ABSTRACT

Peer-to-peer systems are being used for multiple purposes today ranging from file sharing, spam filtering, and commercial product evaluations among many. Only recently, has digital libraries been added to its use. Freelib is a peer-to-peer-based digital library that evolves its participants into communities. In this paper, we address the issue of node discovery in Freelib. Discovery in the context of Freelib refers to finding key information about peers such as their network addresses and the port numbers on which to communicate with them. Discovery is particularly important when nodes sign off from the network and join the network with different IP addresses. It is essential as it enables nodes to connect to their communities and previously known peers when they join back after being offline. We present three discovery protocols: Discovery by Flooding, DHT discovery, Link Discovery. These algorithms are not just useful for Freelib but can be adapted to other peer-to-peer systems. We analyze and compare these protocols and discuss them for other applications.

I INTRODUCTION

Peer-to-peer systems are systems in which nodes are equal. Each node in a peer-to-peer system can act as a server as well as a client that consumes the service. There are several peer-to-peer models in use in various applications. Some peer-to-peer systems follow models that have some centralized components. Napster [6, 11] for example maintains a centralized index of all content in the network. In Napster, file downloads are done in a peer-to-peer fashion between individual peers. Other networks, such as Freenet [7, 8] and Gnutella [9], follow a pure peer-to-peer model. These systems have no centralized components. Some peer-to-peer systems use a hierarchical model called super-peer model. Kazaa [10] is an example of a super-peer network. A study of super-peer networks and guidelines for building them is available in [12].
All the above systems are of ad-hoc nature. Their overlay network topology is arbitrary. Freelib differs from those systems in the fact that nodes are organized into communities according to mutual access between nodes. This significantly enhances system performance as reported in [2]. Discovery in a system like Freelib is essential. It refers to the ability to find information about other nodes in the network such as their IP address and the port number they are using. This information is typically needed to establish connections to these nodes.

The rest of this paper is organized as follows: In Section 2, we present a brief overview of Freelib. In section 3, we discuss node identity in Freelib. Section 4 presents various discovery protocols. In Section 5, we compare the complexity of discovery protocols. We give our conclusions in Section 6.

II OVERVIEW OF FREELIB

The Freelib architecture along with its performance evaluation has been reported earlier [1, 2]. However, for the reader’s convenience, we give a brief summary of the architecture in this section. The Freelib architecture consists of two overlay networks, the Symphony network and the Access network. The Symphony network is a small world network that is built based on the Symphony protocol [3]. Small world networks have the desirable criterion that the network diameter (the maximum number of hops between any two nodes) is small compared to the size of the network [4, 5]. For Symphony the network diameter is \((\log_2 n) / K\), where \(n\) is the number of nodes (the network size) and \(K\) is the number of long contacts per node which is a parameter of the system. The purpose of the Symphony network is to provide a way for new nodes to search and discover their communities. A community is a set of nodes that share the same access patterns to nodes. These nodes are linked through friend links that form them into communities. New nodes use the Symphony layer to submit and forward search queries. When the Freelib client at a node detects that it has enough friend links, it switches to use the Access network for submitting and forwarding all search queries. The Access network is built such that nodes that share common interest evolve into virtual communities. Freelib utilizes adaptive and dynamic techniques for evolving these user communities. Each client monitors its user accesses, identifies peers that are of interest to its local user, and connects itself to few of those peers. To identify peers whose interest is similar to a user, the client maintains an access log and periodically performs a ranking process that uses the most recent accesses. The outcome of the ranking process is an ordered list of peers. The client tries to establish friend links to few peers from the ranked list. Every time the ranking process is performed, the friend links are updated to reflect the latest ranked list. The use of the most recent accesses in the
ranking process ensures that the user current interest is always reflected in the results of the ranking. The ranking measures and the details of the ranking process are explained in full in [1, 2]. The Freelib network architecture is shown in Fig. 1 below.

![Freelib Network Architecture](image_url)

Fig. 1. Freelib Network Architecture.

Like other peer-to-peer systems, nodes in Freelib are autonomous. Users have the freedom to join and leave the network at any time and as often as they wish. Therefore, we need to provide a mechanism for discovery of peers. Node discovery is needed when a node needs to locate peers that have changed positions in the networks. An example scenario is when a node joins back after leaving the network. The information this node has about other peers might have already been outdated. During the time the node is disconnected, nodes might leave the network and join back using different IP addresses and/or port numbers. In addition, nodes get a new ring location on the virtual ring of our support network every time they join. This emphasizes the need for a discovery protocol that enables nodes to rediscover their access contacts and friends when they rejoin. The one question that a discovery protocol answers is: given the unique identifier (nodes get assigned a unique identifier when they join the network for the first time) of a node, what is its current information (e.g., IP, port, ring location). Hence, in our context, ‘discovery’ refers to the finding of a node’s current information given its unique identifier (UUID).
III NODE IDENTITY

We need a system-wide unique node identification mechanism. Node identifiers are needed to support the discovery protocols as well as other parts of the Freelib protocols such as peer ranking[1, 2]. The mechanism for generating node identities must be a distributed one as we do not want to introduce centralized components into our model. Nodes should not change their identifiers; rather, identifiers must be persistent. This is critical for the accuracy of the ranking calculation and for the discovery protocol. This persistence requirement disqualifies session identifiers such as IP and port pairs and ring locations. These are not persistent; rather, they might change every time the user joins the network. To fulfill these requirements, we implemented version 4 of the Universally Unique Identifier (UUID) internet draft [16]. UUID is also documented as part of ISO standard for Remote Procedure calls [13]. It is documented more recently in an International Telecommunication Union standard [14] and IETF published an equivalent RFC [15]. UUIDs are 16 bytes (128 bits) identifiers. The canonical representation of UUIDs is 32 hexadecimal digits separated by hyphens as shown in the XML element:

```xml
<uuid>8b27b39a-a907-4103-bcef-b3e375bc355d</uuid>
```

IV DISCOVERY PROTOCOLS

In a system with no centralized components, discovery is not simple; rather, it is a complicated task. The simplest approach to discovery is to flood the access network with the discovery request. This is, however, bandwidth inefficient. We introduce two new discovery protocols that are much more efficient in terms of bandwidth usage. These are the DHT discovery and the Link discovery methods. The following subsections cover a comparison of discovery protocols. We start by discussing the Flooding method and then we follow it by presenting our new discovery methods. Finally, we compare the bandwidth usage for these discovery protocols.

A. Discovery by Flooding

This approach uses flooding. Discovery requests are forwarded by every node to all its contacts and friends up to certain TTL. Although, this approach is straightforward to implement, it is very bandwidth inefficient as number of messages grows exponentially with increasing TTL. If we have TTL of $h$ hops and $c$ contacts per node on average, the number of messages per discovery request would be $\Omega(c^h)$. For example, for TTL = 7 and 10 contacts per node, the number of messages per discovery request is more than 10 millions. Although that many messages are used, the
request might not reach every node in the network because of the TTL. This could cause the discovery protocol to fail to locate information about existing nodes.

B. DHT Discovery

In the DHT approach, we build and maintain a distributed hash table (DHT) for storing discovery information of Freelib nodes. According to our definition of discovery, we want to enable nodes in our network to discover current information about other nodes including their IP addresses, port numbers, and ring locations. This information represents the values to be stored in our discovery hash table. We use the unique identifiers (UUIDs) of the nodes as keys. Hence, an entry in our discovery hash table corresponds to one node and maps from that node’s UUID to its current information. The basic operations supported by the hash table are \( \text{insert}(\text{UUID}, \text{nodeInfo}) \) and \( \text{retrieve}(\text{UUID}) \). The former inserts a hash table entry with the specified UUID and nodeInfo as its key and value respectively in the distributed hash table. The latter retrieves the nodeInfo record associated with the specified UUID or NULL if no such key can be found in the distributed hash table. In order to perform these two operations efficiently, we need to be able to efficiently route the corresponding messages to the node responsible for storing the corresponding DHT entry. We use Symphony [3] for this purpose. The Symphony protocol, which we use in building the support network, routes messages efficiently to nodes occupying or close to specific ring locations. If we can compute the ring location of the node responsible for storing a discovery entry, then we can use the Symphony protocol to route the discovery message efficiently to that node. Our computation of the ring location that corresponds to a certain UUID can be performed by any node and, hence, no centralized repository is needed. Our way to achieve this is by equipping Freelib nodes with a universal one-way hash function \( h \) under which every UUID is mapped to one and only one ring location \( \text{loc} = h (\text{UUID}) \). Under this mapping, the ring location that corresponds to a certain UUID always remains the same and does not change over time. At any moment, the node that manages ring location \( \text{loc} \) on the support network is responsible for maintaining the corresponding hash table entry \( (\text{UUID}, \text{NodeInfo}) \). We remind the reader that the node that manages a ring location is the closest node, on the support network virtual ring, to that location going anticlockwise. Every discovery request, whether insertion or retrieval, involving a UUID is routed using Symphony to the node at support network ring location \( \text{loc} = h (\text{UUID}) \). Table 1 shows six nodes constituting an example Symphony ring. It shows for each node its ring locations, UUID, \( h(\text{UUID}) \), and the node responsible for storing the discovery entry. For
example $h(\text{UUID1}) = 0.82$, which makes node N5 responsible for storing the discovery entry for N1. N5 is the closest node (anticlockwise) to location 0.82.

### TABLE I

**Nodes in a Symphony Ring and the Corresponding Nodes Responsible for Storing Their Discovery Entries**

<table>
<thead>
<tr>
<th>Node</th>
<th>Ring Location</th>
<th>UUID</th>
<th>h( UUID )</th>
<th>Node storing discovery entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₁</td>
<td>0.120</td>
<td>UUID₁</td>
<td>0.82</td>
<td>N₅</td>
</tr>
<tr>
<td>N₂</td>
<td>0.251</td>
<td>UUID₂</td>
<td>0.452</td>
<td>N₃</td>
</tr>
<tr>
<td>N₃</td>
<td>0.372</td>
<td>UUID₃</td>
<td>0.001</td>
<td>N₀</td>
</tr>
<tr>
<td>N₄</td>
<td>0.714</td>
<td>UUID₄</td>
<td>0.741</td>
<td>N₄</td>
</tr>
<tr>
<td>N₅</td>
<td>0.800</td>
<td>UUID₅</td>
<td>0.130</td>
<td>N₁</td>
</tr>
<tr>
<td>N₆</td>
<td>0.928</td>
<td>UUID₆</td>
<td>0.401</td>
<td>N₅</td>
</tr>
</tbody>
</table>

Listing 1 and 2 outline the algorithms for sending out the requests for hash table entry insertion and retrieval respectively. These algorithms use Symphony routing protocol from [3] to route the discovery insertion and retrieval messages (the call to `SendSymphonyMessage` in Listing 1 and 2). When a node receives a request for inserting a discovery hash table entry, it inserts it into a local hash table. And when a node receives a request to retrieve a discovery hash table entry, it returns the node information if the entry exists in its local hash table or `Unknown-Node` if the entry does not exist. The absence of a discovery entry could happen because of one or more of the following reasons. First, the node may have left the network and deleted its discovery entry upon leaving. Second, the entry might have been lost due to the failure of the node storing it. And third, the UUID in the discovery request might be invalid (invalid in this context means that UUID has never been used by any node). We shall introduce solutions to handle node failure (the second case above) through replication in the following section. For nodes that already left the network and for discovery requests that involve invalid UUID, it is sufficient to return `Unknown-Node` response.

Listing 3 and 4 outline the processing of the insertion and retrieval requests respectively by the receiving nodes. These algorithms are straightforward. One critical feature, however, is the consistency in generating the ring locations that correspond to node UUIDs. All nodes must use the same exact hash function whenever the ring location that corresponds to a certain UUID needs to be generated. And therefore, the ring location that corresponds to certain UUID stays the same regardless of the node that is generating it. Consequently, all discovery requests that involve a certain UUID are routed to the same ring location.
Using this method, a discovery request is routed to its destination using $\Omega\left(\log^2 \frac{n}{k}\right)$ messages, where $n$ is the number of nodes and $k$ is the number of long contacts per node. This is an order of magnitude less than the number of messages used in the flooding method. However, there is a small overhead at the time of joining and leaving the network. When a node joins the Freelib network, it needs to: 1) initiate the insertion protocol for inserting an entry for itself using its own UUID as the key, which costs at most $\log^2 \frac{n}{k}$ messages; and 2) claim its share of the distributed hash table from the neighboring node on the support network, which costs 2 messages (a request and its response). When a node leaves, it needs to: 1) delete its entry from the distributed hash table, which costs $\log^2 \frac{n}{k}$ messages at most; and 2) transfer its portion of the distributed hash table to its neighboring node on the
support network, which costs 1 or 2 messages depending on whether an acknowledgement is sent back. The total overhead per node is \(2 \times \log_2 \frac{n}{k} + 4\) messages, which is still \(\Omega(\log \frac{n}{k})\).

Fault-tolerance of DHT discovery method can be enhanced in many ways. One way to enhance it is to have every node periodically sends its discovery entry for insertion. This recovers the discovery entry if it has been lost due to failure of the node storing it. We call this DHT with Repetition. A second approach to enhancing fault-tolerance of the DHT Discovery protocol is introducing replication. This is the subject of the next subsection.

C. Replication of Discovery Information

Failure of Freelib nodes can adversely affect our DHT discovery protocol. If a node fails without having the chance to transfer the discovery entries it maintains to its neighboring node on the ring, these discovery entries are lost from the discovery distributed hash table. Consequently, any discovery requests asking for these entries will not be fulfilled. This can cause inconvenience and disruption of the discovery service to nodes especially if the rate of failure is high. We use replication to alleviate this issue and enhance fault-tolerance of the DHT discovery protocol.

Replication is a popular technique to significantly enhance availability. If the probability of a node to fail is \(P\), and nodes fail independent from each other, then the probability of \(r\) nodes to fail is \(P^r\). For example, if \(P = 0.1\) and \(r = 3\) replicas, the chance of all three replicas to fail is 0.001, which is order of magnitudes smaller.

We introduce a new replication scheme that works at the granularity of individual discovery entries, instead of replicating at the level of nodes. Instead of having pairs of nodes whose discovery information is exact replica of each others, our replication scheme selects \(r\) nodes for storing each discovery entry, where \(r\) is the number of replicas. In order for all nodes to be able to locate the replicas for a certain UUID, we use a universal hash function \(h\) to determine the ring locations of the \(r\) replicas as follows. The first ring location is calculated based on the UUID of the discovery entry using \(h\) as before, \(loc = h(\text{UUID})\). The remaining \(r-1\) ring locations are chosen such that the whole set of \(r\) replication ring locations are equidistant on the ring. For example, for \(r = 2\), each discovery entry \((\text{UUID}, \text{NodeInfo})\) is stored at two ring locations that are half way across the ring from each other. These ring locations are \(loc_1 = h(\text{UUID})\) and \(loc_2 = (loc_1 + 0.5) \mod 1.0\). Table 2 shows the same nodes in our previous example, from Table 1, with two replicas storing the discovery entry for each node.

To further illustrate what we mean by granularity level of the individual discovery entry, which we mentioned earlier; let’s consider nodes \(N_2\) and \(N_6\) in Table 2. Each of these nodes has \(N_3\) as a first replica. However, the second
replica for each of them is different. For node N2, the second replica is N6 whereas for node N6, the second replica is N5. When two nodes are selected as two replicas for a certain UUID, it does not mean that they are exact replica of each other; rather, it means that they are replicas for storing this specific discovery entry.

Our discovery replicas are all the same level and there is no notion of primary or secondary replicas. In addition, there is no internal synchronization between replicas. Rather, discovery insertion and deletion requests are sent to all the replicas. And when a node leaves, it requests deletion of its discovery entry from all its replicas. Discovery retrieval requests can be handled in several different ways, however. One option is to send discovery retrieval requests to all replicas simultaneously. Another option is to try one replica at a time. In the later case, randomization could be utilized to achieve load balancing. Alternatively, requests could be sent to the closest replica on the support ring as this reduces the number of forwarding steps required to reach the destination as discussed in [3]. In this case, if a discovery fails, the next closest replica on the discovery ring is tried. Listing 5 outlines the procedure for sending out discovery insertion requests to all of three replicas. Similarly, Listing 6 shows the procedure for sending out discovery retrieval requests using all three replicas.

D. Link Discovery

Link discovery is an alternate, new approach to node discovery in Freelib. In this approach, each joining node establishes and maintains a discovery link to a target node whose ring location is specified by hashing the joining node unique identifier. In other words, instead of building a discovery DHT, every node establishes one discovery link to the manager of the ring location \( \text{loc} = h (\text{UUID}) \), where \( h \) is the universal hash function, \( \text{UUID} \) is the unique identifier of the node. We refer to the node that establishes the discovery link as the owner of the discovery link and to the target node of the discovery link of as its discovery contact. The discovery contact of a node can be located anywhere on the support ring. This depends on the ring location generated by the hash function \( h \) and the density of nodes on the ring. When a node joins, it establishes a link to its discovery contact. Symphony routing is used when a request to establish a discovery link is sent. Whenever a node is leaving the network, it notifies its discovery contact and disconnects its discovery link. The discovery link is a live link that requires keep-alive mechanism such as periodical pings. These pings are light-weight messages that are sent directly to the destination. No peer-to-peer routing is involved at all when ping messages are sent. In order to discover information about a node \( N_i \), a discovery request is routed using Symphony to \( N_i \)'s discovery contact, which costs \( \Omega \left( \frac{\log^2 n}{k} \right) \) messages.
Maintenance of the discovery link is needed in two cases: 1) when the discovery contact fails; and 2) when a new node joins using a ring location that is between the location of the discovery contact and the actual ring location associated with the discovery link (which is \( \text{loc} = h(\text{UUID}) \), where \( \text{UUID} \) is the unique identifier of the owner of the discovery link). In the first case, the predecessor of the discovery contact becomes the new discovery contact for the owner of the discovery link. In the second case, the newly joining node becomes the new discovery contact. In both cases, failure of discovery contact and new node joining between discovery contact and the actual ring location associated with the discovery link, the discovery link needs to be reestablished, which costs \( \Omega\left(\frac{\log n}{k}\right) \) messages.

### Table II

**Nodes in a Symphony Ring and the Corresponding Nodes Responsible for Storing Their Discovery Entries, Each Entry is Stored at Two Replicas.**

<table>
<thead>
<tr>
<th>Node</th>
<th>Ring Location</th>
<th>UUID</th>
<th>( h(\text{UUID}) )</th>
<th>Node storing discovery entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_1 )</td>
<td>0.120</td>
<td>( \text{UUID}_1 )</td>
<td>0.82</td>
<td>( N_5, N_2 )</td>
</tr>
<tr>
<td>( N_2 )</td>
<td>0.251</td>
<td>( \text{UUID}_2 )</td>
<td>0.452</td>
<td>( N_3, N_6 )</td>
</tr>
<tr>
<td>( N_3 )</td>
<td>0.372</td>
<td>( \text{UUID}_3 )</td>
<td>0.001</td>
<td>( N_6, N_3 )</td>
</tr>
<tr>
<td>( N_4 )</td>
<td>0.714</td>
<td>( \text{UUID}_4 )</td>
<td>0.741</td>
<td>( N_4, N_1 )</td>
</tr>
<tr>
<td>( N_5 )</td>
<td>0.800</td>
<td>( \text{UUID}_5 )</td>
<td>0.130</td>
<td>( N_1, N_4 )</td>
</tr>
<tr>
<td>( N_6 )</td>
<td>0.928</td>
<td>( \text{UUID}_6 )</td>
<td>0.401</td>
<td>( N_3, N_5 )</td>
</tr>
</tbody>
</table>

This cost can be significantly reduced. Adding little additional information to the ping messages can help in the case of failure of the discovery contact. The discovery contact needs to send information on its ring predecessor in the ping messages. When a node detects failure of its discovery contact, it knows the predecessor and it immediately establishes discovery link to it. This effectively reduces the cost to \( \Omega(1) \). In the case of a new node joining and becoming the new discovery contact (like node \( Y \) in step 4 of the example scenario above) the original discovery contact just notifies the discovery link owner, which establishes a new discovery link to the joining node. This also reduces the cost in this case to \( \Omega(1) \). A separate issue is the failure of the owner of a discovery link. In this case, the discovery contact detects failure of the discovery link owner and releases its resources accordingly. No maintenance is required in this case.
Listing 5: Insertion of discovery information at all three replicas

```java
insertDiscoveryInfo (String uuid_str, String nodeInfo) {
    double loc1 = h (uuid_str), loc2 = (loc1 + 0.33) mod 1,
    loc3 = (loc1 + 0.67) mod 1;
    SendSymphonyMessage(loc1, new
        DiscoveryInsertMessage(uuid_str, nodeInfo));
    SendSymphonyMessage(loc2, new
        DiscoveryInsertMessage(uuid_str, nodeInfo));
    SendSymphonyMessage(loc3, new
        DiscoveryInsertMessage(uuid_str, nodeInfo));
}
```

Listing 6: Retrieval of discovery information from all of three replicas simultaneously

```java
retrieveDiscoveryInfo (String uuid_str) {
    double loc1 = h (uuid_str), loc2 = (loc1 + 0.33) mod 1,
    loc3 = (loc1 + 0.67) mod 1;
    SendSymphonyMessage(loc1, new
        DiscoveryRetrievalMessage(uuid_str));
    SendSymphonyMessage(loc2, new
        DiscoveryRetrievalMessage(uuid_str));
    SendSymphonyMessage(loc3, new
        DiscoveryRetrievalMessage(uuid_str));
}
```

V COMPARISON OF DISCOVERY PROTOCOLS

Table 3 shows a comparison the cost of the various discovery protocols associated with events such as joins, leaves, node failure, and sending out discovery requests. All the costs are per node cost. The repeated insertion and pings are performed periodically at a certain rate per unit time. Other costs are incurred when certain events, such as node failures and sending out discovery requests, happen. The table shows the costs of these operations for the Flooding, Plain DHT, and DHT with Repetitions, Replicated DHT, Replicated DHT with Repetitions, and Link Discovery algorithms. As mentioned earlier, the with-repetitions versions refer to variants of the discovery protocols in which the discovery insertion requests are sent periodically to avoid losing the discovery information when nodes fail.

From the time complexity shown Table 3, we can see that Flooding has a high cost associated with discovery requests. The number of messages grows exponentially with $h$ (the max TTL). Plain DHT discovery and Replicated DHT significantly reduce the cost of discovery but suffer loss of discovery information as nodes fail. The versions with repetition enhance this by making that loss temporary as lost discovery entries are recovered at the time of the next insertion requests by the nodes whose discovery entries were lost. In the last row in Table 3, we show the effect of node failure. The reader should not that for replicated variants, losing a discovery entry happens when all the
replicas storing that entry fail. The chance for this to happen gets significantly smaller as the number of replicas increases. Link discovery beats the other DHT discovery protocols in two cases, which are leave and periodical pings. When the discovery contact of a node fails, the discovery link needs to be reestablished. This is equivalent to re-insertion of the discovery entry in DHT with repetition. Based on this cost comparison, we chose to implement and use the Link discovery protocol in our implementation of Freelib universal client. It avoids permanent loss of discovery information. In addition, it replaces unnecessary re-insertion requests, which cost \( \Omega \left( \frac{\log^2 n}{k} \right) \) each with simple ping messages.

### TABLE III: COMPARISON OF THE VARIOUS DISCOVERY ALGORITHMS

<table>
<thead>
<tr>
<th></th>
<th>Flooding</th>
<th>DHT Discover</th>
<th>DHT with repetitions</th>
<th>Replicated DHT</th>
<th>Replicated DHT with repetitions</th>
<th>Link Discovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join / Establish</td>
<td>0</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
</tr>
<tr>
<td>Discovery</td>
<td>( \Omega (c^k) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
</tr>
<tr>
<td>Leave</td>
<td>0</td>
<td>( \Omega (1) )</td>
<td>( \Omega (1) )</td>
<td>( \Omega (1) )</td>
<td>( \Omega (1) )</td>
<td>( \Omega (1) )</td>
</tr>
<tr>
<td>Periodical Insertion / Pings</td>
<td>N/A</td>
<td>N/A</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>N/A</td>
<td>( \Omega \left( \frac{\log^2 n}{k} \right) )</td>
<td>( \Omega (1) )</td>
</tr>
<tr>
<td>Failure of ( r ) Node (for non-replicated protocols, ( r = 1 ))</td>
<td>0</td>
<td>1 discovery entry lost temporarily until next insert</td>
<td>1 discovery entry lost temporarily until next insert</td>
<td>1 discovery entry lost temporarily until next insert</td>
<td>1 discovery entry lost temporarily until reestablish</td>
<td>( \Omega (1) )</td>
</tr>
</tbody>
</table>

**VI Conclusions**

We introduced discovery protocols that enable nodes in Freelib to find their previously discovered friends and peers. The straight-forward flooding technique is very inefficient. It consumes considerable network bandwidth. We then introduced two new discovery protocols that are built on top of the Freelib support overlay network. The first is DHT discovery, which builds a distributed hash table that maps node identities to node information. We discussed variants of the DHT discovery protocol that utilize repetition of the insertion operation and other variants that use replication of the discovery information. The second major discovery protocol we presented is Link discovery in which each joining node establishes and maintains a discovery link to a target node whose ring location is specified...
by hashing the joining node unique identifier. In this protocol, all nodes use the same hash function, which guarantee that all discovery requests for a certain node identifier are routed to the same ring location and reach the correct target node. We compared the complexity associated with these discovery protocols. Our analysis shows that Link discovery outperforms the other discovery protocols. We implemented Link discovery in our Freelib client.

REFERENCES


