registrar is located in the furthest and largest bucket, containing half of the N network nodes. Therefore, the registrar will only receive requests from, on average, $\frac{N_b \cdot K_{register}}{N/2}$ advertisers for topic B.

Extending Equation 6 by the topic similarity score and still assuming the "worst case" of complete IP address diversity, the average number of ads d_a and d_b for topic A and B, respectively, in the registrar's table will be:

$$d_a = \frac{N_a}{1 + (b + \frac{d_a}{d}) \cdot (1 - \frac{d}{n})^{-P_{occ}}} \tag{7}$$

$$d_b = \frac{\frac{N_b \cdot K_{register}}{N/2}}{1 + (b + \frac{d_b}{d}) \cdot (1 - \frac{d}{n})^{-P_{occ}}} \tag{8}$$

$$d_x = \frac{N_x}{1 + (b + \frac{d_x}{d}) \cdot (1 - \frac{d}{n})^{-P_{occ}}} \tag{9}$$

where $d = d_a + d_b + d_x$. N_X is a "background load" representing the other topics in the system.

Again, the above system of non-linear equations must be solved numerically for d_a and d_b. If topic A is far more popular than topic B, i.e., $N_a \gg N_b$, d_a will be greater than d_b. However, because of the topic similarity score $\frac{d_a}{d}$, the advertisements for topic A will not occupy the entire table.

For a registrar the closest to topic B and the furthest from topic A, the same equations, but with the roles of topics A and B switched, are obtained. We assume that both registrars will experience the same background load N_x.

2.2. Load distribution of lookup operations. As described in Section 4.4, a searcher looking for topic A will start at the furthest bucket and progress toward the closest registrar to that topic until it has received N_{lookup} responses. Let bucket 1 be the furthest bucket containing $N/2$ registrars, bucket 2 the closer bucket containing $N/4$ registrars etc. In each bucket i, the searcher will query $\max(K_{lookup}, \frac{N}{2^i})$ registrars. The lookup analysis in Section 7 requires to calculate the probability that a searcher will reach the last bucket, i.e., the closest registrar to the topic. In the following, we calculate the distribution $p_{1..t}(Resp = S)$ of the number of responses S that a searcher will have received after traversing buckets 1 to t.

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

$$\begin{aligned} p_{rr,i}(Resp = S) &= \sum_{R=0}^{N_a} p(rr \text{ received } R \text{ registrations} \\ &\wedge \quad rr \text{ has } \min(S, N_{return}) \text{ ads}) \end{aligned}$$

where N_{return} is the maximum number of responses a registrar will return. The probability that the registrar received R registrations is the probability that R of the N_a advertisers chose the registrar. It is given by the binomial distribution:

$$p(rr \text{ received } R \text{ registrations}) = \binom{N_a}{R} \left(\frac{K_{register}}{N/2^i}\right)^R \left(1 - \frac{K_{register}}{N/2^i}\right)^{N_a - R}$$

Given R registrations, the number of ads that the registrar returns can be calculated using Equation 7 by substituting N_a with R. Combining both results, we can calculate the joint probability $p_{rr,i}(Resp = S)$.

Calculating the exact probability $p_i(Resp = S)$ that a searcher obtains S responses in bucket i and the probability $p_{1..t}(Resp = S)$ to obtain S responses after visiting buckets 1 to t is numerically intensive. Instead, we use Monte-Carlo simulation to approximate $p_{1..t}(Resp = S)$ from $p_{rr,i}(Resp = S)$. For each bucket $1 \leq i \leq t$, the simulation randomly draws K_{lookup} samples from the distribution $p_{rr,i}(Resp = S)$, in this way simulating the querying of K_{lookup} registrars per bucket. This approximation assumes that the distributon 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 Figure 9. The probability to reach the last registrar is then $p_{1..u}(Resp < N_{lookup})$, where u is the number of buckets.

3. Security

The probability that a searcher is eclipsed when looking up a topic in t buckets is the probability to only receive malicious ads from the contacted registrars. The probability that a searcher will contact a malicious registrar in a bucket i is $P_{m,i}$. The probability P_i to receive only malicious ads from a random registrar in bucket i is therefore

$$P_i = P_{m,i} + (1 - P_{m,i}) \cdot P_{h,i}$$

where $P_{h,i}$ is the probability that a honest registrar in bucket i only returns malicious ads. Assuming independence, the probability to only receive malicious ads from the entire bucket i is $P_i^{K_{lookup}}$ and the probability $P_{1..t}$ to be eclipsed in all t visited buckets is then

$$P_{1..t} = \prod_{i=1}^{t} P_i^{K_{lookup}}$$

3.1. Finding $P_{m,i}$. The probability $P_{m,i}$ that an individual registrar in bucket i is malicious depends on what source the searcher used to add nodes to that bucket. If the searcher itself is located in bucket u, it can be expected that, on average, the buckets 1 to u are fully populated by the searcher from its own routing table and that the routing table is also able to provide 17 uniformly distributed nodes for the remaining buckets. Since the buckets get smaller and smaller, we can except that we will get 8 nodes for bucket $u+1$, 4 nodes for bucket $u+2$, 2 nodes for bucket $u+3$, and one node for bucket $u+4$ from the routing table. The probability for such a node to be malicious is $\frac{n_a}{N}$, where n_a is the number of malicious nodes in the network.

This means that, for $K_{register} = K_{lookup} = 5$, the searcher has to use the nodes returned by the queried registrars to fill the empty slots in buckets $i > u+1$, as explained in Section 4.2. If the queried registrar is honest, it will return nodes that are malicious with probability $\frac{n_a}{N}$. However, if the registrar is malicious, all the nodes it returns are likely malicious, too. Consequently, the probability that a node in bucket $i > u+1$ is malicious will be greater than $\frac{n_a}{N}$ and will increase with i.

For the results shown in Figure 10, we calculate $P_{m,i}$ using Monte-Carlo simulation and assume that $u = 2$ which is the expected bucket for a random searcher.

3.2. Finding $P_{h,i}$. To calculate $P_{h,i}$, we need to know the distribution of honest and malicious ads in a honest registrar. If the registrar contains d_h honest and d_a malicious ads, P_{hi} is the probability to randomly select only malicious ads from those $d_h + d_a$ ads, which is $\prod_{i=0}^{d_h+d_a} \frac{d_a-i}{d_h+d_a-i}$.

Calculating the distribution of d_h and d_a analytically is difficult due to the complexity of the waiting time calculation. Instead, we approximate $P_{h,i}$ using the averages of d_h and d_a. On average, a registrar in bucket i will receive $R_h = \frac{N_h \cdot K_{register}}{N/2^i}$ honest and $R_a = \frac{N_a \cdot K'_{register}}{N/2^i}$ malicious registration requests, where N_h and N_a are the number of honest and malicious registrars and $K_{register}$ and $K'_{register}$ their respective number of registration attempts.

The computation of the average d_h and d_a is, in principle, similar to Equations 7 and Equations 9. However, since an attacker will only have a limited number of IP addresses, we cannot ignore anymore the impact of the IP similarity score on the waiting time:

$$d_h = \frac{R_h}{1 + (b + \frac{d_h}{d} + IPscoreH) \cdot (1 - \frac{d}{n})^{-P_{occ}}} \quad (10)$$

$$d_a = \frac{R_a}{1 + (b + \frac{d_a}{d} + IPscoreA) \cdot (1 - \frac{d}{n})^{-P_{occ}}} \quad (11)$$

where $d = d_h + d_a$. Obviously, the IP score for honest ads (*IPscoreH*) and malicious ads (*IPscoreA*) 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. Again, these non-linear equations must be solved numerically.

In the next section, we will 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.

4. IP Score

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

Note: 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 already n + k addresses in the tree, however n are random and k are identical. What score will a new random IP address get?

Note: 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 b out of m fair coin tosses are head:

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

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:

$$p\left(n, \left\lfloor \frac{n+k}{2} \right\rfloor + 1 - k\right)$$

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:

$$p\left(n,\ \left\lfloor\frac{n+k}{2}\right\rfloor+1\right)$$

In total, the score for the first level is

$$\frac{1}{2}p\left(n,\ \left\lfloor\frac{n+k}{2}\right\rfloor+1-k\right)+\frac{1}{2}p\left(n,\ \left\lfloor\frac{n+k}{2}\right\rfloor+1\right)$$

On the second level, we have four branches, of which one contains with certainty the k identical addresses:

$$\frac{1}{4}p\left(\frac{n}{2},\ \left\lfloor\frac{n+k}{4}\right\rfloor+1-k\right)+\frac{3}{4}p\left(\frac{n}{2},\ \left\lfloor\frac{n+k}{4}\right\rfloor+1\right)$$

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 $\mathbf{n+k}$ addresses in the tree, however $\mathbf{n}$ are random and $\mathbf{k}$ are identical (also random). What score will an IP address get that is identical to the $\mathbf{k}$ 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 $\mathbf{n+2^a\cdot k}$ addresses in the tree, however $\mathbf{n}$ are random and there are $\mathbf{2^a}$ "groups" of $\mathbf{k}$ entries with identical addresses. Those $\mathbf{2^a}$ addresses are distributed perfectly over the tree. What score will a new random IP address get?

Since the 2^a addresses are distributed perfectly over the tree, the new random address will see exactly $2^{a-1}\cdot k$ of them in its branch at the first level, $2^{a-2}\cdot k$ 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 $1\leq j\leq a$ is

$$p\left(\frac{n}{2^{j-1}},\ \left\lfloor\frac{n+2^ak}{2^j}\right\rfloor+1-2^{a-j}k\right)$$

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

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

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

The score for levels 1 to a is the same as in situation 4. However, for the levels $>a$, 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.

$$\text{IPScoreA}(n,k)=\frac{1}{32}\sum_{j=1}^{a}p\left(\frac{n}{2^{j-1}},\ \left\lfloor\frac{n+2^ak}{2^j}\right\rfloor+1-2^{a-j}k\right)+\frac{1}{32}\sum_{i=a+1}^{32}p\left(\frac{n}{2^{i-1}},\ \left\lfloor\frac{n+2^ak}{2^i}\right\rfloor+1-k\right)$$