Agyaat: Providing Mutually Anonymous Services over Structured P2P Networks

Aameek Singh

College of Computing
Georgia Institute of Technology
aameek@cc.gatech.edu

Ling Liu

College of Computing
Georgia Institute of Technology
lingliu@cc.gatech.edu

Abstract

In the modern era of ubiquitous computing, privacy is one of the most critical user concerns. To prevent their privacy, users typically, try to remain anonymous to the service provider. This is especially true for decentralized Peer-to-Peer (P2P) systems, where common users act both as clients and as service providers. Preserving privacy in such cases requires mutual anonymity, which shields the users at both ends. Most unstructured P2P systems like Gnutella [15], Kazaa [16] provide a certain level of anonymity through the use of a random overlay topology and a flooding based routing protocol, but suffer from the lack of guaranteed lookup of data. In contrast, most structured P2P systems like Chord [7], are Distributed Hash Table (DHT) based systems and provide guarantees that any stored data item can be found within a bounded number of hops. However, none of the existing DHT systems provide any mutual anonymity.

In this paper, we present Agyaat[1] - a decentralized P2P system that has the desired properties of privacy-preserving mutual anonymity and still accomplishes the performance benefits of scalable and guaranteed lookups. A unique characteristic of its design is its low-cost, yet highly effective approach to support mutual anonymity. Instead of adding explicit anonymity services to the network [26], Agyaat advocates the utilization of unstructured topologies, referred as *clouds*, over structured DHT overlays. Cloud topologies have an important feature of local query termination, which is critical to facilitate mutual anonymity. To overcome the drawbacks of typical Gnutella like systems, Agyaat introduces a number of novel mechanisms that enhance the scalability and efficiency of routing. Compared with existing pure DHT based systems, Agyaat provides mutual anonymity while ensuring similar routing performance (differing only by constants) in terms of both number of hops and aggregate messaging costs. We validate the Agyaat solution in two steps. First, we conduct a set of experiments to analyze the system performance and compare it with other popular pure DHT based systems. Second, we perform a thorough security (anonymity) analysis under the passive logging model. We discuss possible privacy compromising attacks and their impact, and propose various defenses to thwart such attacks.

[1] Hindi for "anonymous"

1 Introduction

With the growth of WWW and the increase in number of services provided through the internet, users have never been more concerned about their privacy. Privacy can be from the desire of escaping some form of censorship, a business concern or simply a need to prevent disclosing any information about self. Most efforts have been made to incorporate client-side privacy, in which a service requester can remain anonymous to a service provider. Services like anonymizers belong to this category. However, a consistent shift in paradigm from client-server to peer-to-peer has made possible for casual users to act as service providers and hence brought about the need for end-to-end or mutual anonymity.

As an example, assume a group of HIV patients communicating amongst themselves. With the kind of social stigma attached to such a disease, they might want to remain anonymous, both while asking questions and while responding to queries. P2P systems offer a great infrastructure for supporting such community-based groups, since it does away with any third party (like a website hosting such a discussion forum), which the users might find tough to trust. As another example, a user might want to disseminate information without the scare of retaliation from the establishment and users wanting to access it also want to stay anonymous for the same reasons. This is like an anonymous file system, where publishing and accessing the files is totally anonymous.

With such existing motivating scenarios, significant amount of work has been done to provide anonymity solutions in various systems. Projects like Crowds [19], Freenet [4], Mix [3], Onion [25] and various others [12, 13, 14] have been fairly popular. There has also been work on analysis of various anonymous protocols [8, 10]. The P2P domain has also seen some work in this area with both design of protocols like P^5 [17], APFS [21], [26] and applications of anonymity [22]. All of these are based on Gnutella like P2P systems [15], the so-called unstructured P2P networks, which have been predominant to date. However, the unstructured P2P systems suffer from the lack of guaranteed location of data, which motivated a bulk of activity in research focusing on more structured overlay networks. Utilizing the idea of consistent hashing [6], a number of distributed hash tables (DHT) based systems like Chord [7], Pastry [20], CAN [18] have been developed. These systems have attractive properties like guaranteed location of data, scalable and efficient routing, but none of them provides support for mutual anonymity.

With these issues in mind, we have developed **Agyaat** (**A**dd-on toplo**GY** for mutu**A**l **A**nonymi**T**y) – a decentralized P2P system, which provides mutually anonymous services over structured P2P networks, while maintaining the scalability and efficiency of DHT routing schemes. Compared with existing pure DHT based systems like Chord [7], CAN [18], Pastry [20], its routing performance differs only by constants in terms of both the number of hops and the aggregate messaging costs. A unique characteristic of Agyaat design is its low-cost and yet highly effective approach of adding unstructured random topologies over the DHT based overlays. It breaks the standard data-to-peer DHT mapping into two steps and utilizes an important feature of local query termination within cloud topologies to facilitate mutual anonymity. Also, to overcome the non-deterministic problem of unstructured networks, Agyaat introduces a number of mechanisms (1) to regulate the cloud topology, (2) to strengthen the scalability and efficiency of routing both between and within Agyaat clouds, and (3) to ensure the guaranteed location of data. As an interesting side effect, Agyaat allows for better management of data since its add-on cloud topologies can potentially be used to support semantic grouping based routing schemes [5], a feature found lacking in DHT based systems. To the best of our knowledge, there is no prior work similar to Agyaat, providing the best of both Gnutella-like and DHT based systems.

We validate Agyaat through two steps of performance analysis. First, we report a set of experiments conducted to analyze the system performance comparing with other pure DHT-based systems. Second, as part of our security analysis, we discuss a number of privacy compromising attacks under a passive logging model and possible defenses against these attacks.

The paper is organized as follows. We first give a brief overview of the basics of DHT based P2P systems in Section-2 and define the problem of mutual anonymity in that context, including identifying the important properties for supporting mutual anonymity in a P2P overlay network. We describe the design of Agyaat in Section-3. It includes a description of the routing protocols and details about various system parameters. In Section-4 we carry out the scalability analysis (both analytical and experimental) in terms of messaging costs and routing cost (the number of hops). In Section-5 we analyze possible attacks on the Agyaat system. We also propose pragmatic defenses against such attacks. We discuss various design benefits and flexibility of the design in Section-6. Finally, we conclude in Section-7.

2 Structured P2P Networks and Mutual Anonymity

This section overviews the basics of DHT based P2P systems and defines the problem of mutual anonymity in that context. Then, we compare the DHT overlays with Gnutella-like random toplogies and explore an important topology feature, that plays a pivotal role in facilitating mutual anonymity. We use it as our design principle.

2.1 DHT Based P2P Systems

In the recent past, most of the research on P2P systems was targeted at improving the performance of search. This is because of the inefficiency of the Gnutella type networks, where a lookup occurs by means of broadcasting the query. Every peer that receives the query sends it to its immediate neighbors (barring the one it received the query from) and so on. Whichever peer wishes to respond, traces back the reply on the same path. Since a query cannot be kept active in the network for an infinite period of time, it is typically prematurely terminated after a certain number of application level network hops (called Time-To-Live, typically 7). As a result, there is no guaranteed lookup of data. Also, due to the broadcast of queries, the resources used are exponential.

This led to the emergence of a class of P2P systems that include Chord [7], CAN [18] and Pastry [20]. These are fundamentally based on distributed hash tables and store the mapping between a particular key and its $value$ in a distributed manner across the network i.e. one node will have information about the location of only a subset of all keys. However, these techniques guarantee that given any key, the location of its $value$ can be looked up in a bounded and scalable number of hops within the network. Typically, a query takes $O(logN)$ steps, where N is the total number of nodes.

To achieve this, each peer is given an identifier and is made responsible for a certain set of keys. This assignment is typically done by normalizing the key and the peer identifier to a common space (like hashing them using the same hash function) and having policies like numerical closeness between the key and peer identifier, to identify the keys which each peer will be responsible for. For example, in Chord, all peers and keys are hashed onto a ring with identifiers ranging from 0 to $2^p - 1$ (for a p bit hash function) and the key k will be stored at the node whose hash value (position on the ring) immediately succeeds the hash of k. We call this relationship as the node being *responsible* for the key k. Also a node, N_j (node whose identifier hashes onto position j on the ring) maintains a small routing table of size p, where the i^{th} entry points to the node responsible for the key $j+2^i$. We refer the reader to [7] for more details.

As an illustration in the context of a file sharing application, the key can be a file name. All file names and the available peers' IP addresses are hashed onto a ring. Each of the peers will store information about a subset of all files and a routing table to locate other peers and to find other files. To locate a particular file, one needs to perform a *lookup* operation, which locates the peer responsible for that file. This is done by hashing the filename using the same hash function and following the underlying routing protocol. In contrast to unstructured P2P systems, all DHT-based systems require that the files (or some metadata information) are redistributed amongst the peers through a global key to file mapping scheme, so that peers responsible for their keys actually have data corresponding to them. This is an undesired feature and we discuss it later in Section-6.

In the rest of this paper, we will illustrate our concepts using Chord [7] for its elegance in description, though it is easy to build Agyaat over other structured systems.

2.2 Mutual Anonymity

To understand the basic properties of mutual anonymity in the context of P2P systems, we first examine unstructured P2P systems and then discuss the important challenges for providing mutual anonymity in DHT based systems. It is important to mention that knowing a peer's IP address is sufficient to break its anonymity.

The unstructured P2P systems provide mutual anonymity by cloaking the exact origin and termination of a query. In other words, one can say that the query comes from this group of nodes, but can not pin point the exact location. This is because such systems follow a simple message forwarding mechanism, in which a peer only gets to know the neighbor peer which forwarded the message to it and nothing about any peer in the query path before this immediate neighbor. As a result, nothing conclusive can be said of the origin of a particular message (since it can not be ascertained how far it originated in the network). Similarly, when a peer wishes to reply to a query, it sends the reply back on the same path. This is possible since every peer in the query path caches the query that passes through it and the ID of its immediate neighbor that forwarded the query. Later the reply is just forwarded to that neighbor. This process continues till the query reply reaches the querying peer. Again, nothing conclusive can be said about where the reply originated or where it terminated. Hence, mutual anonymity is assured. Note that we are assuming that one query and its reply forms the complete transaction.

In the context of DHT based systems, the problem of mutual anonymity reduces to protecting the identities of the peer issuing the query (specified by a key) and the peer responsible for that particular key. This is challenging because it contradicts the basic DHT routing table based lookup mechanism itself. Concretely, each peer has a routing table containing a set of peers responsible for certain keys, and each step in the lookup process brings the query closer to the destination peer. This ability to reach the peer responsible for a key by combining information from routing tables of various peers, is in contrast to the goal of mutual anonymity. As a result, any mechanism that ensures mutual anonymity would need to counter this basic DHT routing property. Furthermore, the support for mutual anonymity in DHT-based systems should preserve levels of scalabity, and continue to guarantee the location of data (files) within bounded number of hops.

In the next subsection, we first look at how a DHT-based system, Chord, performs a lookup and discuss the desired properties of mutual anonymity in the context of P2P routing. Then, we examine the possibilities and technical challenges of performing flooding based message forwarding over Chord.

2.3 Local Initiation, Local Termination and Mutual Anonymity

There are two ways in which a Chord lookup can proceed - *iterative* and *recursive* [7]. In iterative Chord, at each hop, the querying node gets the address of the next hop node and is itself responsible for forwarding the query to it. This clearly takes away the anonymity of the querying peer. Also, since it would know the peer that finally responds to the query, the service provider is not anonymous either. On the other hand, in recursive Chord, the node in the lookup path forwards the query, much like the message forwarding mechanism of Gnutella and a reply can be traced back using the same path. This, in a first glance, looks to be sufficient for maintaining mutual anonymity. However, we show that it fails to ensure the service provider's anonymity.

Note that the termination condition in original Chord algorithm (as proposed in [7]) is that if a node finds out that the query item hashes onto a region between itself and its immediate successor, then the successor is responsible for that item. Clearly, this fails to provide anonymity to the service provider, since it's predecessor would know where the query forwarding terminates. One may immediately think that a simple revision of the Chord protocol can make it more anonymous. Concretely, instead of letting the predecessor of a service provider peer to decide the termination of a lookup operation, we can revise the protocol to let the service provider terminate the lookup. This can be done by allocating the region between two peers to the predecessor, which is the one *before* the other on the identifier ring (following the clockwise order). Note that changing the termination condition implies making changes to the routing table as well. Now each entry would point to the node immediately *before* the query item on the ring.

However, even this revised scheme does not work. For example, consider the ring as depicted in Figure-1. Based on the routing table for the node $N48$, it is responsible for query items 49 and 50 (modified termination condition). Even if we neglect the deterministic location of key value 51 (which will be on $N51$), there exist many keys which can be deterministically linked to the responsible peers. For instance, node $N48$ knows that node $N51$ is responsible for keys 52, 53, 54, 55 and 56, because it's routing table indicates that there does not exist any node between $N51$ and $N57$ (else $N48 + 8$ entry would have pointed to it). As a result, the revised scheme still does not anonymize the service provider.

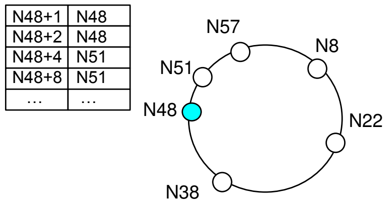

Figure 1: Recursive Chord does not work

There is one crucial insight learnt from the above analysis. Notice that when we changed the termination condition, we made certain termination scenarios local to the responding peer (i.e. the peer which is responsible for that query item). This made the system more anonymous. In fact the successful strategy to providing mutual anonymity is that such termination should *always* be local to the responding peer. Strictly local termination of queries ensures that no other peer knows where the query terminates. Also, when the responding peer initiates a reply, peers in the path cannot ascertain its origin (since the peer could just be forwarding a message received from some other peer - *local initiation*). Any topology that possesses this *local termination* and a similar *local initiation* property will be conducive to mutual anonymity. We call such a topology an *LTI topology* (Local Termination and Initiation preserving topology). While it is relatively easy to provide local initiation (like recursive Chord), local termination is a complicated task. It is easy to see that Gnutella is an LTI topology. Only the query issuer knows where the query originates and only the responding peer knows where it terminates. Also note that the reply is symmetrical to the query, except that the message initiates at the responding peer and terminates at the querying peer.

This insight serves as an important motivation and design principle for the development of Agyaat. We add-on LTI topologies (clouds) over the underlying structured DHT system to provide mutual anonymity.

3 Agyaat

In Agyaat, we provide mutual anonymity by adding Agyaat clouds (LTI topologies) on top of a DHT-based P2P network. We guarantee local termination/initiation properties by enabling the query to initiate and terminate inside the Agyaat clouds, while using normal DHT-based routing to link service requester's cloud to service provider's cloud. Such a design associates with itself a host of challenging issues, including *(a)* how to maintain routing properties in spite of two different topologies - the unstructured Agyaat clouds and the underlying DHT overlay, *(b)* how to ensure scalable and guaranteed lookups with routing performance comparable to Chord-like pure DHT systems, and *(c)* how to defend against possible privacy-compromising attacks. We will discuss these challenges and our technical solutions in the following sections and begin with an overview of the complete design.

3.1 Design Overview

For a querying peer to be made anonymous, we have to make sure that the origin of the query is not disclosed. As mentioned earlier, this is possible if the query originates from an LTI topology. Similarly if the query terminates in an LTI topology, the service provider anonymity can also be ensured. For anything between these two end points, the query can just proceed as on the DHT ring, which provides guaranteed location of the Agyaat cloud to which the responsible peer of the query item belongs. In this paper, for simplicity in exposition, we will use Gnutella as an example LTI topology.

We let peers desiring anonymity, form small unstructured LTI topologies, which we call "*clouds*" for their cloaking effect, and initiate/respond to queries only through these clouds. Every peer, in addition to being a part of the DHT ring, connects to a few other peers (neighbors) in a Gnutella like fashion. This enables the formation of small clouds on top of the DHT ring. Figure-2 shows two clouds with nodes being part of both the cloud and the DHT ring.

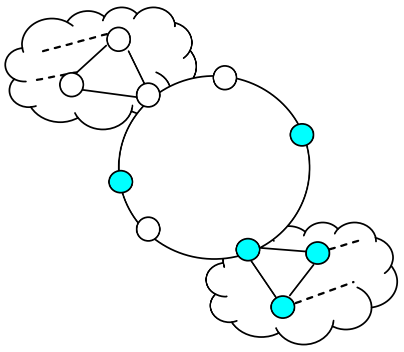
Figure 2: Add-on Clouds over DHTs

In Agyaat, each peer can initiate a query by forwarding the query to peers in its cloud. Then, one of those peers takes the query out of the cloud, onto the main DHT ring and a normal DHT lookup takes over to locate the cloud to which the responding peer belongs. At the responder's end, some peer in the cloud of the responding peer gets the query and broadcasts it in its cloud. The broadcast is required since the identity of the responding peer is not known, and so the message is sent to every peer in the cloud. When the responding peer gets the query, it initiates a reply in the response cloud, which follows a similar path back to the query originator's cloud. In order to make sure that (1) the cloud does not become too big that the cost of broadcasting becomes prohibitive and (2) the query items, if existing will be found in a cloud (guaranteed location of data), we control the size of Agyaat cloud using system parameters like maximum cloud diameter and node degree. We also study the settings of these parameters and their impact on the overall performance of Agyaat (see Section 3.3.2 for detailed discussion).

At this point, it is important to point out that while normally DHTs map a *key* to a *peer*, Agyaat breaks that process into two steps - a key now maps to a cloud (through the DHT routing) and then the cloud links to the appropriate peer (through flooding). Agyaat uses the second step to anonymize the service provider. Also, the querying and replying are symmetrical. The reply is initiated in the same manner as the query, by forwarding the message in the Agyaat cloud.

3.2 Providing Mutual Anonymity: Clouds

A cloud is a small unstructured network with local termination and initiation properties. Since a *key* is mapped to a cloud using normal DHT operations, it is essential to represent the clouds on the DHT ring i.e. to find an entry point into the cloud. This is accomplished using the concept of rendezvous nodes, similar to the work done in P2P multicast [2, 11]. Each cloud has a name and using some hash function, it is hashed onto the DHT ring. The node responsible for that region (according to the normal DHT policy) is found and it acts as an entry point into the cloud. This node is called the *rendezvous node* and is required to be a member of that cloud. Because of the consistent hashing properties of the DHT based systems, the load on a node due to its rendezvous properties will be approximately equally distributed amongst all the nodes. Also, in case some rendezvous node leaves, a new one is found by virtue of the dynamics handling of the DHT protocols and it simply replaces the old one in the cloud. Note that given a cloud name, it is always possible to find its rendezvous nodes, by just doing a DHT lookup for its hashed name.

In order to create a cloud, the desired name for the cloud is hashed onto the ring using multiple hash functions and thus multiple rendezvous nodes are found. These nodes connect to each other to form a small Gnutella network. The number of hash functions used depends upon the desired initial membership of a cloud and can be set as a system parameter[2]. For example, for two hash functions, h_m and h_n, we will select two nodes which will serve as rendezvous nodes for that cloud. The rendezvous node which was selected when h_m was used to hash the cloud name on the ring is called RN_{h_m}. Any node wishing to join a cloud, uses any one of the hash functions to get to a rendezvous node which then bootstraps it into the cloud. This is similar to the bootstrapping process of Gnutella-like systems, in which nodes give out IP addresses of other recent members of the cloud and they are contacted for any open slots. A willing member would accept the incoming peer as a neighbor and make

[2]This number effects the amount of initial anonymity of the cloud. For example, if there was only one rendezvous node, then clearly at the time of cloud creation, no immediate anonymity can be provided.

it a member of the network (a cloud in this case). Members of a cloud continue to be part of the DHT ring. Concretely, each peer needs to maintain a list of neighbors in the cloud it belongs to, along with the DHT routing tables. This condition can potentially be relaxed (see a discussion in Section-6). To allow peers to get a list of clouds currently active in the network, each peer caches the cloud names it sees queries/replies from, when it is in the lookup path of a query on the DHT. This list can be shared with an incoming peer, so that it can choose a cloud to join.

An important question still remains. How can we link the service provided by *any* member of the cloud to that cloud? It is essential since we need a mechanism ensuring that a query for a service reaches the cloud of the service provider. In other words, for a key k, how do we find the cloud which contains the peer responsible for k? This is the first step of our data-to-peer mapping scheme and can be a complicated task depending upon the kind of services being supported by the system. We have classified the types of services into three categories:

1. *Semantic Groups:* This is a kind of service in which clouds are formed in some semantic manner, i.e. the services being offered by peers in a cloud are semantically linked to each other. For example, for a file sharing application, the peers belonging to a same cloud could be sharing music from a single artist. The artist's name is used as the cloud name and queries for its songs are tagged with the cloud name. The DHT lookup will lead to the rendezvous node for that cloud and the query is forwarded to it.
2. *Services with Discovery of Service Mechanisms:* While it may be possible to link a query to a cloud semantically for many cases, there may be cases when it is not possible, for example, only the song is known and not its artist. For such a group of services, there might exist a discovery of service mechanism, which links the query item to a cloud, e.g. a central directory service. This category is actually a generalization of the semantic groups category, in which the discovery was due to the semantic nature of the services being offered.
3. *Dynamic Services:* This is a class of services when there is no possible discovery of service mechanism or it is prohibitory to use a centralized mechanism. For example, a co-operative decentralized web crawling application like [23], where peers dynamically decide which web site to crawl accordingly to a DHT based system policy. In such a case, it is not feasible to have a centralized directory service for the prohibitory performance costs. Now, for a given URL there is no way to determine the name of the cloud responsible for crawling that URL. For such services, we use *R-Rings*.

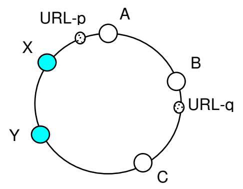

Figure 3: Dynamic services example

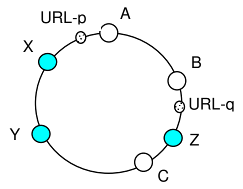

Figure 4: Entry of a peer violates guaranteed lookup

It is important to notice that for the first two categories, it is the responsibility of the peer to join an appropriate cloud. This is acceptable, since peers using Agyaat *want* to offer services and remain anonymous while doing so. For the third category, the peer can join any cloud.

3.2.1 Mapping Services to Clouds: R-Rings

Let us take a look at mechanisms that can be used to support dynamic services. We assume a similar web crawling application, where a cloud is responsible for crawling a certain set of domains. Figure-3 shows an example scenario. Peer-A, B and C belong to Cloud-1 and Peer-X and Y belong to Cloud-2. Let us assume that we assign a query item (a URL in this case) to a cloud if it hashes onto any of its members. In the figure, URL-p and URL-q hash onto Cloud-1 since they hash on Peer-A and Peer-C respectively. Remember that actually URL-p would not be crawled by Peer-A, rather by some other member of Cloud-1 (since otherwise the system would not be anonymous). These mechanisms are determined by the application's policies. An example for this scenario would be that a member peer will broadcast the domain names, which it would crawl and any other peer that sees the broadcast message chooses not to crawl that particular domain. Because of the LTI topology, this whole process can be made anonymous.

Now assume that Peer-Z enters the DHT ring at the position shown in Figure-4 and joins Cloud-2. As a result, URL-q will map to Cloud-2 as opposed to Cloud-1, which was initially responsible for it. We would have lost all information about all such URLs, thus violating the guaranteed lookup principle of DHT systems. Note that keeping forwarding information at Peer-Z that URL-q belongs to Cloud-1 will not scale and eventually become unmanageable.

Closer inspection will reveal that this problem occurs since the routing of query items is based on peers, which tend to be very dynamic in nature. On the other hand, a

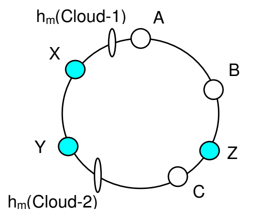

Figure 5: Main DHT Ring

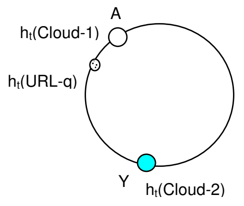

Figure 6: R-Ring for h_m

cloud is static and it persists even when existing members leave or new ones join. Therefore, any mechanism in which the routing occurs based on cloud names, will be successful to handle this issue. This leads to the idea of *Rendezvous Rings* (R-Rings).

An R-Ring is a special DHT ring, consisting of one rendezvous node from each cloud. The idea is to create a smaller ring comprising only of one node per cloud, and route queries on that ring. Concretely, we select a new hash function, say h_t which will be used in the lookup of queries. Then, we select one rendezvous node from each of the clouds, create a new DHT-based R-Ring, where each selected rendezvous node is placed at the position specified by the h_t hash value of its corresponding cloud name. For example, assume Peer-A and Peer-Y are the RN_{h_m} rendezvous nodes for Cloud-1 and Cloud-2 (Figure-5). Then we create a new R-Ring consisting of only these two nodes. An important point in the construction is that the position that a rendezvous node occupies on that ring is the h_t-hash of its **cloud name** as shown in Figure-6. Notice that Peer-A and Peer-Y occupy different positions from their original ones in Figure-5. This is done because a rendezvous node can change for a cloud, but the cloud name remains static, which is what we desire. So even when a rendezvous node leaves, the new rendezvous node would occupy the same position on the ring and hence consistent routing can be ensured.

The process of creating an R-Ring is similar to that of the main DHT rings and the protocols are well understood. Given such a system, we can route the query items on the R-Rings rather than the main DHT rings. This will always take items to particular clouds and not peers. Any change in memberships of the clouds has no effect on the lookup of items since there always will be a rendezvous node for the cloud at precisely the same position on the R-Ring. As a result, for our example application, URL-q will always map to Cloud-1 even when it maps onto a peer of Cloud-2 on the main DHT ring.

However, this can lead to significant loads on the rendezvous nodes selected to take a part on the R-Ring. To prevent this, we balance load across various rendezvous nodes of the cloud by constructing an R-Ring for each rendezvous node of the cloud. For example, if we used two hash functions, say h_m, h_n for creating rendezvous nodes, then we will have two R-Rings: one R-Ring consisting of RN_{h_m} and another consisting of RN_{h_n} nodes from all clouds. All rendezvous nodes will occupy the positions specified by $h_t(\textit{cloudname})$ in each of the R-Rings they belong to, so that all of them yield the same mappings of keys to clouds. Then a query can be routed on any of these R-Rings thus balancing the load between different rendezvous nodes for each cloud.

To summarize Agyaat, there are two key adjustments that are made with respect to the DHT systems:

- Cloud Creation: Peers willing to remain anonymous (either while querying/responding) become a member of an existing cloud or create a new cloud. The number of clouds is bounded, since a node can act as a rendezvous node for only one cloud per hash function. A node can, however, join any number of clouds.
- Routing to Clouds: All anonymous services are routed to clouds and not to peers. This can occur by a publicized discovery of service mechanism or by using R-Rings as described previously. The anonymity is ensured by the local termination properties of the cloud topology.

3.3 Agyaat Protocols

In this section, we discuss the exact protocols with which a query is generated, routed and responded to. Also we discuss other issues like sizes of clouds and the system parameters used to control it.

3.3.1 Query: Crossover Peers

A query in Agyaat originates in a cloud and is later brought out either on the DHT ring (services with discovery of service mechanisms) or the R-Rings (dynamic services) and it terminates in another cloud. An important issue in this regard is the selection of the peer which will perform the first crossover from the cloud to the ring. There are a number of important considerations:

- The crossover should not be done by the querying peer, since it takes away its anonymity during the DHT lookup.
- It should be done in a way that load caused by querying is shared almost equally among the members of the cloud.
- To prevent multiple copies of the query in the system, it should be done by a unique peer.

We enable this using a random walk in the cloud. Concretely, the querying peer makes the query message, sets up a random TTL for the message and forwards it to one of its neighbors, selected randomly. The TTL is selected randomly to cloak the origin of the message. The neighbor decrements the TTL by 1 and again selects one of its neighbors randomly and forwards the message to it. Note that, it could not have determined that the message originated at the querying peer by just looking at the message and the TTL. Now after a few hops, the TTL will reduce to zero. The peer, at which that happens, is responsible for crossing over and taking the query to the ring. Such a peer is called a *crossover peer*. This mechanism ensures that there is only a unique peer performing the query. Secondly, because of the random walk the load will be distributed equally amongst the members of the cloud. Also, similar to P2P systems, all peers in the path of the query will cache the query and remember the peer they received it from. This aids in sending back the reply through the same path.

In case of services with discovery of service mechanisms, the query is tagged with a cloud name. Therefore, at the crossover, only a DHT lookup on the cloud name would suffice. All peers are part of the DHT ring, so the crossover peer will have a routing table and can easily initiate the query. In case of dynamic services, the query goes out on an R-Ring and the crossover peer might not have its routing table (only the rendezvous nodes have routing tables for R-Rings). Then, it can query a rendezvous node of its cloud for *only* the first step of the lookup. The rendezvous node just needs to give the address for the first hop on the R-Ring and the crossover peer can continue the rest of the query in the normal iterative fashion. We call this phase the *ring phase* since the query proceeds on either the DHT ring or the R-Ring. At the end of the ring phase, the crossover peer would have found the rendezvous node for the cloud of the service provider. The query is then forwarded to the rendezvous node, which will *broadcast* the query in its cloud. This broadcast, similar to Gnutella broadcasts, can be an expensive step effecting the scalability of the system. We tackle this issue in Section-3.3.2.

Since we have successfully anonymized the service requester, we can expedite the manner in which the reply is traced back to it. The crossover peer, while forwarding the query to the rendezvous node of the response cloud, includes its IP address and a port number where a reply can directly be sent. Clearly, this does not compromise any anonymity because the crossover peer is not the peer initiating the query and many peers in the DHT ring would have seen it performing the query (in the ring phase) anyway. However, this saves us critical time and number of messages, since after adequately anonymizing the responding peer (service provider), the reply can be directly sent to the crossover peer without any intermediate ring phase. Then, the reply can be forwarded back to the querying peer.

In the response cloud, every peer would get the query message because of the broadcast. The peer wishing to provide the desired service can then make the reply message (with IP address and port information for the querying crossover peer) and start a random walk with a random TTL, similar to the walk used while initiating a query. Because of the random walk, again the load of anonymizing the responding peer is evenly distributed to all members of the cloud and there is a unique peer taking the reply out of the response cloud. However, there is no caching done in the intermediate path and the peer at which TTL reduces to zero just forwards the reply to the crossover peer of the querying cloud. This way the service provider is also anonymized.

It is important to note that Agyaat provides guaranteed lookup of data. For every data item available in Agyaat, a lookup will always succeed. This is because the query is first routed to the *appropriate* cloud using the discovery of service mechanisms or R-Rings and then *broadcasted* in that cloud, which ensures that every peer in the cloud receives the query and can respond.

3.3.2 Size of the Clouds

As mentioned before, since the query is broadcasted in the response cloud, we need to make sure that the size of the cloud does not become too large to make the broadcast costs prohibitive. In order to control the size of the cloud, we use two parameters:

- *R-Diameter*: It is the maximum distance of any peer from *any* rendezvous node. The distance is measured in number of application level hops in the underlying topology. It is denoted by r_{diam} and serves as the "length" dimension of the cloud. This parameter is important, since any query broadcasted by the rendezvous node will have an upper bound of r_{diam} on the number of hops required to reach any peer of the cloud. Also, the distance from *any* rendezvous node is used, since the query can be broadcasted by any of the rendezvous nodes (depending upon which R-Ring is used, for example).
- *Degree*: It is the maximum number of neighbors (direct connections), a peer can have in the cloud. It is denoted by m and serves as the "breadth" dimension, controlling the density of the cloud.

Restricting r_{diam} and m to reasonable values will restrict the size of the cloud. In Section-4, we will show empirically how these parameters effect the overall costs for Agyaat and try to get best possible values.

To enforce these parameters, every peer keeps a vector of its distance from all the rendezvous nodes and stops accepting new neighbors when the limits are reached. The distance vector is easy to compute in a recursive fashion. In every ping cycle[3], a peer computes its distance vector from the distance vectors of its neighbors. For example, for a cloud with three rendezvous nodes, the distance vector of a peer with k neighbors is equal to $[1 + min(d_{i1}), 1 + min(d_{i2}), 1 + min(d_{i3})]$ where $1 \leq i \leq k$ and d_{ij} is the distance of i^{th} neighbor from j^{th} rendezvous node. Since we are taking a minimum, this number will converge. Now a peer which has m neighbors or is at r_{diam} distance from a rendezvous node will not accept any new incoming peers and the cloud is said to be *saturated*. Small temporary aberrations can be tolerated, since they will be short-lived due to the convergence and hence, not cause a major performance dip.

4 Performance Analysis

Next, we evaluate Agyaat's performance. We will also discuss the effect of system parameters and provide another mechanism for restraining the size of clouds.

4.1 Scalability: Analytical Analysis

We analyze Agyaat's scalability in terms of both the number of hops each query transaction requires and the aggregate number of messages used. A single query transaction includes the costs for both the query message and the reply message to reach their appropriate destinations. First, we look at the number of hops. We can divide the cost in the following components:

- *Query Cloud Hops:* It is the number of hops in the querying cloud. It is denoted by h_{query}. If the length of the initial random walk is q_{rand}, then clearly $h_{query} = 2 * q_{rand}$, since the reply is traced back on the same path.
- *Ring Lookup Hops:* It is the number of hops on the DHT ring or the R-Ring, once the crossover happens. It is denoted by h_{ring}.
- *Response Cloud Hops:* It is the number of hops in the response cloud. It is denoted by h_{resp} and is equal to the sum of hops due to the broadcast of the query in the response cloud (h_{bcast}) and the random walk initiated for the reply message (r_{rand}); that is, $h_{resp} = h_{bcast} + r_{rand}$.

Also remember that there can be two more hops required when (1) the crossover peer in the querying cloud queries the rendezvous node for the first hop for an R-Ring and (2) a peer in the response cloud sends the reply directly to the crossover peer. Hence, the total number of hops is given by:
$hops = h_{query} + h_{ring} + h_{resp} + 2$
$hops = 2 * q_{rand} + h_{ring} + h_{bcast} + r_{rand} + 2$

Now, we restrict the random walks by r_{diam}, since that would traverse the whole length of the cloud. Therefore, $1 \leq q_{rand} \leq r_{diam}$ and $1 \leq r_{rand} \leq r_{diam}$. Also, $0 \leq h_{bcast} \leq r_{diam}$, since r_{diam} is the farthest the responding peer can be. As a result,
$hops \leq 2 * r_{diam} + h_{ring} + r_{diam} + r_{diam} + 2$
$hops \leq 4 * r_{diam} + 2 + h_{ring}$

Also we know that $h_{ring} = O(logN)$, where N is the number of nodes on the ring. In case, the query occurs on the DHT ring, N is the total number of nodes in the system. In case of R-Rings, N is the number of clouds, since there is one node per cloud in the R-Ring. Also, typical r_{diam} values will be small constants like 7. As a result, $hops = O(logN)$, which implies that Agyaat is as scalable as DHTs.
The analysis is interesting, specially for dynamic services. It shows that the total costs are $O(logN)$, where N is the number of clouds as opposed to the total number of nodes for normal DHTs. In case the number of clouds is less than the total number of nodes by a big margin, it can compensate for differing constants and potentially take lesser number of hops than normal DHT based systems! However, the caveat is that lesser number of clouds implies greater number of nodes in each cloud, which would require increasing r_{diam} and m values to accommodate them. This increases the differing constants and also increases the messaging costs.

For aggregate number of messages used, similar analysis will hold. In the query cloud and the ring phase there is one message per hop. The only difference is because of the broadcast of the query in the response cloud. Since size of the clouds[4] is bounded by constant parameters, Agyaat will still be similarly scalable. We omit the exact proof because of space constraints. Note that the differing constant in this case would be much higher because of the broadcast in the response cloud.

4.2 Scalability: Experiments

To show empirically that Agyaat is as scalable as DHT systems, we created various Chord networks varying the number of nodes from a few to 5,000. We then added-on

[3]Gnutella requires each members to periodically ping its neighbors to check for node failures. As a result, no extra messages are used.

[4]In a saturated cloud with Gnutella like topology, there can be at most $m^{r_{diam}}$ nodes

clouds to each topology. Each peer becomes a member of some cloud and a cloud is created when all existing ones are saturated. We used Gnutella as a sample LTI topology for clouds. Then we ran a large number of queries in the system and computed the average costs. The queries were selected randomly, that is, a peer belonging to some cloud is randomly selected to query for a service by another random peer. Figure-7 shows the various costs for dynamic services (based on R-Rings). As it can be seen the costs for Agyaat and Chord follow the same trend and differ by a constant. In this figure, we used an r_{diam} of 7 and m of 5. We have also depicted the number of hops on the R-Ring. As we discussed before, this number is smaller than for Chord, since the number of clouds formed (size of R-Ring) was smaller than the total number of nodes in the system (size of Chord ring).

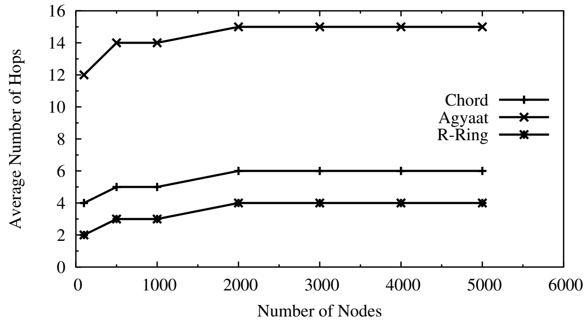

Figure 7: Agyaat Scalbility: Hops

The constants are due to the random walks in the query cloud and the response cloud and the tracing back of the reply. Since their lengths are bounded by r_{diam}, the average difference would always be close to $\frac{1}{2} * (3 * r_{diam})$ ≈ 10 in this case. This shows that Agyaat is as scalable as typical DHT systems and confirms our analysis.

Next we look at scalability in terms of aggregate number of messages transmitted for a single query transaction. This includes the messages exchanged during the random walks, tracing back of the reply and most critically the broadcast of the query. Analytically, we mentioned that Agyaat should be as scalable as DHT systems. Figure-8 confirms this hypothesis. As we can see, the overall trend for Agyaat is similar to Chord, in which case number of messages is equal to the number of hops. Also, a big component of the costs of Agyaat is the number of messages in the Response Cloud, which includes the primary costs of broadcasting. The number of messages in the Query Cloud and on the R-Ring are small in comparison. Note that for bigger networks (P2P networks are millions in strength), the gap between Chord and Agyaat would be insignificant. For smaller networks, where this gap can be an issue, we can have different cloud topologies (Section-6.2).

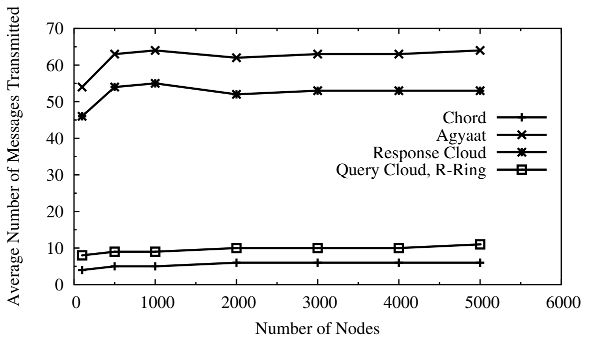

Figure 8: Agyaat Scalability: Messages

Overall, we can see that Agyaat lives up to the promise of providing mutual anonymity without sacrificing the scalability properties of the DHT based systems.

4.3 Effect of System Parameters

Figure-9 shows the average number of hops when r_{diam} is varied from 3 to 9 with m fixed at 5 for a 5,000 node topology. Increasing r_{diam} allows more peers to be added in a single cloud, increasing the cloud size. From the figure, we see an interesting trend. While the cost of Chord stays the same (r_{diam} does not effect the main DHT ring), Agyaat begins to take more number of hops even when the number of hops on the R-Ring decreases.

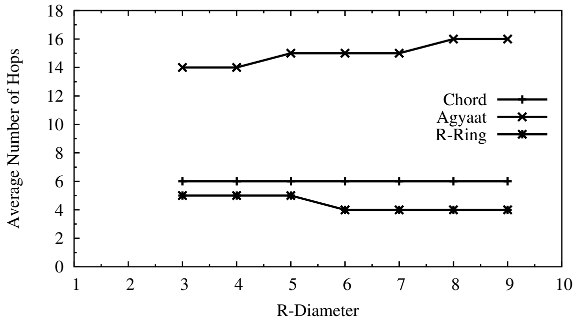

Figure 9: Effect of R-diameter on number of hops

This happens because with the increase in r_{diam}, even though the total number of clouds drops (because each cloud can accommodate more), the average lengths of the random walks and the number of hops before the broadcast reaches the responding peer in the response cloud also increases. This increase offsets the decrease in R-Ring lookup hops. While this would indicate that we should keep r_{diam} to a minimum, notice that it effects the level of anonymity offered by a cloud. Very small r_{diam} values lead to very small clouds and that provides little anonymity.

Next, we look at how r_{diam} effects the total number of messages transmitted. Figure-10 shows the average number of messages transmitted for a similar topology. As can be seen, with the increase in r_{diam}, the number of messages increases linearly, with the main component being the messages transmitted in the response cloud.

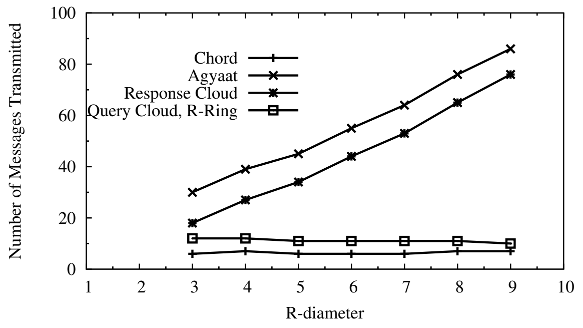

Figure 10: Effect of R-diameter on number of messages

Next, we plot similar graphs for varying values of the degree m with r_{diam} fixed at 7 (5,000 nodes). Figure-11 depicts the effect of the degree on the average number of hops. Note that the increase in m decreases the average number of hops for Agyaat and infact there is a drastic decrease in the number of hops on the R-Ring. This occurs since increasing m allows clouds to become very dense and allows more and more peers to join the same cloud. This reduces the number of clouds very quickly.

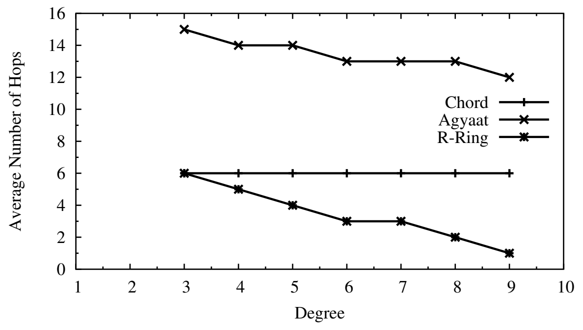

Figure 11: Effect of Degree on number of hops

However, with the clouds becoming more dense, there is a penalty to be paid in regards to the number of messages transmitted. This is because now, a large number of messages will be transmitted in a response cloud because of the broadcast of query messages. This can actually be seen in Figure-12, where increasing m drastically increases the total number of Agyaat messages, with the biggest component being the broadcasting. In addition, larger m demands more resources from the peers, since they keep m open connections at all times.

This analysis indicates that we can control system performance by varying the two parameters with varying m providing fast changes and varying r_{diam} allowing for smaller fine tuning. Also, it appears that smaller r_{diam}

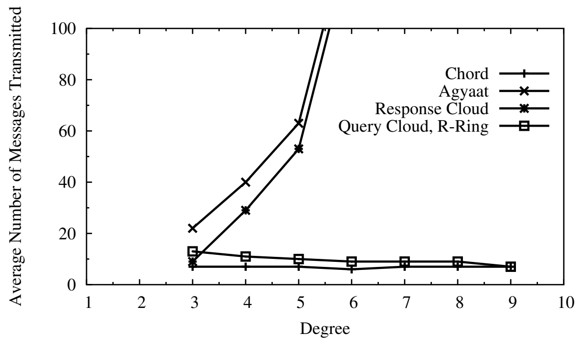

Figure 12: Effect of Degree on number of messages

and m values, as limited by the amount of anonymity required, would be the best.

4.4 Another Control Mechanism

Notice that varying the system parameters have an effect on the system in an indirect way. Changing parameters changes the number of clouds required to accommodate all peers and that effects the overall performance. Figure-13 shows how the two parameters effect the number of clouds for a 5000 node topology. As can be seen from the graph, m has a much more drastic effect.

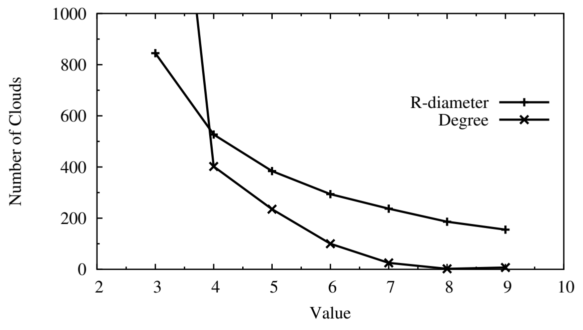

Figure 13: Effect of parameters on number of clouds

This leads to another possible mechanism to control anonymity/performance trade-offs. A system can fix the number of clouds, and control its characteristics that way. It would be implemented by creating all the clouds in advance and peers joining any cloud that is not saturated. This scheme requires apriori knowledge of the capacity of the system and might over/under provision the network with a wrong estimate. We plot the system performance using this mechanism in Figure-14 and 15.

5 Security Analysis

In this section, we look at possible privacy-compromising attacks on Agyaat and measure their effect. We explain how tough it is to design a group defense, in which non-malicious nodes follow a com-

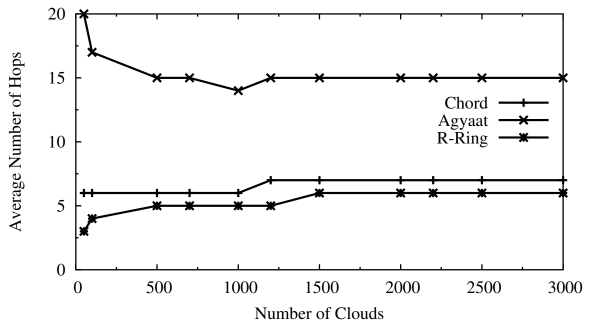

Figure 14: Effect of number of clouds on hops

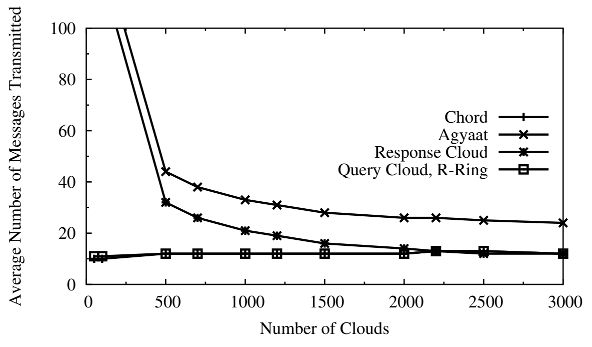

Figure 15: Effect of number of clouds on messages

mon strategy to thwart such attacks and then design few pragmatic individual defenses. First, we explain the model of our security framework.

5.1 Model

Our primary aim with Agyaat was to support mutually anonymous services over structured P2P networks. In this paper, we concern ourselves only with privacy compromising attacks. We intend to investigate other attacks like DoS, routing attacks, impersonation attacks in our future work. Since, we anonymize the system through clouds at the end points, all privacy compromising attacks have to target the cloud topologies. For example, a group of malicious nodes can join a cloud and try to locate querying/responding peers.

We assume the passive logging model [10, 8] for the analysis of our system. In this model, a number of co-operating nodes passively log the messages going through them and share the logs offline to try and find out origins/destinations of the messages. This model does not allow for any online communication between malicious nodes. Hence the nodes cannot strategize on the fly. Also, nodes are not allowed to manipulate existing protocols (it can give out their malicious nature). Thus, in the system, the malicious nodes will follow all protocols and there is no way to identify a malicious node by scrutinizing its behavior in the system. Also, we call an attack successful if the malicious nodes can find out the origin/destination of a message with probability 1. In this case the victim node is said to have been compromised. In what follows, the term “bad” stands for malicious intent and “good” for no malicious intent.

Note that for a good node, say Peer-A, to stay anonymous, all that is required is one good edge, that is, any connection with a non-malicious node, say Peer-B is enough to stay anonymous. This is because malicious nodes cannot conclusively say (with probability 1) that the message originated at Peer-A or Peer-B. If they see it coming out of Peer-A, maybe Peer-B sent it to Peer-A and Peer-A is only forwarding it. On the other hand, it could very well have originated at Peer-A. This implies that for an attack to be successful, the malicious nodes need to completely surround the victim node.

There is another subtle requirement. Not only do the malicious nodes need to surround the good node, they need the *knowledge* that the good node does not have any other edge (other than the connections to the malicious group), since the good node might also be connected to some other good node in the cloud. This *knowledge* is tough to obtain in a Gnutella like network, since the peers only hold information about their immediate neighbors. However, given significant resources at disposal of malicious nodes, some design features of Agyaat can provide this critical information. For example, each peer maintains a distance vector from the rendezvous nodes. Provided enough bad nodes surround the good nodes in the cloud, looking at the distance vector, it may be possible to observe if there is an edge between two good nodes. Also, if a good node has m bad connections already, it rules out that possibility.

There is still one more scenario. Assuming there are less than m bad connections, looking at the distance vectors might not be enough because the good node might be connected to a degree-1 node, i.e. a node which is only connected to that good node and thus is unobservable from the bad nodes' perspective. Unfortunately, it is possible to exclude even this case. That is because of Gnutella pings. When a node receives a ping, it returns a list of peers which can be used to establish more connections (the return message is called a pong). This list is typically a list of the node's neighbors or other peers it connected to, in that session. As a result, it is highly likely that enough information is available to know all the nodes in the cloud and thus rule out a phantom node. It might appear that a node can avoid sending the list of peers in a pong. We do not advocate this since it is a critical step of bootstrapping and the only way incoming peers establish connections within the cloud. We would also like to point out that attacks like predecessor and intersection attacks [10, 8] cannot be successful in P2P kind of environments, where it is unlikely that a pair of peers interact with each other repeatedly and for an extended period of time (i.e. create a session).

5.2 Attacks

As described, a malicious group of nodes can mount an attack by surrounding the good nodes in the cloud. Given a fixed amount of resources, a configuration as in Figure-16 (for 3 good nodes and $m = 2$), in which malicious nodes (solid) surround maximum number of good nodes possible, gives the best possible attack scenario. Infact the following lemma holds:

Lemma: *In a network with G good nodes and degree m, minimum number of malicious nodes required to completely surround all good nodes is given by m if $G < m$ and G otherwise.*
Proof: For $G < m$, it is trivially true, since we need atleast m bad nodes to saturate each good node. For $m \leq G$, consider a network, where i^{th} good node is connected to the bad nodes numbered from $(i-1)m+1$ $mod\ G \ldots (i.m + 1)\ mod\ G$. It will saturate all good nodes with only G bad nodes. Figure-16 also is a similar network for $G = 3$ and $m = 2$.

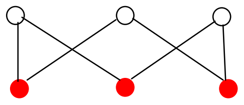

Figure 16: Best Possible Attack Scenario

However, this topology requires malicious nodes to join in a particular manner and is not possible without sophisticated and dynamic manipulation, which is not permissible under the passive logging model. On the other extreme, if all the nodes join the clouds in a random fashion, there is always close to zero compromises. This happens because in almost all cases, each good node finally connects to at least one other good node, thus foiling a successful attack. This observation leads us to believe that for a successful attack in the assumed model, malicious nodes need to attack in groups and completely surround a few good nodes if not most. For our discussion, we will always consider the attack as happening when a cloud is being created, though it is easy to extend it to the dynamics of the cloud[5].

First, we consider a simple attack. In this attack, the group of malicious nodes creates a new cloud and join en masse waiting for good nodes to join. This way, they can surround good nodes as and when they come in. Notice that when a node joins a cloud, it immediately establishes a few connections and keeps some available for incoming peers. We simulated this situation and measured the effects. It is easy to see that the system parameter m will have an effect in these scenarios, since it controls the number of edges each node can have. We plot two graphs to analyze this situation for different values of m. Figure-17 shows the graph for $m = 5$ and Figure-18 for $m = 7$. The graphs are plotted to indicate the evolution of clouds. Assume the fraction of malicious nodes in the cloud (when fully created) to be f. The graph plots the quality of the cloud, while it is being created, measuring percentage of good nodes in the cloud which are compromised (Y-axis) when some percentage of good nodes are added in the system (X-axis). For example, $f = 0.8$ indicates that 80% of the total nodes are malicious in the cloud *when it is fully populated*. Then the point (15,50) indicates that when 15% of the total good nodes were added, 50% of them were compromised. We chose to represent it in this form, since it better illustrates the general performance of the system during its most critical phase of cloud creation.

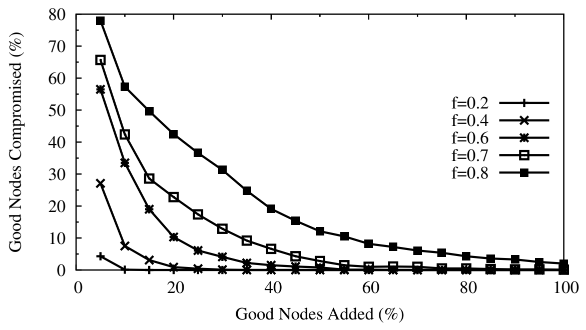

Figure 17: Simple Attack (m=5)

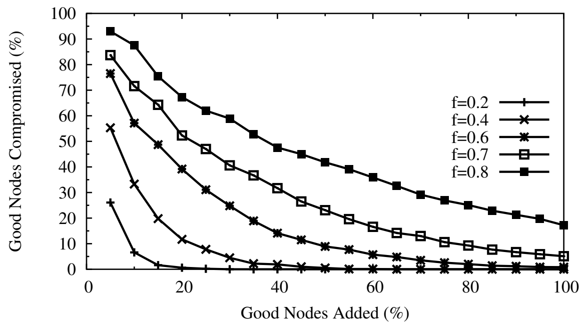

Figure 18: Simple Attack (m=7)

As the graphs indicate, less and less percentage of nodes remain compromised as and when more good nodes join the cloud. In many cases, none of the good nodes are finally compromised. This occurs since with more good nodes added, there is a greater chance for a good node to have a good edge. For large fraction of bad nodes,

[5]In that case, it can be assumed that the cloud is always *being* created, i.e. never reaches a stable state

the attacks have greater effects. For example, in Figure-18, for $f = 0.8$, even when the cloud is fully populated, around 20% of the good nodes are compromised. Also more nodes are compromised for greater values of m.

There is another possible attack in which malicious nodes can cause greater havoc. Note that the previous attack decayed since with more good nodes coming into the system, there was a greater chance of each having a good edge. This was aided by the fact that the bad nodes had joined the cloud before the good nodes started coming in. So when the good nodes entered, they left some slots available for later use and that was primarily responsible for the good edges. Any attack that surrounds the good nodes when they join the cloud by using up all possible slots would certainly be more successful. For example, an attack in which bad nodes come in blocks of a few, arriving after intervals of good node arrivals. This should work better, since there is a greater chance of surrounding the node (a block of bad nodes receives the good nodes and another block joins later to saturate the good nodes). As shown in Figure-19, this attack is certainly more successful and more good nodes remain compromised at the end. In this simulation, bad nodes arrived in 10 equi-sized blocks after every 10% of good nodes joins. In actual situations, this might be tough to achieve though approximations can be made, e.g. malicious nodes can arrive based on time intervals derived from the historical network behavior.

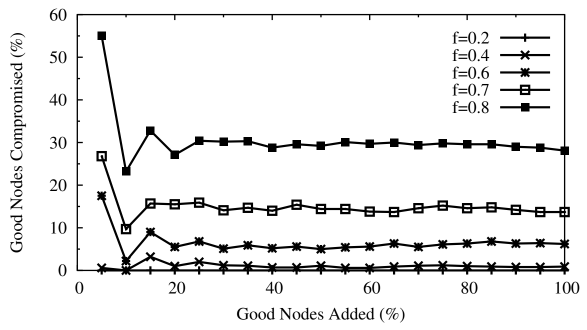

Figure 19: Block Attack (m=7)

5.3 Defenses

In this section, we discuss defenses against various privacy-compromising attacks. We show that it is tough to design a group strategy, which the good nodes can follow collectively, to reduce the risk of attacks. We then propose a number of pragmatic individual defenses.

It is important to recognize the fact that it is impossible to identify a malicious node under our security model. All malicious nodes behave completely identical to the good nodes. Hence, a good node while trying to defend itself from such an attack can only do so probabilistically, hoping that all its neighbors are not malicious. For example, it may decide to move within the network (dump its existing connections and create new ones), hoping that it will have at least one good edge. Assuming that there are G good nodes and B bad nodes with available slots[6], then this will happen with probability $1 - \left(\frac{B}{B+G} \cdot \frac{B-1}{B+G-1} \cdot \dots \cdot \frac{B-m+1}{B+G-m+1}\right)$ for degree m. However, if all nodes (including malicious) move randomly, this effect will wither out, since that brings it to the same situation as during the creation of the cloud. Any similar group behavior can always be countered by malicious nodes, since they possess greater knowledge about the network. They can monitor a large part of the network and can also identify the good nodes. Therefore, it is tough to design any effective group defense. However, individual nodes can still try to move around in the cloud (maybe immediately before querying) hoping that they connect to atleast one good node.

There is one complete defense to any kind of attacks. We call this a **Get A Buddy Along (GABA)** defense. In this, pairs of good nodes, Peer-A and Peer-B, join a cloud together and establish one edge between them. This would require the good nodes to trust each other that they will not be sharing logs with any other peer. Other than that requirement, this is an excellent defense. The nodes need to trust each other only for not sharing the logs. No other private querying/replying information needs to be shared. This is because for any message that Peer-A receives from Peer-B, Peer-A cannot conclusively ascertain that it originated at Peer-B.

There are a few other defenses that might work. Peers might want to connect to the rendezvous nodes, since they are randomly selected based on their positions on the DHT ring and are less likely to be part of a malicious group active in the cloud. However, it is likely that the malicious nodes would connect to the rendezvous nodes and fill all their available slots. Also, if one assumes that malicious nodes would be a group of people from the same area network (a dorm or college), it might be beneficial to connect to nodes from different domains. Even though it is not an accurate defense, it does make the task of malicious nodes a little tougher.

To summarize, we presented possible privacy compromising attacks. We showed that while there might not be a group strategy for defense, individual nodes can defend themselves by moving randomly, using GABA defense and trying to connect to rendezvous nodes.

[6]Note the term *available slots*. Saturated nodes cannot accept new connections and hence cannot contribute.

6 Design Benefits

In this section, we will discuss other benefits of the Agyaat design, which can be seen as by products. We will also briefly discuss possible extensions that can make Agyaat more secure and provide greater features.

6.1 Data Management

Let us first look at how Agyaat influences the management of data. In typical DHT systems, a data item is stored at a peer that immediately succeeds the hash of the data item on the DHT ring (or appropriate meta information is stored at such a peer). This leads to a common drawback of DHT systems, in which the publisher does not necessarily control the access to its data and some work has been done [9] to partially remove this problem. It is interesting to see that Agyaat can avoid this issue to a great extent. For example, a company wishes to provide some services (say publish some content) and needs to use a cluster of machines. Now, if it uses a simple DHT based mechanism, its data may get hashed to various different nodes in the network and it in turn, may be required to host some other company's data (maybe its competitor!). With Agyaat, the company can create an appropriate cloud and its cluster of machines can join the cloud. The cloud name can be the name of the company or an appropriate entry is made in a central directory service. Thus, queries for its services can be routed to its own cloud of machines.

Also, greater semantic grouping is possible with various companies offering similar services belonging to a single cloud. This leads to a more structured management of data, which ironically, structured P2P systems do not provide. It is still possible to have a few outside nodes - the rendezvous nodes, since they are selected based on the DHT ring, though the cloud name can be manipulated to select a particular node as the rendezvous node [2]. Also, for advertisement of services, clouds can tag both queries and replies with cloud based information like statistics (parameter settings) and services offered. This can be cached at intermediate peers and made available to peers looking to join a cloud.

6.2 Cloud Topologies and DHT Membership

As shown in the analysis section, Agyaat has higher messaging costs primarily due to the broadcast of the queries in the response cloud. It was from the fact that it is required that each message reaches all peers in the cloud. We can alleviate this problem by changing the topology of a cloud. Recall that our only requirement for a cloud was that it should be an LTI topology. If we take an overall look at Agyaat, it itself is an LTI topology (since both the query initiation and termination happen locally in the clouds). So we can imagine a two level hierarchical topology in which the top-level of the cloud is another DHT ring. The queries coming into the cloud are hashed onto this top level ring and appropriate level-2 rendezvous nodes are found, which broadcast the query in the lower level Agyaat cloud. The idea is to reduce the number of nodes receiving the message by appropriately routing it to a smaller subset of nodes. This can be extended to greater than two levels as well. However, note that the anonymity provided will be less since now the anonymizing components of the clouds will be smaller. Hence there is a lesser cloaking effect. Also note that now, joining a cloud would be a little more complicated, since based on the services being offered, the peer would join a particular unstructured component. It is also possible to have clouds with different topologies in the same Agyaat system, offering variable levels of anonymity and performance benefits. The clouds can advertise this feature through mechanisms described earlier. Then, incoming peers can tradeoff between the two based on their desired applications.

Another interesting idea is to allow peers to be part of a cloud without being on the main DHT ring. This will offload the load on the DHT ring and is quite easy to support in Agyaat. During a crossover, if the peer is not a member of DHT, it can query the rendezvous node for the first step, similar to the procedure used for R-Rings.

6.3 Securing Agyaat

To a careful reader, it might be evident that other than privacy, there are potential vulnerabilities in the system. Many of these vulnerabilities are due to the underlying DHT topology. For example, how to ensure that the routing takes place according to the protocol and a node does not mislead the querying peer by giving a wrong next-hop address? Another issue is the basic issue of identity management - how to ensure that peers actually are physical nodes and that a single peer is not pretending to be a group of nodes. Also how can we ensure that the reply reaches the querying peer un-tampered? These and other issues are being extensively explored [1, 24] and solutions can be directly applied to Agyaat. Also, there are a few vulnerabilities due to the design of Agyaat. For example, there is clearly more power with the rendezvous nodes and the crossover peers. We need to ensure that these behave properly and work around against possible malicious behavior (We can have multiple random walks, hoping that one of them will succeed). A closer analysis would probably reveal more vulnerabilities and we plan to take on this challenge of securing Agyaat in the future.

7 Conclusions and Future Work

We have presented the design and development of Agyaat, a decentralized sytem that provides mutually anonymous services over structured P2P networks and still ensures scalable lookup and guaranteed location of data. We identified critical topology properties such as local termination/initiation which are essential for mutual anonymity and introduce clouds (LTI topologies) to incorporate such properties into Agyaat. A unique characteristics of the Agyaat design lies in its low-cost and yet highly effective approach to supporting end-to-end (mutual) anonymity. We described a number of mechanisms to enhance the scalability and efficiency of routing between and within Agyaat clouds, ensuring the guaranteed lookup of data. We showed analytically and empirically that Agyaat is as scalable as DHT based systems in terms of both number of hops and aggregate messaging costs (differing only by constants). We also studied the effect of various system parameters and performed a security analysis of the system including possible anonymity-compromising attacks and proposed defenses to such attacks. In future, we intend to work on other security aspects of Agyaat, aiming at providing mutually anonymous and secure services over structured and decentralized overlay networks.

References

[1] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, 2002.

[2] M. Castro, P.Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002.

[3] D. Chaum. Untraceable electronic mail return addresses, and digital pseudonyms. *Communications of ACM*, 24(2):84–88, Feb 1981.

[4] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *LNCS*, 2001.

[5] A. Crespo and H. Garcia-Molina. Semantic overlay networks for p2p systems, 2002.

[6] D. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 97.

[7] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, Aug 2001.

[8] M. Wright et al. An analysis of the degration of anonymous protocols. In *Annual Symposium on Network and Distributed System Security*, 2002.

[9] N. Harvey et al. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of USITS*, 2003.

[10] P. Syverson et al. Towards an analysis of onion routing security. In *Workshop on design Issues in Anonymity and Unobservability*, July 2000.

[11] S. Ratnasamy et al. Application-level multicast using content addressable networks. *LNCS*, 2001.

[12] M. Freedmain, E. Sit, J. Cates, and R. Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *IPTPS*, 2002.

[13] Freedom. http://www.freedom.net/, 2003.

[14] E. Gabber, P. Gibbons, D. Kristol, Y.Matias, and A. Mayer. Consistent, yet anonymous web access with lpwa. *Communications of ACM*, 42(2), 2002.

[15] Gnutella. http://gnutella.wego.com/, 2002.

[16] Kazaa. http://www.kazaa.com/, 2002.

[17] B. Bhattacharjee R. Shwewood and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, Aug 2001.

[19] M.K. Reiter and A.D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1:66–92, Nov 1998.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms*, 2001.

[21] V. Scarlata, B.N. Levine, and C. Shields. Responder anonymity and anonymous peer-to-peer file sharing. In *ICNP*, Nov 2001.

[22] A. Singh and L. Liu. TrustMe: Anonymous management of trust relationships in decentralized p2p systems. In *Proceedings of IEEE International Conference on P2P Computing*, 2003.

[23] A. Singh, M. Srivatsa, L. Liu, and T. Miller. Apoidea: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web. *Lecture Notes in Computer Science*, 2924, 2004.

[24] E. Sit and R. Morris. Security considerations for peer-to -peer distributed hash tables. In *Proceedings of IPTPS*, March 2002.

[25] P.F. Syverson, D.M. Goldschlag, and M.G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 44–53, 1997.

[26] L. Xiao, Z. Xu, and X. Zhang. Low-cost and reliable mutual anonymity protocols in peer-to-peer networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):829–840, Sept 2003.