4P: Performant Private Peer-to-Peer File Sharing

Niels Zeilemaker, Johan Pouwelse, and Henk Sips
Delft University of Technology, The Netherlands
niels@zeilemaker.nl

***Abstract*—In recent years fully decentralized file sharing systems were developed aimed at improving anonymity among their users. These systems provide typical file sharing features such as searching for and downloading files. However, elaborate schemes originally aimed at improving anonymity cause partial keyword matching to be virtually impossible, or introduce a substantial bandwidth overhead.**

In this paper we introduce 4P, a system that provides users with anonymous search on top of a semantic overlay. The semantic overlay allows users to efficiently locate files using partial keyword matching, without having to resort to an expensive flooding operation. Included into 4P are a number of privacy enhancing features such as probabilistic query forwarding, path uncertainty, caching, and encrypted links. Moreover, we integrate a content retrieval channel into our protocol allowing users to start downloading a file from multiple sources immediately without requiring all intermediate nodes to cache a complete copy.

Using a trace-based dataset, we mimic a real-world query workload and show the cost and performance of search using six overlay configurations, comparing random, semantic, Gnutella, RetroShare, and OneSwarm to 4P. The state-of-the-art flooding based alternatives required approximately 10,000 messages to be sent per query, in contrast 4P only required 313. Showing that while flooding can achieve a high recall (more than 85% in our experiments) it is prohibitively expensive. With 4P we achieve a recall of 76% at a considerable reduction in messages sent.

I. Introduction

In recent years, a number of fully decentralized file sharing systems were developed aimed at providing users with improved anonymity while searching and downloading files. Examples of such systems are Freenet [6] and Gnunet [5] which were released in the early 2000s, while RetroShare [13] and OneSwarm [9] are more recent solutions. These systems rely heavily on encryption techniques in order to provide users with the best possible preservation of publisher, sender, and receiver anonymity.

However, the increase in complexity of these solutions has given rise to substantial drawbacks, either due to their P2P overlay structure or the approach used to search for files. E.g. in order to route search queries both Gnunet and Freenet hash the keywords, thereby unintentionally prohibiting partial keyword matching. Similarly, flooding induced search overhead found in Gnunet, RetroShare, and OneSwarm prohibits those systems to scale as their costs rise exponentially compared to the network size.

In 4P, we build upon a semantic overlay which has shown to yield a high recall at a fraction of the cost [18]. Additionally, we combine the semantic overlay with privacy enhancing features which protect publisher, sender, and receiver anonymity from collaborating peers.

Contributions In this paper we present 4P, a fully decentralized private file sharing system which introduces a trade-off between overhead and recall. 4P requires substantially fewer messages to be sent per query compared to flooding based alternatives, at a marginally lower recall rate.

Our main contributions in this paper are:

1) Leveraging semantic similarity among users, to query for files without resorting to flooding.
2) We introduce a method which is able to negotiate a session-key between two peers without the need of a PKI, while still being able to overcome a man-in-the-middle.
3) Employing probabilistic query forwarding, path uncertainty, caching, and encrypted links we preserve publisher, sender, and receiver anonymity from collaborating peers.
4) Finally, we present the design, implementation, and evaluation of 4P using a trace-based dataset mimicking a real-world query workload.

4P is built on top of previous work [21] which allows us to construct a semantic overlay in a privacy preserving manner. By employing homomorphic encryption techniques, peers in our network exchange preferences and compute their overlap in the encrypted domain. This results in both peers knowing how many preferences overlap, without actually knowing which. Moreover, a peer replying to a request to compute the overlap cannot guess the contents of the request, and therefore cannot fake similarity. After finding and connecting to semantically similar peers, 4P uses the semantic overlay to achieve a high recall while incurring a low overhead. Our experiments show that peers achieve an average recall of 76% while requiring only 313 messages to be sent, substantially less than the approximately 10,000 messages sent by current systems.

Additionally, as 4P is not hashing or encrypting keywords during search, we can implement true fuzzy search. This alleviates users from determining beforehand by which keywords their files need to be found, a requirement in both Freenet and Gnunet. Moreover, partial keyword matching allows us to implement content based search.

II. Related Work

Searching in a Peer-to-Peer (P2P) network without the use of centralized components has been widely researched over the last decade. However, it remains a very difficult problem to this day. Adding privacy preserving features to the mix has increased the complexity of the proposed systems, while at

978-1-4799-6201-3/14/31.00

the same time substantially decreased their efficiency. In this section we will describe the current state-of-the-art in privacy preserving P2P file sharing.

One of the first fully decentralized and unstructured P2P network providing search is Gnutella. In the Gnutella network [1], [15], peers connect to each other in an unstructured manner. At the same time no single peer is more important than another one. Search is implemented by creating a query message, which a peer sends to all of its neighbors. This query message includes a *time to live* (TTL) value, which prevents a single query from being forwarded more than TTL times. Upon receiving a query message a peer reduces the TTL value by 1, and forwards it to all its neighbors while the updated TTL is greater than 0. Sending messages in this manner is called flooding and causes all peers within TTL hops to receive the query.

Freenet, introduced in 1999, aims at being a more privacy oriented alternative to then existing filesharing systems [6]. It uses an unstructured network in which every peer and every file is assigned to a key. Files can be found by generating a key, using one of three different key-schemes. The key-schemes provide peers with increasingly more features, such as claiming their own keyspace, and storing files in encrypted form. However, they also make finding files increasingly more difficult as peers need to know public keys and or passwords beforehand.

In Freenet, peers forward a query to a single neighbor based on the XOR distance between the keyword hash and the keys of its neighbors. If a peer exhausts its neighbors and cannot forward a query, it will send a request-failed message telling the previous peer to forward the message to another peer. If a file was found, a SendData message is sent to the peer which claimed to be issuing the query. However, as any peer in the *search-path* (the chain of peers a query traverses over) can probabilistically rewrite the source identifier of the requesting peer, the SendData message will be forwarded over some hops. At each hop, the content of the SendData message is cached locally. This causes popular content to be cached in the network while unpopular content is dropped.

Sending a query in Freenet using a property called *depth* to record how many times a query is forwarded. However, this depth value is not initialized at 0, but is set probabilistically to a small value. Hence, the first peer receiving the query cannot automatically assume that the peer it received the message from is the initiator of the query. In contrast, in Gnutella a peer receiving a message with TTL set to 6 can immediately deduce that the predecessor created the query [19]. Additionally, instead of sending a reply, a peer will forward a query with a TTL value of 1 with a finite probability, extending the path and obscuring the total path-length from all peers.

Gnunet aims at providing users with a reliable anonymous backup system [5]. However, in contrast to Freenet, Gnunet uses a flooding-variant similar to Gnutella. In their protocol, the initiator of a query has to specify a priority and TTL value, together with a triple hashed keyword using the RIPE160 hash algorithm. When receiving a query, a peer will use the priority value to charge for the service of handling the query by subtracting the priority from the credit of the sender. If a query is received with a priority value of 0, no charge for the service can be applied and thus the query should only be honored if a peer has excess bandwidth. Content that matches the hashed keyword will be sent back in a reply. If a peer does not have matching content, it will forward the query to n neighbors selected at random if the TTL value is higher than 0. The priority value for each of the forwarded messages is reduced to $\frac{p}{n+1}$. The number of neighbors selected (n) depends on the system load.

A query in Gnunet contains a triple hashed keyword, data is stored at a double hashed keyword. This makes it is less likely that a peer can reply to a query without having the actual data, as the properties of the chosen hash function makes it computationally intractable to deduce the double hashed keyword from the triple hashed keyword.

Files in Gnunet are split into pieces, which can be downloaded by traversing a tree. A reply to a query consists of a 1Kbyte *root-node* containing a pointer to the actual data (the root of the tree), a description, a CRC checksum, and a optional pseudonym and signature. For each keyword by which a file should be able to be found, a root-node is created. Using the associated keyword, every root-node is encrypted, thus preventing a peer storing a root-node from knowing what it is storing.

RetroShare is a *friend-to-friend* (F2F) network which aims at providing users with a private social network [13]. On top of this social network, in which users have to manually exchange their public keys in order to become friends, features such as instant messaging, group chat, voice over IP, and file sharing are built. Users can find and download files using the TurtleRouter [14]. This component of RetroShare provides users with anonymous multi-hop file transfer, performed over encrypted tunnels.

Searching for a file is called *digging for a tunnel*, and peers dig a tunnel by sending a tunnel request to all their neighbors. This tunnel request message contains a file hash, a depth (comparable to the inverse of a TTL value), a randomly generated ID, and a half-tunnel ID. Requests are forwarded to all neighbors while the depth field is lower than 6, but cycles are avoided. After locating a file, an encrypted tunnel is established using the hash of the file hash, source peer ID, and destination peer ID.

In 2010, OneSwarm [9] took an approach similar to RetroShare and extended BitTorrent with a F2F network. In OneSwarm, peers can decide locally what data is shared with whom. Using a combination of private and public files, they achieve a higher download speed compared to Tor and Freenet. Exchanging public keys with friends is somewhat easier in OneSwarm, as they *piggy-back* on existing social networks.

All traffic between peers is encrypted using SSLv3. Searching for files uses flooding, which does not have a TTL value and hence is unconstrained. Only a Bloom filter prevents peers from forwarding a single query more than once. If a peer has found enough sources, it may cancel a query by sending a

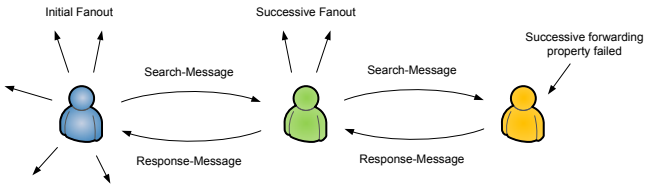

Fig. 1. Query being forwarded across a search-path, highlighting the search properties.

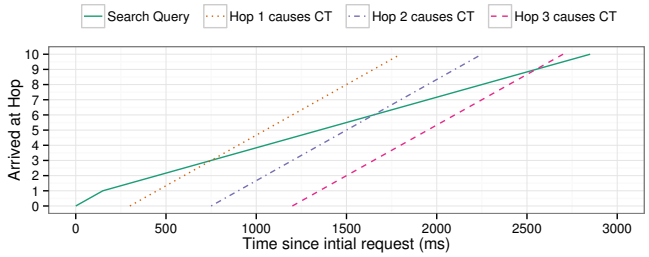

Fig. 2. OneSwarm cancel mechanism, if a peer in hop 2 causes the CT, the cancel message catches up with the query at the 6th hop.

cancel-message. OneSwarm delays forwarding a query for 150 ms at each hop, to allow for the cancel-message to catch up if necessary. Messages forwarded to untrusted peers are delayed with an additional 150-300 ms to obfuscate the path-length. After a file is found, a BitTorrent transfer can be initiated over the same path as the search. To allow for this search results include a path-id variable.

III. Cost and Limitations of Current Systems

All five related works implement a file sharing system on top of an unstructured network. In order to compare the approaches used, we show a generalization of their search properties in Figure 1. The *initial fanout* and *successive fanout* properties determine how many neighbors the query is send/forwarded to. The *successive forwarding property* defines when a message should stop being forwarded. Table I shows the search approaches of the previously mentioned systems using the properties introduced in Figure 1.

A. Bandwidth Cost

We can distinguish between two different approaches to search in an unstructured network. First, a P2P network can employ routing to find the peer which is the most likely to have matching files. This approach is implemented by Freenet, but is more commonly found in DHTs. Second, a P2P network can employ (constrained) flooding. Here each peer forwards a query until the forwarding condition is not satisfied anymore. This approach is implemented by Gnutella, Gnunet, RetroShare, and OneSwarm.

A big drawback of routing approaches is that partial keyword matching becomes almost impossible to implement. A search query is routed to a particular peer based on the hash of the keyword, therefore partial keyword matching requires multiple queries as they must be routed to different peers. Moreover, for each keyword (or partial keyword), a peer inserting a file must announce its availability. These announces often consume large amounts of bandwidth in DHTs, as they have to be repeated frequently due to churn [17]. *Churn* denotes the frequent leaving and joining of new peers in the overlay, which causes information stored in the DHT to be stale. In Freenet, the situation is worse as complete files have to be stored at the peer which is responsible for a keyword, e.g. in order to implement partial keyword matching multiple copies of a single file have to be stored in the network.

However, flooding a network can cause even greater bandwidth usage. In 2000, Gnutella used a default TTL value of 7 for queries, causing 95% of the peers in the network to receive each query message. Ripeanu et al. [16] discovered that flooding the network with query messages caused Gnutella to consume roughly 330 TBytes per month excluding file-transfers in a relatively small network consisting of 50,000 peers. Using a similar flooding approach to Gnutella, RetroShare employs a TTL of 6 when digging for a tunnel, leading us to believe that it has comparable bandwidth usage.

OneSwarm constrains its flooding in a different manner. It does not incorporate a TTL value, but delays query messages at each hop, allowing cancel messages sent by the issuing peer to catch up with the flood and stop it after enough sources are found. Peers cancel their query after 10 sources have been found. For simplicity, we call the peer which sends this 10th reply the *cancel trigger* (CT).

Figure 2 shows the time it takes for a query and cancel message to arrive at a certain hop, depending on which hop sends the CT. From the figure we can see that if hop 2 causes the CT, the reply will arrive at the initial peer after 750 ms (using 150 ms for RTT, and delaying 150 ms at each hop before forwarding). When the issuing peer then immediately sends the cancel message, it catches up with the flood at the 6th hop (1650 ms after the initial query). This is similar to using a TTL of 6 and hence not a scalable solution. [1] [2]

Out of all of the introduced related works, Gnunet has constrained its flooding the most; reducing the number of neighbors contacted based on load, charging peers for forwarding a query, and using a TTL value to limit the number of times a query is forwarded. However, Gnunet requires many queries to be performed in order to download a file, as it needs to traverse the tree. The actual number of request can be computed using the following equation wherein P equals the number of initial pieces (filesize divided by 1024 bytes), as described in [5].

$$RequiredQueries = P + \sum_{i=0}^{\lfloor \log_{51}(P) \rfloor} 51^i$$

[1] Please note that a higher RTT will increase the time it takes for the cancel message to catch up, and a lower RTT will the reduce it.

[2] Both the number of sources required to cancel a search, and forward delay are specified in the OneSwarm paper [9]. However in the current implementation, the number of sources required to cancel a search is increased to 40 https://github.com/CSEMike/OneSwarm/blob/88556421a7572565c882c6fcc7c5858cf55eabed/oneswarm_az_mods/mods_constants/org/gudy/azureus2/core3/config/impl/ConfigurationDefaults.java (f2f_search_max_paths).

TABLE I
SEARCH APPROACHES IMPLEMENTED BY CURRENT SYSTEMS

	Approach	Keywords	Initial fanout	Successive fanout	Initial Fwd property	Successive Fwd property	Fwd remarks
Freenet	Routing	Hashed	1	1	TTL = 18 - small random value	TTL > 0	Probabilistically extend TTL = 1
Gnutella	Flooding	Plaintext	10	10	TTL = 7	TTL > 0	-
RetroShare	Flooding	Plaintext	10	10	TTL = 6	TTL > 0	-
OneSwarm	Flooding	Plaintext	0.5 * #Friends + System load	0.5 * #Friends + System load	-	Not seen yet	-
Gnunet	Flooding	Hashed	System load	System load	TTL = 7	TTL > 0	-

If a peer wants to download a 1 GByte file, it is required to perform 1,000,052 queries. In order to have an estimate of the bandwidth required to perform this many queries, we assume that a single query requires 28 bytes to be send over the network (20 bytes hash, and two integers: TTL and priority). If we additionally assume that, by constraining its flooding, Gnunet manages to send a query to only 5% of all peers, sending 1,000,052 queries would require 65 GBytes of overhead traffic in a network consisting of 50,000 peers. In "version 2.0" of their protocol, Gnunet uses 32 Kbyte pieces resulting in 32,770 queries and 2 GBytes of overhead traffic [4].

B. Limitations

Besides bandwidth cost, both Gnunet and Freenet are unable to provide partial keyword matching. Gnunet uses triple hashed keywords in order to prevent intermediate nodes from being able to decipher the query. Thereby making it impossible to perform partial keyword matching, as partial hashes cannot be constructed.

Moreover, all three key-schemes implemented by Freenet suffer from the same problem, but two of those schemes (SSK and CSK) additionally require users to know both the public key of the user inserting a file and its keywords before being able to locate them. Although this solves the spam problem which renders the KSK scheme useless, it also causes locating files to be virtually impossible without an additional index.

IV. 4P DESIGN

Another approach to search in an unstructured network is to cluster peers according to their preferences. *Preferences* can be defined as the files a user downloaded, ratings for movies, etc. A network which is clustering peers according to their preferences is called a semantic network or overlay, and previous work [18] has shown that by organising peers in such a manner, recall can be high without flooding the network. Typical values for a semantic overlay include an initial fanout of 10 and a successive fanout of 0. Therefore in a semantic overlay, as opposed to routing or flooding, a peer searching for a file sends a query to its similar neighbors which then reply, i.e. a query is not forwarded limiting the privacy exposure of a peer. However, similar to Gnutella, neighbors in a semantic overlay still know from which peer the query originated.

4P improves upon a traditional semantic overlay by implementing three features which allow us to preserve the anonymity of both the sender and the receiver of a query. Using a similar approach as Freenet, peers probabilistically forward query messages. Initially, a TTL value is specified by the initiator of the query. This TTL value is decremented at each hop, and when it reaches 0 a reply message is sent back. Additionally, the TTL value of a message is used to determine to how many peers the message is forwarded. If a peer receives a message with a TTL value of 5, it decrements it to 4 and then forwards it to 4 peers. We define the max/initial TTL value as the agreed TTL value used by all peers when creating a new query message.

Moreover, a peer can probabilistically decide not to decrement the TTL of a message with the agreed upon max/initial TTL value. This prevents peers receiving a message with the max/initial TTL value from knowing if the previous peer has created or forwarded the query. We define this probability as the *initial extension probability* or IEP. Finally, a peer can probabilistically decide to keep forwarding a query with a TTL value of 1, preventing an attacker from being able to query a single peer. We define this probability as the *final extension probability* or FEP.

To protect the privacy of the peers replying to a query we use caching. When receiving a reply-message from another peer all results within it are cached locally. This achieves two goals, first the cache of a peer now contains not only its own data, but also cached results from others. Consequently, if after enabling caching, an attacker tries to determine the contents of a peer's local cache he will not be able to distinguish between cached and local results. Second, as we have built 4P on top of a semantic overlay the cached results will likely be relevant to a peer and its neighbors, thus improving overall recall of the system.

However, in contrast to Freenet, peers in 4P do not cache complete files by default. In 4P, we split searching for and downloading files into two separate actions. First, the peer issuing a query will receive a message containing all metadata found by peers on the search-path. Second, after deciding upon which content to download, a peer can use the established content-path to start downloading it. The *content-path* is a tunnel from the initiator of a query to a peer which has the content. A content-path is constructed for each result a peer back-propagates to the original query initiator. During the search, each peer in the search-path will rewrite the source IP protecting the identity of the peer it received the query from,

and hence a tunnel is required to download a file. While a content-path is active, the metadata associated to it is cached. Content-paths are kept *active* by sending ping/pong messages to the next hop. As long as this peer replies, the content-path is marked as active. An active content-path, allows peers to use the locally cached metadata while replying to other queries. If a content-path breaks, a peer removes the metadata from its cache as it is unable to provide other peers with the actual contents of the file.

V. 4P Implementation

Based on our design, we need to build three components in order to implement 4P. First, we provide a method which is able to construct a semantic overlay in a privacy preserving manner. Second, we define the messages used when sending a query. Third, we describe how a peer can download files after finding them. In this section we elaborate on the actual implementation of these components.

A. Semantic Overlay

In order to construct a semantic overlay in a privacy preserving manner we have modified our earlier work [21]. In this protocol, peers are able to discover and connect to similar peers without having to reveal their preferences/items. By applying homomorphic encryption, two peers can compute the overlap between their preferences sets without disclosing which preferences do overlap, and which don't. Our extension in this work is a method which allows Bob to reply with a session-key which can be used to encrypt the link without being vulnerable to a man-in-the-middle attack.

For convenience we list the basic steps of the protocol below:

1) Alice builds a polynomial having roots in each of the items contained in her set $\mathcal{I}_A$. That is, she computes the $n+1$ coefficients $\alpha_0, \ldots, \alpha_n$ of the polynomial
$$f(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_n x^n \qquad (1)$$
for which $f(\mathcal{I}_{A,i}) = 0$ for any item in her set.
2) Next, Alice encrypts the coefficients and sends them to Bob, e.g. sending $\mathcal{E}_{pk_A}(f(x))$.
3) Bob uses the homomorphic properties of the encryption scheme to evaluate the polynomial for each item in his set Γ_B. He multiplies each result by a fresh random number r_i, and adds a session-key generated for Alice obtaining $\mathcal{E}_{pk_A}(r_i \cdot f(\mathcal{I}_{B,i}) + \mathcal{S}_{A,B})$.
4) Bob adds all evaluated polynomials to a list, permutes the order, and sends it back to Alice.
5) After receiving the list of evaluated polynomials from Bob, Alice decrypts each ciphertext. The ciphertexts decrypt to $\mathcal{S}_{A,B}$ in case of an item in the intersection $\mathcal{I}_A \cap \mathcal{I}_B$, or to a random value otherwise.

Alice can detect $\mathcal{S}_{A,B}$, if more than one item overlap between her and Bob. Moreover, it is very likely that the random values are larger than the 128 bit key we're using for $\mathcal{S}_{A,B}$. Hence, if Alice finds a single 128 bit value in Bob's decrypted list, she assumes it is the session key. By counting the occurrences of the session-key, Alice can compute $sim_{A,B}$

The session-key generated by Bob cannot be intercepted by any man-in-the-middle, as its encrypted with the key of Alice. Moreover, it is intractable to decrypt either the encrypted coefficients of the polynomial, or the encrypted evaluated polynomials, as they are encrypted using the Paillier scheme [11].

In step 4, Bob permutes the order of the list to hide the index of the items which are overlapping. Without this permutation, the index of the overlapping items can reveal information on when Bob added this particular item to his preference list, or even worse if his preference list is sorted it could reveal the distribution of the values in the list.

Furthermore, please note that Bob cannot simply reply with all preferences, as the list of possible preferences is very big (in our implementation we use a hash function to limit the size to 2^{40}), and Alice can easily detect this as Bob will reply with many evaluated polynomials.

Finally, in order to prevent an attack in which Alice will attempt to discover the preferences of Bob by repeatedly running the protocol with only one preference, Alice is required to append a proof-of-work to each request. This proof-of-work can only be used once, and requires Alice to spend a considerable amount of time for each request. An example of such a proof-of-work is HashCash [2], wherein a user sending an email is required to compute a hash based on the current time, destination address, and a random seed. This hash needs to start with at least 20 zero's, and can only be computed brute forcing different random seeds.

B. Search Messages

In the semantic overlay, peers use two messages while searching for files. Initially, a peer sends a *Search-Message* consisting of 3 fields. First an identifier, this is a randomly chosen 2 byte unsigned short. The identifier is used to detect cycles in the search-path. Next, a 1 byte unsigned TTL value used by peers to determine if this message needs to be forwarded. Finally, the message contains the keywords specified by the user, represented as a string with a dynamic size of at most 256 characters.

Upon receiving a Search-Message, a peer updates the TTL value. If the TTL value is greater than 0, the peer creates a new Search-Message using the same identifier and the updated TTL. This message is then forwarded to the next TTL peers, which are randomly chosen from the list of most similar peers.

If the updated TTL value equals 0, a peer replies with a Response-Message instead. To create this message, a peer will perform a local search and constructs a *Response-Message* with the same identifier. We only include the metadata of each result, e.g. the name of the file, a description, the size, etc. to reduce the size of the Response-Message. Additionally, each result has a unique identifier used by peers in order to download it.

Upon receiving a Response-Message, each peer in the search-path appends its local results and creates a new

Response-Message which it sends back to the peer it originally received the Search-Message from. This will eventually cause the Response-Message to back propagate to the initiating peer[3].

Moreover, each peer in the search-path sorts the results after appending its local results using the same relevance-ranking and only includes the Top-K most relevant results in the Response-Message. This aims to prevent peers from guessing the length of the search-path by counting the number of results. Choosing a K value is application dependent, setting it low will make guessing the path-length harder but also reduces the number of results per query.

If a peer receives a Search-Message with an identifier it has seen before (indicating a cycle), it still forwards the Search-Message. However, a peer will make sure that it does not forward a Search-Message with the same identifier to the same neighbor twice. When a peer exhausts all possible neighbors for a particular identifier, it aborts the search-path and replies with a Response-Message.

C. Downloading Content

As searching for and downloading a file are two separate actions, a peer has to be able to request to download a file after receiving a Response-Message. However, as described in the previous section, a Search-Message is sent to a random peer at each hop. Therefore, in order to satisfy a download request, the path to the peer which has the content has to be kept open, as attempting to send a request for the file will probably not end up at the source. This content-path is established by keeping open a connection to the peer it received the result from when back-propagating a Response-Message. When the initiating peer attempts to download the file its request is forwarded along this content-path, and the data is back-propagated.

In order to support downloading files from multiple sources we have modified the Peer-to-Peer Streaming Peer protocol (PPSP) [3]. At its core, PPSP can be compared to BitTorrent. It splits data into smaller pieces which then can be downloaded in parallel from multiple sources. However, in contrast to BitTorrent it uses UDP, a Merkle tree to both identify and verify pieces of the data, and multiplexes parallel downloads over a single socket. We modify PPSP such that intermediate nodes in the content-path will forward/back-propagate all PPSP messages. This tunnel allows the PPSP protocol to operate as normal. If a peer requires more sources for a download, it can perform an exact search using his remaining similar neighbors. If a hit is found, the new content-path is added to the PPSP download.

Moreover, the content-path of a file does not necessarily have to overlap with the search-path. If a peer has cached metadata from another request, then this cached metadata is kept while the content-path is active. Cached metadata is used while replying to queries, and hence successive searches

[3]Note that only the initiating peer knows that it was the initiator, all other peers cannot tell whether they are sending the Response-Message back to a forwarding peer or the initiator of the query.

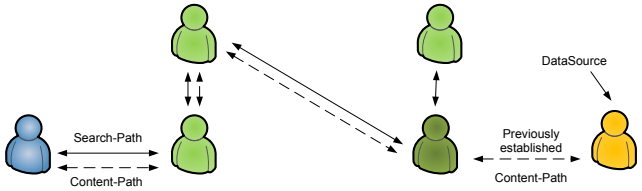

Fig. 3. Example of a search-path and an content-path. Note, that the DataSource is not actually queried during the search.

matching cached metadata are used to boost recall. Additionally, because we are using PPSP, we can keep multiple content-paths alive without incurring high costs as PPSP multiplexes traffic over a single socket (re-uses), and thus only one open socket is required for each of our similar neighbors.

Figure 3 gives an example of a search-path and the established content-path. While performing this search, the DataSource of the matched file did not actually receive the query. However, a peer caching the metadata of the file used the previously established content-path in a reply.

Downloading a file in this manner causes each peer in the path to download/upload part of a file. This is clear drawback, but it will preserve the privacy of both the downloader and uploader. If privacy is less of a concern, then by enabling *peer exchange* (PEX) in the PPSP protocol a peer can decide to (partially) collapse the path and thereby reducing the number of hops between it and the DataSource. Not rewriting the source address is implemented in both Freenet and Gnunet. However it has been shown to be exploitable by a shortcut attack in Gnunet allowing an attacker to determine the initiator of a download request [10].

D. Choosing the initial TTL, IEP, and FEP

Searching for files in 4P, resembles very much (constrained) flooding. As flooding is something we definitely want to avoid, the initial TTL, IEP, and FEP need to be chosen carefully as they might cause 4P to behave similar to flooding, i.e. a combination of these values needs to be found which limits the number of messages sent.

Please note that we can constrain the flooding much more because of the semantic overlay we're searching on top of, the semantic overlay does not need many messages to be sent as the peers which have the best results are nearby.

Whether 4P will flood the network is determined by the combination of IEP and the chosen initial TTL. These two variables determine the probability that one of the peers extended the initial path. If this probability is too high, larger than 0.5, the initial path extension is unlikely to stop. Hence causing flooding of the network. We can compute the probability of at least one peer extending the path (PIE) as follows:

$$PIE = 1 - (1 - IEP)^{\text{initial TTL}}$$

Empirically we found that if we choose the initial TTL and IEP such that the PIE stays below 0.5, then no flooding occurs. However, by lowering the IEP, and thereby being able to increase the initial TTL, the initial path extension is reduced.

TABLE II
ANONYMITY EVALUATION

Attacker	Sender Anonymity	Receiver Anonymity
Local eavesdropper	Provably exposed	Provably exposed
Collaborating peers	Probably innocent	Probably innocent

Causing less uncertainty on whether or not the previous peer created the query. The expected value of the initial extension can be computed using:

$$IEV = \sum_{i=1}^{\infty} i \times PIE^i$$

In our experiments, we have set the initial TTL to 4, IEP to 0.15, and the FEP to 0.5. This results in a PIE of 0.45, and a IEV of 1.5.

VI. SECURITY

In this section we briefly discuss possible attacks, and the extend to which 4P protects the anonymity of peers. In the original Freenet paper [6], the authors evaluate their protocol by considering sender and receiver anonymity. *Sender anonymity* defines the extent to which the identity of an initiator of a query is protected, *receiver anonymity* as the extent to which the identity of a peer which replied to a query is protected. The levels of protecting anonymity were described in a paper by Reiter et. al. [12], and range from *absolute privacy* in which communication cannot be observed to *provably exposed* in which an adversary can prove that a peer was the initiator. Please note that being able to prove that a peer was the initiator does not necessarily mean that the attacker can read the query.

The anonymity of both sender and receiver is exposed in Freenet, when attacked by either a local eavesdropper or collaborating peers. However, sender anonymity is preserved *beyond suspicion* for collaborating peers.

Table II shows a summary of the anonymity of 4P. Against a local eavesdropper 4P cannot provide any anonymity. We defined a local eavesdropper, as an ISP which can observe all communication in which the computer of a peer participates. Therefore, it can detect that a peer has created a new query, as it can observe that no query was sent to the peer beforehand, and detect that a peer replied to a query, as it might be able to observe new results in a Response-Message due to the increase in size. Encrypting, padding, and combining messages can improve anonymity while dealing with a local eavesdropper as it can prevent the eavesdropper from detecting new queries or new responses.

Against collaborating peers, 4P is better equipped. The sender anonymity of a peer is protected as a query with the initial TTL value is probabilistically forwarded. Additionally, receiver anonymity is protected as every peer in the search-path will always forward a query until TTL = 0. Therefore, no peer can determine if a reply originated from the peer it just received the Response-Message from or if it originated from a peer higher up the path. Moreover, as peers may forward a query with TTL = 1, an attacker is not able to use TTL = 1 queries in order to determine the actual files being shared by a peer. Finally, all links are encrypted using the session-keys peers exchanged during the construction of the semantic overlay, ensuring that no man-in-the-middle can listen-in on the queries/message sent by a peer.

VII. EXPERIMENTS

In order to evaluate the performance of 4P, we have designed several experiments which compare the recall performance and associated cost to several other overlay configurations and their search approach. In the following subsections we describe the dataset, emulation setup, and the metrics we used during the evaluation.

A. Dataset

During the experiments we used a dataset extracted from a deployed semantic overlay network [20]. In this P2P network peers create a semantic overlay by exchanging plaintext messages containing all preferences of a peer. By comparing the preferences of it and other peers, a peer is able to compute which other peers are similar to it. By deploying instrumented clients into the semantic overlay we have created dataset consisting of the preferences of more than 75,000 peers.

From this dataset, we selected 1000 peers at random and made sure that each peer has at least 10 preferences. Additionally, we require that each item was preferred by more than one peer (all other items were removed from the dataset). Next, we have split the preferences of each peer in a training/testing set using a 80%-20% split. Dividing a dataset in this manner allows each peer to use its training set to find similar peers, and then use the testing set to check if those peers are similar enough to provide it with search results. E.g. we issue a query for each item in the testset, and see if we can find it using the search algorithm being tested. A 80%-20% split of training and testing set is common practice in collaborative filtering and used to asses the quality of the discovered similar neighbors [7], [8]. Finally, we computed the 10 most similar peers for each peer using its training set.

B. Emulation setup

In order to emulate the semantic overlay we implemented the protocol as described in Section V-A on top of ANON [22]. ANON is a P2P data replication platform which implements a semi-random peer selection algorithm, NAT-traversal, and fully decentralized permissions. Building the overlay on top of ANON allows us to easily deploy and test the system both on our supercomputer and in the future, deploy it to the Internet.

We create a process for each peer and configure it with its own separate database containing its own training and testing set, and the 10 most similar peers it must connect to. In each scenario we deploy 1000 peers onto 10 nodes of a supercomputer, and force them to connect to their most similar

TABLE III
SEARCH STRATEGIES, AS EVALUATED

	Overlay	Initial fanout	Successive fanout	Forwarding condition
Random	Random	10	0	-
Semantic	Semantic	10	0	-
Gnutella	Random	10	10	TTL > 0
RetroShare	Semantic	10	10	TTL > 0
OneSwarm	Semantic	0.5 $\times$ #Friends	0.5 $\times$ #Friends	Not seen yet
4P	Semantic	TTL	TTL	TTL > 0

peers[4]. Next, 200 peers use their test set (20% of their data) to send queries. Using the random identifiers of all queries we track to which peers a query is sent, how many peers in total receive a query, at what point in time they receive it, and when the initial peer receives a reply. If two peers during the experiment use the same identifier we remove all data regarding those queries, to prevent possible misinterpretation of the results.

During the experiments each peer writes its recall to a file, allowing us to compute the overall mean recall of those peers and thus the search-performance. Recall is computed as follows:

$$Recall = \frac{\#\text{found items}}{\#\text{queries sent}}$$

C. Evaluated strategies

In order to compare the performance of 4P, we implemented and evaluated several other strategies derived from the related work as shown in Table III. We have chosen to evaluate a random strategy to show the gain of using a semantic overlay. Furthermore, we implemented the flooding strategy of Gnutella, in order to show the performance gain and higher bandwidth cost of such an approach. Note that we actually limit the flooding scenario as we "only" forward a query to 10 neighbors.

Finally, we compare the approaches of RetroShare and OneSwarm to 4P using the same semantic overlay (i.e. each peer connects to its 10 most similar neighbors). RetroShare floods the network similar to Gnutella, however it uses a slightly lower TTL. For OneSwarm, we converted their latest published Java code to Python and integrated it into our experiment setup. Moreover, we used all their defaults in order to constrain the flooding of the network. We configured 4P using the values as described in Section V-D.

D. Results

Figure 4 shows the mean recall of the evaluated strategies. From the figure it is clear that having a semantic overlay greatly improves recall over simply using a random overlay. A random overlay only obtains a recall of 9%, clustering peers according to their preferences results in 40% recall. However, we note that building and maintaining a semantic overlay is not free. During our experiment a single handshake (used to find your most similar peers) using our Paillier-based algorithm costs roughly 79.5 Kbytes on average.

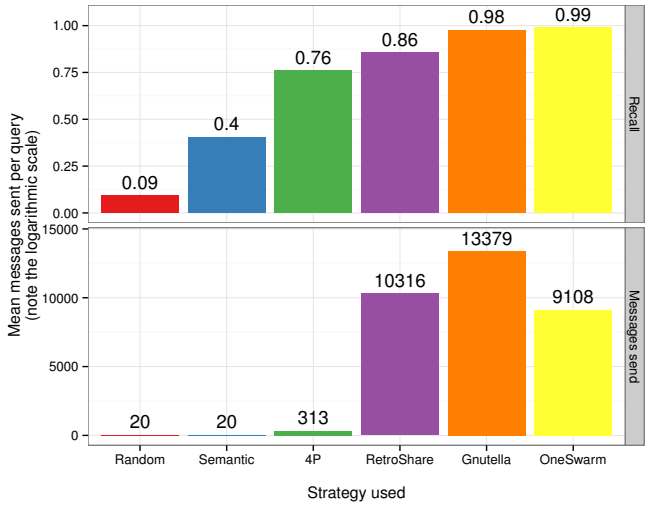

Fig. 4. Recall and Messages send of evaluated strategies

Next, using 4P improves recall to 76%. This improvement can be explained by the increased number of messages being sent due to fowarding the request. We believe that the combination of a semantic overlay and our very transparent method for caching yields the increase in performance.

The flooding strategies of RetroShare, Gnutella, and OneSwarm show that indeed if only recall is considered, flooding a network results in the best performance. RetroShare achieves a recall of 86%, Gnutella 98% and even more impressive OneSwarm achieves a recall of 99%. Altough this is somewhat expected as OneSwarm does not incorporate any constraints on the depth of the flooding.

However, if we look at the number of messages send we can see that flooding has one very significant drawback: it requires many messages to be sent in order to achieve the recall performance. During the experiment, all three flooding strategies required roughly 10,000 messages to be sent. RetroShare requires less messages than Gnutella (due to having a TTL of 6, instead of 7), but not less than OneSwarm. Although OneSwarm does not use a TTL value to constrain the flooding, it does manage to require less messages per query. We believe this is mostly caused by a peer forwarding a query to only half of its friends, and the limit on outgoing searches (max 300 per minute). We note that in this configuration, OneSwarm never cancels its requests as a peer does not receive enough replies. After we lowered the number of replies received to 1, the number of messages sent was reduced to 6,200.

4P requires only 313 messages to be sent. These numbers include the replies of all peers, hence only 157 SearchMessages are sent. We do not expect more messages to be sent whenever the size and/or topology of the network changes. This is because the number of messages sent is influenced by the initial TTL value, the IEP and the FEP, and not by the

[4]By doing this we achieve a performance which does not reflect the actual performance when deployed, as the speed of discovering similar peers, dealing with churn, etc. are not taken into account. However, these factors influence all systems we compare negatively, and both RetroShare and OneSwarm additionally rely on specific friends being online to provide a peer with search results (not depending only on those peers which have a similar preferences).

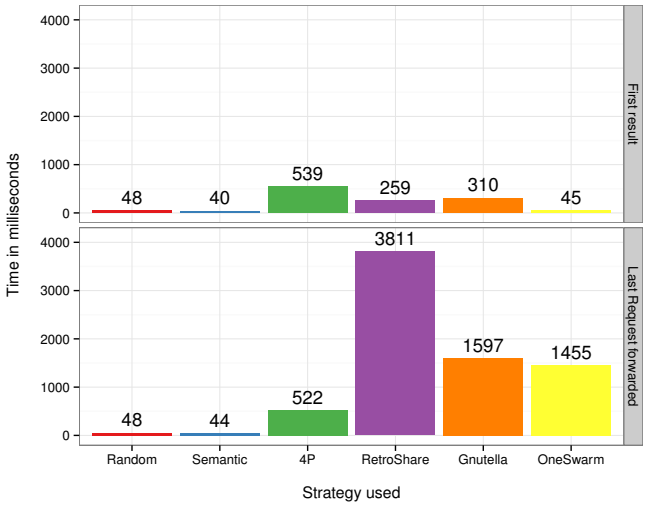

Fig. 5. First response received, and last request forwarded during a search.

size of the network. In contrast, highly connected friends in the RetroShare and OneSwarm networks influence to how many peers a query will be forwarded, as they both do not limit the number of friends a request is forwarded to (but limit it to 0.5 of friends). The semantic strategies only require 20 messages to be sent (10 requests and 10 replies).

If we consider both the recall and bandwidth cost of the evaluated strategies, we believe that 4P clearly improves upon current state of the art flooding based private file sharing systems.

Figure 5 shows the time it takes for the initial peer to receive the first result, and when its Search-Message was last forwarded. Please note that the RTT of our supercomputer is very low, hence we get very low response times for all protocols and searches for which we do not get a response are not used when computing the mean.

Considering the first received result random, semantic, and OneSwarm average below 50 ms, which for the first two is not surprising. Those strategies consist of a single request/reply which should be very quick. However, the performance of OneSwarm is surprising as we expected it to perform equal to RetroShare. As these protocols reply immediately whenever a result is found, and both run on top of the same semantic overlay. We believe that the difference is caused by the larger fanout of RetroShare, resulting in a higher load on the nodes as they receive more messages in a smaller time frame. Looking at the CPU consumption of both protocols during these runs seem to be consistent with this, RetroShare requiring roughly twice as much CPU-time than OneSwarm. The CPU-time required by Gnutella is even higher (almost three times as much as OneSwarm) and hence is causing the first result to arrive after 310 ms. 4P performs the worst, but this is due to all peers waiting for a reply of their neighbors before back-propagating all results to the initial peer.

Considering the last request forwarded durations, 4P performs as expected. Peers only add their replies when back-propagating a request, and hence slightly lower numbers for when the last request was forwarded are expected. Furthermore, both random and semantic perform as expected showing roughly the same numbers as when the first reply was received.

For the flooding approaches, all three strategies continue to forward a request even after the first reply has arrived. This is as expected, however RetroShare is on average still forwarding a request almost 4 seconds after the initial peer created it. We expect that this is caused by two factors: first the higher fanout is overloading the nodes, second due to the semantic overlay some peers get much more requests than others. These nodes are similar to many peers (due to downloading popular files) and hence get more requests compared to other peers do not download these. Overloading the popular peers creates a back-log which slows down the forwarding process. Although Gnutella causes an even higher average CPU load, its random overlay does not cause some peers to be overloaded (as we expect all peers to have a similar in-degree), and hence the last request is forwarded after 1.6 seconds. OneSwarm performs similarly to Gnutella, and is helped by the fact that it causes a lower CPU load. Moreover, OneSwarm has a lower fanout which reduces the load on the popular peers and hence does not overload them as we have seen in RetroShare.

VIII. Conclusion

In this paper we have shown that it is possible to create a privacy preserving and performant file sharing system which in contrast to current systems does not flood the entire network. Moreover, since we do not hash keywords, peers in the network can perform partial keyword matching, a feature that does not exists current private file sharing system such as Freenet and Gnunet.

After issuing a query, we provide peers with a list of results from which they can decide which file to download. Using the pre-established tunnels to one or more datasources of a file, downloading does not require new queries to be sent. If more than one datasource, or multiple paths to a single datasource are found then those paths can be used in parallel. This reduces the load imposed on intermediate nodes which otherwise are required transfer a complete file. Moreover, multiple paths to datasources help us bypass possible bottlenecks in the network, and thus improve overall download performance.

Thanks to extensive experimental evidence we have shown that 4P requires substantially fewer messages to be sent compared to a (constrained) flooding approach, while still achieving a mean recall of 76%. Moreover, in contrast to flooding the number of messages used by 4P during search is constant and does not scale exponentially with the network size.

References

[1] Gnutella Protocol Specification 0.6: http://rfc-gnutella.sourceforge.net .
[2] A. Back. Hashcash - a denial of service counter-measure. Technical report, 2002. available at http://www.cypherspace.org/hashcash/hashcash.pdf.
[3] A. Bakker, R. Petrocco, and V. Grishchenko. Peer-to-Peer Streaming Peer Protocol (PPSPP), Feb. 2013. available at http://datatracker.ietf.org/doc/draft-ietf-ppsp-peer-protocol.
[4] K. Bennett, C. Grothoff, T. Horozov, and J. T. Lindgren. An encoding for censorship-resistant sharing, 2003.

[5] K. Bennett, T. Stef, C. Grothoff, T. Horozov, and I. Patrascu. The gnet whitepaper. 06/2002 2002.
[6] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, volume 2009, pages 46–66. Springer Berlin Heidelberg, 2001.
[7] A. Demiriz. Enhancing product recommender systems on sparse binary data. *Data Mining and Knowledge Discovery*, 9(2):147–170, 2004.
[8] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.
[9] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Privacy-preserving p2p data sharing with oneswarm. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 111–122, New York, NY, USA, 2010. ACM.
[10] D. Kgler. An analysis of gnunet and the implications for anonymous, censorship-resistant networks. In *Privacy Enhancing Technologies*, volume 2760 of *Lecture Notes in Computer Science*, pages 161–176. Springer Berlin Heidelberg, 2003.
[11] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *LNCS*, pages 223–238. Springer, May 2-6, 1999.
[12] M. K. Reiter and A. D. Rubin. Anonymous web transactions with crowds. *Commun. ACM*, 42(2):32–48, Feb. 1999.
[13] RetroShare. Retroshare aims to be a private f2f social network, May 2010. available at http://sourceforge.net/blog/retroshare-aims-to-be-a-private-f2f-social-network/.
[14] RetroShare. Turtle router - retroshare, Oktober 2010. available at http://retroshare.sourceforge.net/wiki/index.php/Turtle_Router.
[15] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100, 2001.
[16] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, Jan. 2002.
[17] M. Steiner, W. Effelsberg, and T. En-najjary. Load reduction in the kad peer-to-peer system. In *In Fifth International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P*, 2007.
[18] S. Voulgaris and M. Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 1143–1152. Springer Berlin Heidelberg, 2005.
[19] R.-Y. Xiao. Survey on anonymity in unstructured peer-to-peer systems. *Journal of Computer Science and Technology*, 23:660–671, 2008.
[20] N. Zeilemaker, M. Capotă, A. Bakker, and J. Pouwelse. Tribler: P2P media search and sharing. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 739–742, New York, NY, USA, 2011. ACM.
[21] N. Zeilemaker, Z. Erkin, P. Palmieri, and J. Pouwelse. Building a privacy-preserving semantic overlay for peer-to-peer networks. In *Proceedings of the IEEE International Workshop on Information Forensics and Security (WIFS 2013)*, Guangzhou, China, November 2013.
[22] N. Zeilemaker, B. Schoon, and J. Pouwelse. Large-scale message synchronization in challenged networks. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*, Gyeongju, Korea, March 2014.