

Peer-to-Peer Document Sharing using the Ant Paradigm

Hein Meling[‡] Alberto Montresor^{*} Özalp Babaoğlu^{*}

Abstract

The peer-to-peer (P2P) paradigm for building distributed applications has recently gained renewed attention, partly due to the enormous success of systems like Napster and Gnutella. Subsequently, a multitude of projects focusing on anonymity, security, routing and reliability aspects of P2P have been initiated. A framework that supports the design, evaluation and implementation phases of P2P application development is currently missing. The Anthill project is an attempt to fill this void.

In this paper we give a brief overview of Anthill and describe Gnutant, a document sharing application that combine the scalability property of Freenet with the free search capabilities of Gnutella. In addition, we present preliminary simulation results obtained by running the Gnutant application in the Anthill simulation environment. The simulation results indicate that after an initialization period, Gnutant is effective in finding documents.

1 Introduction

Several recent distributed applications based on the *peer-to-peer* (P2P) paradigm have drawn media headlines and industry attention. Informally, P2P applications are composed of a collection of *peer* nodes that cooperate in order to perform some task and share their resources by direct exchanges [10]. In the P2P model, each node may be both a provider and consumer of services (i.e., a *peer*), which differs from the client-server model where only a relatively small number of server nodes provide services to a potentially large number of client nodes.

The enormous interest around P2P is motivated by the success of Napster [11] and Gnutella [9], two file-sharing applications that have been able to attract millions of users. In addition to its popular appeal, P2P also offers interesting technical properties. In particular, the completely decentralized nature of the model enables development of applications with reliability and availability characteristics previously unimaginable over the Internet. Other application domains where P2P technology has been successfully applied include scientific computing [1] and messaging and collaborative tools [7]. And these are just a few examples of a much broader class of potential applications suitable for the P2P model.

Since P2P computing involves distribution of resources and services across a network of independent systems, new issues related to reliability, availability and security arise.

[‡]Department of Telematics, Norwegian University of Science and Technology, O.S. Bragstadspllass 2A, N-7491 Trondheim (Norway), Email: meling@item.ntnu.no

^{*}Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy), Email: {babaoglu,montresor}@CS.UniBO.IT

However, traditional techniques for dealing with them are not directly applicable to P2P systems due to scalability problems and the extremely dynamic nature of the operating environment where nodes may be added and removed in rapid succession. Recent industry-lead initiatives such as JXTA and the Peer-to-Peer Working Group are attempts to address these problems, focusing mainly on standardizing the communication infrastructure on which peers operate [10, 8].

Anthill [2] is yet another framework for P2P application development, deployment and testing. The goals of Anthill are to: (i) provide an environment for simplifying the design and deployment of new P2P systems, and (ii) provide a “testbed” for studying and experimenting with P2P systems in order to understand their properties and evaluate their performance. Anthill is based on the *multi-agent systems* (MAS) paradigm [6], in which a collection of *autonomous agents* can move around, observe and modify their environment, performing predefined tasks. MAS often exhibit a property called *swarm intelligence*, in that a collection of simple agents with limited individual capabilities achieves “intelligent” collective behavior [12], enabling them to solve problems that are beyond the capabilities or knowledge of individual agents. MAS are particularly suitable for solving problems in highly dynamic environments [5], subject to incomplete and imprecise information [13] and without centralized control. Thus, we believe that the MAS paradigm is an appropriate basis for modeling and building P2P applications, which exhibit exactly these properties. A particular type of MAS are based on the ant colony metaphor, in which *ants* take the role of agents. Throughout the paper we adopt terminology derived from the ant paradigm.

Anthill simplifies P2P application development and deployment by freeing the programmer of all low-level details including communication, security and ant scheduling. Thus, developers can focus on writing appropriate ant algorithms for solving a particular problem, using the Anthill API and defining the structure of the P2P system. Anthill also includes a simulation environment to help analyze and evaluate the behavior of P2P systems. A particular advantage provided with Anthill is that the same ant algorithm implementations can be used without modification in both the simulation and real network environments, thus avoiding the cost of maintaining two different implementations. This important feature allows developers to tune their ant algorithms in the simulation environment, and then deploy them directly in a real network environment.

In this paper we present *Gnutant*, a novel document sharing application that provides the user with free search capabilities without imposing strict limits on its scalability. This is achieved by implementing a set of ant algorithms and related data structures using the Anthill framework, that facilitates routing based on the query string entered by the user. The routing tables in the network are continuously evolving to improve the search performance. The collective behavior of individual ants enable Gnutant to evolve a replicated and distributed document index with high availability and strong fault tolerance characteristics. In addition to its novelty, Gnutant also illustrates the simplicity of developing P2P applications using Anthill.

The rest of this paper is structured as follows. Section 2 introduces the main components of the Anthill framework and their interfaces. Section 3 presents the details of the Gnutant document sharing application and its implementation in the Anthill simulation environment. Section 3 also includes some preliminary simulation results of the Gnutant application. Section 4 discusses other work that relates to the Gnutant application. Finally, Section 5 concludes the paper and indicates directions for future work.

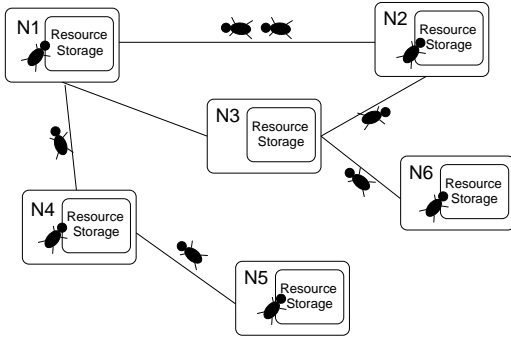


Figure 1: Overview of a nest network.

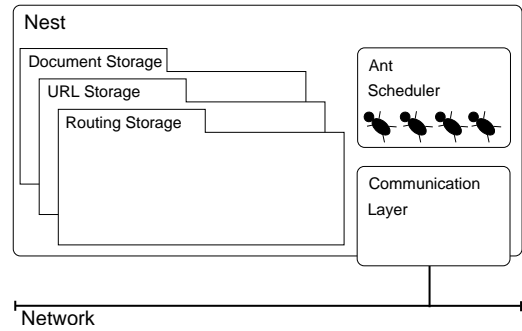


Figure 2: The architecture of a nest.

2 Anthill Overview

Anthill uses terminology derived from the ant colony metaphor. A P2P system based on Anthill is composed of a network of interconnected *nests*. Each nest is a peer entity capable of performing computations and hosting resources. Nests handle user requests by generating one or more *ants* – autonomous agents that travel across the nest network trying to satisfy the request. Ants communicate indirectly with each other by modifying their environment (i.e., modifying information stored in nests.)

In this section, we introduce the *Anthill model* and describe its core entities. We then briefly discuss the characteristics of the *runtime* and the *simulation* environments included in Anthill. These two distinct implementations of the model are used for real network deployment and for evaluation purposes, respectively.

2.1 The Anthill Model

The Anthill model consists of two entities: *ants* and *nests*. A *nest network* is the infrastructure on which P2P applications run and corresponds to the “can communicate directly” relation between nests (see Figure 1.) It is an instance of an *ad hoc* network in that there is no fixed structure and its topology is highly dynamic as nests come and go and discover each other on top a communication substrate (e.g., TCP/IP). Each nest is capable of performing computations and storing information on behalf of either its owner or visiting ants. Ants are autonomous agents that move across the nest network, interacting with nests that they visit in order to accomplish their task. The actual services provided by a P2P application are implemented through *ant algorithms*. See Section 3.1 for two example algorithms. In the following, the term *ant species* is used to refer to a set of ant algorithms that work collectively to solve a problem (i.e., provide a particular service.)

2.1.1 The Nest

Figure 2 illustrates the architecture of a nest that is composed of three logical modules: resource storage, ant scheduler and communication layer. Resource storage modules are specialized depending on the type of information they contain and each nest includes those that it needs (the nest in Figure 2 has three such modules). Each storage module type is associated a set of policies for managing the available (inherently limited) memory or disk space. For example, a *least-recently-used* (LRU) policy may be used to discard documents when space is needed for new documents. The *ant scheduler* module multiplexes

```

public interface Nest {
    void request(Request request,
        ReplyListener listener);
    void addAntSpecies(AntFactory factory);
    void addNeighbor(NestId nid);
    void removeNeighbor(NestId nid);
    NestId[] getNeighbors();
}

```

Figure 3: The **Nest** interface.

```

public interface AntView {
    boolean move(NestId nid);
    Storage getStorage(String name);
    void addNeighbor(NestId nid);
    NestId[] getNeighbors();
    void reply(ReqId rid, Reply reply);
}

```

Figure 4: The **AntView** interface.

the nest computation resource among visiting ants. It is also responsible for enforcing nest security by limiting the actions and resources available to foreign ants. Finally, the *communication layer* is responsible for the movement of ants between nests and for nest network topology management by monitoring reachability of known remote nests.

Each nest has a unique identifier. An ant must know the identifier of a remote nest in order to be able to move to it. Ants find out about remote nests by interrogating the local nest so that they may explore new regions of the nest network. Each nest maintains a set of *neighbors* as the remote nests that it knows about. As noted above, the collection of neighbor sets defines the nest network which may be highly dynamic: a nest may find out about a new neighbor either through the user or a visiting ant, and it may forget about a known nest if the communication layer considers it unreachable.

2.1.2 Requests and Replies

Users interact with nests by performing requests and listening for replies. For example, in a music-sharing network, a request would be a query for the songs of a particular artist, and the reply would contain a set of URLs to songs by the given artist. In a P2P lookup service associating name-value pairs (such as Chord [4]), a request would be a lookup for a particular name, and the reply would contain the value associated with that name.

Anthill does not impose any particular format on requests and replies. The interpretation and satisfaction of requests are delegated to ants. Furthermore, Anthill does not specify which services a nest should provide. When a nest receives a request from its user, it selects an ant species that is appropriate for the request from a set of ant species known to it. This set is dynamic, as new ant species may be installed by the user, or ants belonging to new species may arrive from remote nests.

Interface `Nest` shown in Figure 3 contains the methods that may be invoked by the P2P application to interact with a nest. The main method of this interface is `request()`, which is used to perform new requests and to register a listener for replies. Furthermore, the interface also provides methods for nest administration, such as addition and removal of neighbors and registration of new services (*ant species*.)

2.1.3 Ants

Ants are generated by nests in response to user requests; each ant tries to satisfy the request for which it has been generated. An ant will move from nest to nest until it fulfills its task, after which it “may” return back to the originating nest. Ants that cannot satisfy their task within a *time-to-live* (TTL) parameter are terminated. When moving, the ant carries its state, which may contain the request, results or other ant specific data. Note that the ant algorithm itself need not be transmitted if the receiving nest already knows it.

The behavior of an ant is determined by its current state and its algorithm, which may be non-deterministic. For example, an ant may probabilistically decide not to follow

what is believed to be the best route for accomplishing a task, and choose to explore alternative regions of the network. Ants must implement an Ant interface (not shown), which basically consists of a run() method. The run() method describes the ant algorithm and is executed at each nest visited during the ant's trip.

2.1.4 AntView – Limiting the Actions of an Ant

When a nest invokes the run() method of an ant, an object implementing the AntView interface (Figure 4) is passed to the method. This interface contains those actions that the ant is allowed to perform when executing within the nest. These include:

- move to another nest specified by its identifier;
- obtain access to local data storages;
- inform the nest of the existence of additional neighbors by adding their identifiers;
- obtain the neighbors known to the local nest;
- notify the nest about a request for which the ant has produced a reply.

2.2 The Runtime Environment

The runtime environment is a nest implementation for deploying ants in a real distributed environment. It manages low-level functions such as communication, security, resource management and ant scheduling. It implements the Nest and AntView interfaces, enabling the execution of ant-based distributed algorithms that perform various services.

2.3 The Simulation Environment

To evaluate new ant algorithms, Anthill includes a simulation environment through which the behavior of a particular ant implementation may be simulated and assessed. The nest implementation included in the simulation environment is simpler than the one included in the runtime environment since remote communication can be achieved through local interactions. Nevertheless, it is important to note that ant algorithms are totally independent from the nest implementation and continue to work in both environments without any changes.

Each simulation study, called a *scenario*, is specified using XML by defining the nest network and a request generator. The nest network is specified through the total number of nests and the number of neighbors that each nest has. In the current implementation, the nest network is generated prior to the simulation and assumed to remain static throughout the simulation. The set of neighbors for each nest are generated randomly over the set of all nests. We plan to allow dynamic nest networks by specifying the total number of nests and the neighbor degree as *probability distributions* rather than static values. The actual sequence of requests that are to be satisfied are generated on-the-fly during the simulation through the request generator. Simulating different P2P applications requires developing appropriate ant algorithms and a corresponding request generator characterizing user interactions with it.

The simulation proceeds by executing the sequence of generated requests on the nest network and by monitoring performance parameters such as the number of request initiated, satisfied, ant moves performed, network generated traffic, etc. Using the simulation environment, programmers can evaluate several different scenarios and obtain average figures for the collected statistics.

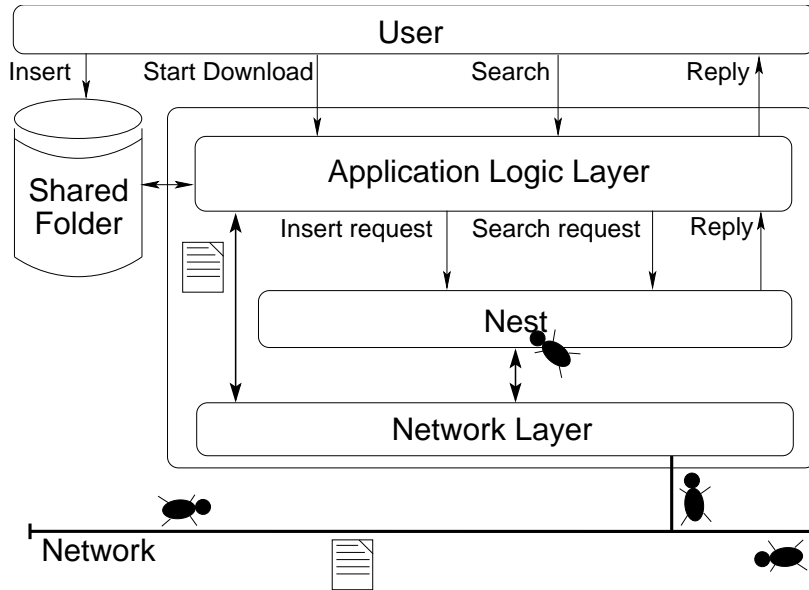


Figure 5: Gnutant Application Overview.

3 The Gnutant Document Sharing Application

In this section, we present our preliminary experience in using Anthill to build a document sharing application called *Gnutant*. In order to facilitate document searches, Gnutant builds a replicated document index in which document URLs are stored. The index is distributed across a network of nests. Different types of ants are used to perform searches and insertions in the distributed index, as described below.

Figure 5 gives an overview of Gnutant. The user inserts documents for sharing by simply copying them to a local folder. The user may also issue search queries and listen for replies, or may select documents for downloading from remote sites. The *application logic layer* detects new documents placed in the shared folder and receives search queries from the user. When a new document is detected in the shared folder, an *insert request* is issued to the local nest in order to advertise the presence of the document to other nests in the network. A search query is simply passed on to the nest as a *search request*. Upon receiving a request, the nest will use *ant factories* to generate appropriate ants to handle the request.

Each nest is configured as shown in Figure 2. It includes a *document storage* for managing documents in the shared folder; a *URL storage* containing URLs to documents; and a *routing storage* that ants may access or modify in order to make routing decisions or improve the routing of future ants, respectively. Each URL storage in the network constitutes a portion of the distributed document index.

In Gnutant, each document is associated some *meta-data* comprised of a set of textual *keywords* and a unique *document identifier*. The keywords associated with the document are used by Gnutant for routing and to organize the distributed index, and may be provided by the user who inserted the document, or obtained automatically from the filename. The document identifier is composed of the file size and a digest computed over the document itself, and enable comparison of documents for equality. Thus, different URLs to replicas of the same document will have the same document identifier. We exploit this property in order to provide faster downloads of documents; Gnutant may attempt to download disjoint fragments of the document from multiple locations. Assuming that the replication

degree is above a certain threshold, load balancing downloads in this manner will have several positive effects:

- faster and more timely downloads;
- reduced load on peers providing document replicas;
- increased fault-tolerance and document availability.

An insertion request for a document contains the document identifier, a URL and the collection of keywords, while a search request simply specifies a collection of keywords. We say that a document “satisfies a search request” if its set of associated keywords contains *all* keywords included in the search request.

3.1 The Gnutant Ant Species

Real world ants, even though of the same species, may have different specializations, such as workers, soldiers, or queens. The Gnutant species similarly contains a set of *ant algorithms* designed to perform various tasks, such as hunting for documents or updating the distributed document index. Gnutant includes the following ant types:

- *InsertAnt*: an ant specialized in advertising the existence of documents by insertion of URLs into the distributed index. It is generated by Gnutant when there is a new document available in the shared folder, either because the user placed it there or after the document has been downloaded.
- *SearchAnt*: an ant specialized in document searches. It is generated by Gnutant in response to user queries. It exploits the information left in the routing storages by other ants, trying to determine the shortest path to documents matching the user query. Upon reaching its TTL, the ant will return to the originator nest backtracking its path. During the return trip, the ant will update both the distributed index and the routing storages to reflect its findings.
- *ReplyAnt*: an ant used to reduce the response times of searches. It is generated at each nest where a SearchAnt locates a document. The ReplyAnt returns immediately to the originator nest while the SearchAnt may continue its exploration hoping to find other documents satisfying the search request.

To advertise a new document, an insert request is sent to the local nest. In response, the nest generates an InsertAnt for each keyword associated with the inserted document. Similarly, a SearchAnt is generated for each of the keywords contained in a search request. Each of these ants carries the entire search query (list of keywords) and they attempt to satisfy the given request concurrently, exploring different regions of the nest network since they will be routed based on their associated keywords. Next we will explain how this routing mechanism works.

3.1.1 Gnutant Routing

To make routing decisions, both InsertAnt and SearchAnt make use of the specialized routing storage provided with Gnutant. It is based on the concept of *hashed keyword routing*, which is similar to the routing technique used in Freenet [3]. The routing storage associates the hash value of a keyword with a set of nests that are believed to store URLs for documents associated with the corresponding textual keyword. As previously explained, each InsertAnt/SearchAnt instance is associated with exactly one hashed keyword, and when visiting a nest, an ant may inspect the routing storage using its associated

hashed keyword. If an exact match is found in the routing storage, the ant selects a nest from the set corresponding to the matching hashed keyword; otherwise, a nest associated with the “closest” hashed keyword is selected.

The hash value of a keyword is computed using the Secure Hash Algorithm (SHA) to obtain a 160 bit value. This mapping from the textual string space to the bit string space enables us to compare hashed keywords to determine their closeness. Furthermore, hashing the keywords also helps disperse the load evenly on the routing storages due to the uniformity property of SHA. Basing routing storages on the raw textual keywords would result in highly unbalanced load since keywords tend to be highly clustered in textual string space.

The notion of closeness between hashed keywords is fundamental to Gnutant’s routing scheme. It allows nests to become biased toward a certain portion of the hashed keyword space. If a nest is listed in a routing storage under a particular hashed keyword, it will tend to receive more requests for hashed keywords similar to that hashed keyword. Moreover, nests become specialized in storing URLs of documents having similar hashed keywords, since forwarding a request will result in the nest itself gaining a URL for the requested document. This clustering property will improve the search performance over time as the routing storages evolve their knowledge, enabling ants to quickly find the relevant region in the nest network.

3.1.2 Gnutant Algorithms

The algorithm for an InsertAnt is given in Algorithm 1. The ant constructor takes three parameters: the identifier of the inserted document, the document URL, and the hashed keyword associated with this ant. InsertAnt also carry the path followed by the ant, allowing it to avoid duplicate visits to the same nest. When an InsertAnt visits a nest, it first adds the URL to the local URL storage, associating it with the provided document identifier. Next, it updates the routing storage by associating the hashed keyword with the ant’s originating nest. Finally, it obtains an ordered list of nests that are believed to contain keywords close to that associated with the ant, and moves to the first one that is reachable.

Algorithm 2 illustrates the algorithm for a SearchAnt. The ant constructor takes three parameters: a request identifier, used by the originating nest to associate replies with a previous requests; a string containing the keywords to search for; and the hashed keyword associated with this ant. In addition, SearchAnts also carry the path followed by the ant, allowing it to backtrack and to avoid duplicate visits to the same nest. When a SearchAnt arrives at a nest, it queries the local storage for documents satisfying the search query. These are added to its set of matches (*urls*) that is carried with the ant. Unless the TTL of the SearchAnt has been exhausted, it obtains an ordered list of nests that are believed to contain keywords close to that included in the ant, and it moves to the first reachable nest in the list. Once the SearchAnt has reached its TTL, it will backtrack to the originating nest. While going back, the SearchAnt will update both the distributed index and routing tables in its path, allowing other ants to improve their performance in finding similar documents.

The only task of a ReplyAnt is to return to the originating nest and deliver a reply. Given its simplicity, we omit the algorithm.

Algorithm 1 InsertAnt

```
class InsertAnt implements Ant {
  InsertAnt(DocId docid, URL url, Key keyhash) {
    this.docid = docid;
    this.url = url;
    this.keyhash = keyhash;
    path =  $\emptyset$ ;
  }
  void run(AntView view) {
    urlStore = view.getStorage(URL_STORAGE);           {Obtain reference to local storages}
    route = view.getStorage(GNUTANT);
    path.add(view.getNestId());                         {Update the path with this nest}
    urlStore.addResource(docid, keyhash, url);          {Add the url to the local URL storage}
    route.addKeyhash(keyhash, path.getFirst());        {Associate originating nest with keyword hash}
    nxtList = route.getNextNest(keyhash, path);        {Get list of possible next nests}
    do {
      moved = view.move(nxtList.get(i + +));           {Move to the next nest that is reachable}
    } while (!moved);
  }
}
```

Algorithm 2 SearchAnt

```
class SearchAnt implements Ant {
  SearchAnt(ReqId rid, String query, Key keyhash) {
    this.rid = rid;
    this.query = query;
    this.keyhash = keyhash;
    path =  $\emptyset$ ;
  }
  void run(AntView view) {
    urlStore = view.getStorage(URL_STORAGE);           {Obtain reference to local storages}
    route = view.getStorage(GNUTANT);
    if (view.getTTL() > 0) {
      path.add(view.getNestId());                       {Update the path with this nest}
      size = path.size();
      urls.add(urlStore.getResources(query));           {Get matching urls from local URL storage}
      nxtList = route.getNextNest(keyhash, path);      {Get list of possible next nests}
      nxtList.addLast(path.getFirst());                 {Move home if nxtList exhausted}
      do {
        moved = view.move(nxtList.get(i + +));         {Move to the next nest that is reachable}
      } while (!moved);
    } else {
      if (size > 0) {
        urlStore.addResources(urls);                    {Update local storage with resources found}
        route.addKeyhash(keyhash, path.getLast());     {Update routing storage}
        view.move(path.get(size - -));                  {Move backward}
      } else {
        view.result(rid, urls);                          {Deliver results}
      }
    }
  }
}
```

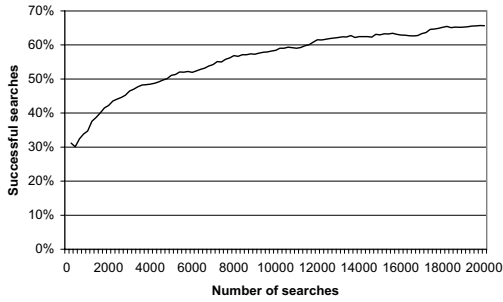


Figure 6: Search success rate.

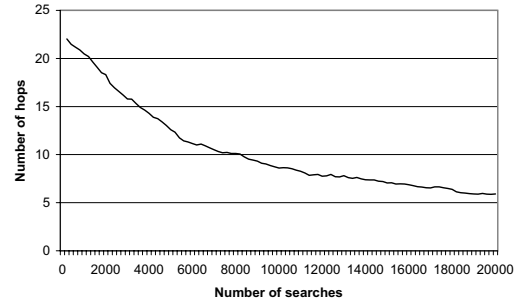


Figure 7: Number of hops until first reply.

3.2 Simulation Results

In this section we present an evaluation of preliminary results for the Gnutant application obtained using the Anthill simulation environment. In order to render our simulation more realistic, we have collected a set of 10,000 query strings by monitoring the Gnutella network over a period of approximately 30 minutes. The obtained query strings were also used as the names for 10,000 documents, all of which were inserted into the nest network *a priori* to running the simulation. Thus potentially, all of the queries could have been satisfied. Furthermore, the routing storages was initialized with randomly generated SHA keys, causing the ants to move randomly in the beginning. The simulation was run on a static 2,000-node nest network with a logically fully meshed connectivity. After the insertion phase, 20,000 search requests were issued and statistics for the behavior of the system was collected. Search requests for the simulation were generated by randomly picking queries from the set of 10,000 Gnutella query strings. The TTL parameter for the search ants was fixed at 100 hops. The simulation was repeated ten times in order to obtain average values.

The simulation results are shown in Figures 6 and 7. Figure 6 shows the success rate for search requests, while Figure 7 shows the number of hops necessary for the first reply to a search request. As expected, both figures confirm that the performance of the system improves over time, as the total number of requests performed increases and the content of the distributed index evolves. Furthermore, we can see from the figures that the system converges towards a 65% success rate for searches and approximately six hops for the average search depth.

3.3 Discussion

In the simulation we used a logically fully meshed network, however in a real network deployment, information from the local routing storage could be exploited for connection establishment. This would limit the number of connections to the size of the routing storage.

Although not shown in the algorithms in this paper, the Gnutant ant species can easily be adapted to support *reader anonymity*. This can be achieved by the requester augmenting the path with an arbitrary number of nest entries before adding its own nest identifier. This will prevent other nests from inferring the identity of the originator of an ant. Supporting *publisher anonymity* is more difficult, and does not admit use of URL storages, but instead requires copying documents along the path as in Freenet [3].

The current implementation of Gnutant does not support detection and removal of stale

URLs from the URL storage. Assuming network partitions and nest crashes are transient, the main reason for a URL to become stale is that a user cease to share a document for a “long” period of time. One approach to resolve this problem is to add the notion of a *lease* to the URLs stored in the distributed index; nests that do not crash and keep sharing their resources may periodically launch InsertAnt to refresh the leases.

4 Related Work

Our Gnutant application can be compared with existing document sharing systems. In Gnutella [9], queries are text strings transmitted through broadcasting: each node receiving a query forwards it to all its neighbors. Being based on broadcasting, Gnutella is prone to serious scalability problems, and to avoid an exponential growth in the number of messages exchanged, strict limits are imposed on the TTL of messages and the number of neighbors known to each node. Unfortunately, these limits restrict the reach of a Gnutella query and thus the number of matching replies. Gnutant inherits the free search capability of Gnutella, without relying on inefficient broadcasting techniques.

In Freenet [3], each document is associated with a key obtained by hashing the document name. Search requests contain a single key, representing the desired document. Requests are not broadcast; instead, they are routed through the network using information gathered by previous requests. Freenet routing is based on the closeness between keys: if a node is unable to satisfy a request locally, it is forwarded to the node that is believed to store documents whose keys are closest to the requested key. The main limitation of Freenet is that queries are limited to documents with well-known names. Anthill adopts a routing technique similar to that of Freenet, but adds the possibility of performing free search queries by hashing a set of keywords associated with the document, possibly extracted from the filename, rather than using the complete filename. Another major difference between Gnutant and Freenet is that Gnutant builds a distributed document index, and do not move the documents unless requested by the user. From this perspective, Gnutant can be viewed as a distributed search engine.

Despite the fact that Gnutant was designed for document sharing, the underlying mechanism could be used to implement a generic lookup service with an interface similar to that of the Chord system [4]. Our lookup service would be probabilistic, since we cannot guarantee that even existing name-value associations will be returned when performing a lookup. Using an ant-based implementation of a distributed lookup service provides ad hoc replication of the name-value associations, rather than uniform replication as provided by Chord. The drawback of uniform replication in a highly dynamic P2P system is that it requires a fairly complex rearrangement of replicas. Using ad hoc replication has the advantage that popular lookups will return quickly, while the drawback is that some (unpopular) index data may be lost due to lack of storage space at peers. Loss of index data may be resolved by reinserting the association at periodic intervals.

5 Conclusions

In this paper we have presented Gnutant, a novel document sharing application implemented using the Anthill framework. Gnutant provides users with free search capabilities, without leading to strict limits on the overall scalability of the system. Preliminary simulation results indicate that Gnutant performs well with respect to finding documents.

In addition to its novelty, Gnutant also demonstrates the simplicity of developing P2P applications using Anthill. We plan to use Anthill to study and evaluate properties of several existing P2P algorithms. We are implementing ants that mimic the behavior of Freenet, for the purpose of comparing it with Gnutant and studying how the reliability, availability and performance of hash-based routing may be improved.

We also plan to perform additional simulations with Gnutant, to reveal its ability to tolerate dynamic environments and to determine its impact on the network load. Further analysis is also required to determine the propagation time for replicas of a new document, relative to the life-time of the content in distributed index and routing storages.

In addition to our work on Gnutant, we also continue the development of the Anthill framework. In particular we are investigating extensions to allow ant algorithm parameterization to be evolved through evolutionary techniques.

References

- [1] David Anderson. SETI@home. In Andy Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 5. O'Reilly & Associates, March 2001.
- [2] Özalp Babaoğlu, Hein Meling, and Alberto Montresor. Building Peer-to-Peer Systems with Anthill. Submitted for publication, July 2001.
- [3] Ian Clarke, Oskar Sandberg, Branden Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
- [4] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society.
- [5] FIPA. FIPA Peer-to-Peer Positioning Paper. Technical Report F-OUT-00076, Foundation for Intelligent Physical Agents, December 2000.
- [6] Stanley P. Franklin and Arthur C. Graesser. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35, 1996.
- [7] Groove Networks. <http://www.groove.net>.
- [8] Project JXTA. <http://www.jxta.org>.
- [9] Gene Kan. Gnutella. In Andy Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 8. O'Reilly & Associates, March 2001.
- [10] Peer-to-Peer Working Group. <http://www.p2pwg.org>.
- [11] Clay Shirky. Listening to Napster. In Andy Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 2. O'Reilly & Associates, March 2001.
- [12] Tony White and Bernard Pagurek. Towards Multi-Swarm Problem Solving in Networks. In *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS)*, 1998.
- [13] Tony White and Bernard Pagurek. Emergent Behavior and Mobile Agents. In *Proceedings of the Workshop on Mobile Agents in the Context of Competition and Cooperation at Autonomous Agents*, 1999.