

The background of the page features a large, faint, golden seal of the University of Bologna. The seal is circular and contains a central figure, likely a saint or scholar, surrounded by Latin text. The outer ring of the seal reads "UNIVERSITAS BOLOGNENSIS" and "MDCCCXXXIII". The inner ring contains the motto "SAPIENTIA ET VERITAS". The seal is centered on the page.

Anthill: a Framework for the Development of Agent-Based Peer-to-Peer Systems

Özalp Babaoğlu Hein Meling Alberto Montresor

Technical Report UBLCS-2001-09

November 2001

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address *last-name@cs.unibo.it*. Questions and comments should be addressed to `tr-admin@cs.unibo.it`.

Recent Titles from the UBLCS Technical Report Series

- 2000-2 *Compact Net Semantics for Process Algebras*, Bernardo, M., Busi, N., Ribaudò, M., March 2000 (Revised December 2000).
- 2000-3 *An Asynchronous Calculus for Generative-Reactive Probabilistic Systems*, Aldini, A., Bravetti, M., May 2000 (Revised September 2000).
- 2000-4 *On Securing Real-Time Speech Transmission over the Internet*, Aldini, Bragadini, Gorrieri, Rocchetti, May 2000.
- 2000-5 *On the Expressiveness of Distributed Leasing in Linda-like Coordination Languages*, Busi, N., Gorrieri, R., Zavattaro, G., May 2000.
- 2000-6 *A Type System for JVM Threads*, Bigliardi, G., Laneve, C., June 2000.
- 2000-7 *Client-centered Load Distribution: a Mechanism for Constructing Responsive Web Services*, Ghini, V., Panzieri, F., Rocchetti, M., June 2000.
- 2000-8 *Design and Analysis of RT-Ring: a Protocol for Supporting Real-time Communications*, Conti, M., Donatiello, L., Furini, M., June 2000.
- 2000-9 *Performance Evaluation of Data Locality Exploitation* (PhD Thesis), D'Alberto, P., July 2000.
- 2000-10 *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems* (PhD Thesis), Montresor, A., July 2000.
- 2000-11 *Coordination: An Enabling Technology for the Internet* (PhD Thesis), Rossi, D., July 2000.
- 2000-12 *Coordination Models and Languages: Semantics and Expressiveness* (PhD Thesis), Zavattaro, G., July 2000.
- 2000-13 *Jgroup Tutorial and Programmer's Manual*, Montresor, A., October 2000.
- 2000-14 *A Declarative Language for Parallel Programming*, Gaspari, M., October 2000.
- 2000-15 *An Adaptive Mechanism for Securing Real-time Speech Transmission over the Internet*, Aldini, A., Gorrieri, R., Rocchetti, M., November 2000.
- 2000-16 *Enhancing Jini with Group Communication*, Montresor, A., Babaoglu, O., Davoli, R., December 2000 (Revised January 2001).
- 2000-17 *Online Reconfiguration in Replicated Databases Based on Group Communication*, Kemme, B., Bartoli, A., Babaoglu, O., December 2000 (Revised March 2001).
- 2001-1 *Design and Analysis of Protocols and Resources Allocation Mechanisms for Real-Time Applications* (Ph.D. Thesis), Furini, M., January 2001.
- 2001-2 *Formalization, Analysis and Prototyping of Mobile Code Systems* (Ph.D. Thesis), Mascolo, C., January 2001.
- 2001-3 *Nature-Inspired Search Techniques for Combinatorial Optimization Problems* (Ph.D. Thesis), Rossi, C., January 2001.
- 2001-4 *Desktop 3d Interfaces for Internet Users: Efficiency and Usability Issues* (Ph.D. Thesis), Pittarello, F., January 2001.
- 2001-5 *An Expert System for the Evaluation of EDSS in Multiple Sclerosis*, Gaspari, M., Roveda, G., Scandellari, C., Stecchi, S., February 2001.
- 2001-6 *Probabilistic Information Flow in a Process Algebra*, Aldini, A., April 2001 (Revised September 2001).
- 2001-7 *Architecting Software Systems with Process Algebras*, Bernardo, M., Ciancarini, P., Donatiello, L., July 2001.
- 2001-8 *Non-determinism in Probabilistic Timed Systems with General Distributions*, Aldini, A., Bravetti, M., July 2001.

Anthill: a Framework for the Development of Agent-Based Peer-to-Peer Systems

Özalp Babaoğlu * Hein Meling ‡ Alberto Montresor *

Abstract

Peer-to-peer (P2P) systems are characterized by decentralized control, large scale and extreme dynamism of their operating environment. As such, they can be seen as instances of *complex adaptive systems* (CAS) typically found in biological and social sciences. In this paper we describe Anthill, a framework to support the *design, implementation and evaluation* of P2P applications based on ideas such as multi-agent and evolutionary programming borrowed from CAS. An Anthill system consists of a dynamic network of peer nodes (nests); societies of adaptive agents (ants) that travel through this network, interacting with nodes and cooperating with other agents in order to solve complex problems. Anthill can be used to construct different classes of P2P services that exhibit resilience, adaptation and self-organization properties. We describe preliminary experiences with using Anthill to implement a document sharing application.

1 Introduction

Informally, peer-to-peer systems are distributed systems where all nodes are *peers* in the sense that they have equal role and responsibility. In fact, distributed computing was intended to be synonymous with peer-to-peer computing long before the term was invented, but this initial desire was subverted by the advent of client-server computing popularized by the World Wide Web. The modern use of the term peer-to-peer (P2P) and distributed computing as intended by its pioneers, however, differ in several important aspects. First, P2P applications reach out to harness the outer edges of the Internet and consequently involve scales that were previously unimaginable. Second, the environments in which P2P applications are deployed exhibit extreme dynamism in structure, content and load. Finally, P2P by definition, excludes any form of centralized structure, requiring control to be completely decentralized.

Since the original file sharing applications Napster [17], Gnutella [8] and Freenet [2], a flurry of recent P2P projects in different application domains have come to being. These include object location services like Pastry [15] and Tapestry [19], persistent storage services such as Past [16] and Oceanstore [9], distributed lookup services like CAN [14] and Chord [3]. Despite their differences, what unites these and all other P2P systems is the extreme dynamism of their operating environment. The topological structure of the underlying network typically changes rapidly due to nodes voluntarily joining or leaving the system or due to involuntary events such as node crashes, network partitions and recoveries. The load in the system may shift rapidly from one region to another, for example, as certain documents

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

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

become “hot” in a document sharing system; or as new documents are inserted and removed by certain nodes.

Traditional techniques that are typically adequate for building distributed applications are not satisfactory in dealing with the scale and dynamism that characterize the operating environments of P2P systems. For example, Pastry requires modifications to the routing tables of several specific nodes when a failure occurs [15]. In the Past system, documents have a fixed number of replicas, irrespective of their popularity [16]. Satisfying the needs of P2P application development requires a paradigm shift that puts adaptation, resilience and self-organization as primary concerns.

In this paper, we suggest that *complex adaptive systems* (CAS) used to explain the behavior of certain biological, social and economical systems can be the basis of a programming paradigm for P2P applications. In the CAS framework, a system consists of large numbers of autonomous agents that individually have very simple behavior and that interact with each other in very simple ways. Despite the simplicity of its components, CAS exhibit what is called *emergent behavior* that is surprisingly complex and unpredictable. Furthermore, the collective behavior of CAS is highly adaptive to changing environmental conditions or unforeseen scenarios, is resilient to failures and self-organizes towards globally near-optimal configurations. What renders CAS particularly attractive from a P2P perspective is the fact that these global properties are achieved without explicitly “programming” them into the individual agents.

In order to pursue these ideas, we have built *Anthill*, a novel framework for P2P application development, deployment and testing. Anthill is middleware that is written in Java and based on the *multi-agent system* (MAS) paradigm [5]. MAS systems are collections of *autonomous agents* that can observe their environment and perform simple local computations leading to actions based on these observations. The behavior of an agent may be non-deterministic and its actions may modify the environment as well as the agent’s location within the environment. What distinguishes MAS from other agent models is that there is no central coordination of activity. In the context of MAS, emergent behavior manifests itself as *swarm intelligence* whereby the collection of simple agents of limited individual capabilities achieves “intelligent” collective behavior [18]. Ant colonies, which are natural instances of MAS, are known to be capable of solving complex optimization problems including those arising in communication networks [1]. MAS are particularly suited for applications that are to be deployed in highly dynamic environments [4], subject to incomplete and imprecise information [18]. As such, the MAS paradigm is an appropriate basis for modeling and building P2P applications.

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 requests originating at users by generating one or more *ants* — autonomous agents that travel across the nest network trying to satisfy the request. Ants interact only indirectly with each other by modifying their environment through information stored in the visited nests. This form of indirect communication, used also by real ants, is known as *stigmergy* [18].

Anthill simplifies P2P application development and deployment by freeing the programmer of all low-level details including communication, security and ant scheduling. Developers wishing to experiment with new P2P protocols need to focus only on writing appropriate ant algorithms using the Anthill API and defining the structure of the P2P system. Anthill includes a simulation environment to help developers analyze and evaluate the behavior of P2P systems. Simulation parameters, such as the structure of the network, the ant algorithms to be deployed, characteristics of the workload presented to the system, and properties to be measured, are all defined using XML. Unlike other toolkits for multi-agent simulation [10], Anthill uses a single ant implementation in both the simulation and actual execution environ-

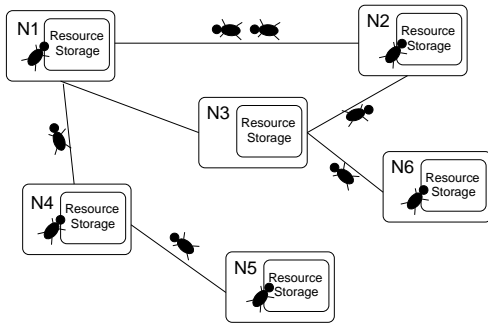


Figure 1: Overview of a nest network.

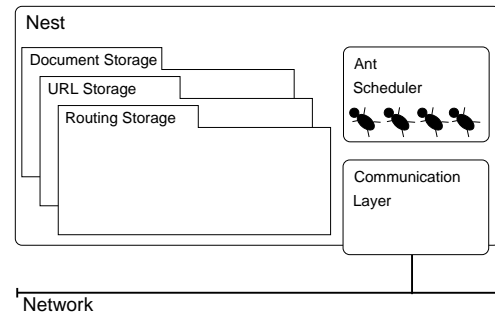


Figure 2: The architecture of a nest.

ments, thus avoiding the cost of re-implementing ant algorithms before deploying them. This important feature has been obtained by careful design of the Anthill API and by providing two distinct implementations of it for simulation and deployment.

In addition to the adaptation properties derived from its multi-agent structure, Anthill pushes the analogy with natural systems even further by “evolving” ant algorithms to better adapt to certain tasks. This is accomplished through evolutionary computing techniques such as genetic algorithms [11] within the simulation environment. The set of parameters that define the behavior of an ant algorithm are considered its “genetic code” and the system automatically evolves ant populations so that successive generations improve upon an appropriate fitness measure.

In order to test our ideas regarding P2P as CAS, we have used Anthill to build a simple document sharing application called *Gnutant*. There is no doubt that building on top of Anthill has simplified the implementation. But more importantly, the resulting system indeed exhibits adaptiveness with respect to a variety of conditions and continues to improve its performance as time goes on, despite starting from a state of total ignorance. *Gnutant* itself is of interest as it combines the best characteristics of the two popular document sharing systems Gnutella and Freenet.

2 The Anthill Model

The Anthill model is based on two logical entities: *nests* and *ants*. A P2P distributed system based on Anthill is composed of a self-organized overlay network of interconnected nests. The network is characterized by the absence of a fixed structure, as nests come and go and discover each other on top of a communication substrate. Each nest is a middleware software capable of performing computations and hosting resources. Any machine that is connected to the Internet and runs the Anthill software can act as a nest. Each nest interact with local instances of one or more *applications* and provides them with a collection of *services*. An example application may be a GUI-based file-sharing system, while a service could be a distributed indexing service used by the file-sharing application to locate files. Nests handle requests coming from applications by generating one or more *ants* — autonomous agents that travel across the nest network, interacting with nests that they visit in order to accomplish their task.

Applications exploit services provided by 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.

```

public interface Nest {
    void request(Request request,
        ReplyListener listener);
    void addService(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();
    NestId getNestId();
    int getTTL();
    void reply(ReqId rid, Reply reply);
}

```

Figure 4: The **AntView** interface.

Anthill does not specify which services a nest should provide, nor impose any particular format on request and replies. The provision of services and the interpretation of requests are completely delegated to ants.

When a nest receives a request from the local application, an appropriate service for handling the request, is selected from the set of available services. The set of available services is dynamic, as new services may be installed by the user. Each service is implemented by one or more ant algorithms.

2.1 The Nest

Figure 2 illustrates the architecture of a nest that is composed of three logical modules: resource storages, ant scheduler and communication layer. Resource storage modules are specialized depending on the type of information they contain. Each storage module type is associated with 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 items in a document storage when space is needed for new documents. Each service installed by a nest is associated with its own resource storages modules; for example, the nest in Figure 2 may represent a node of a file-sharing application based on a distributed index for document retrieval, in which the routing storage is used for ant's routing decisions, the document storage is used for maintaining shared documents and a URL storage is used to maintain the distributed index.

The *ant scheduler* module multiplexes 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 that 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.

The details of the Nest interface is shown in Figure 3. It contains the methods that may be invoked by the P2P application to interact with a nest. The main method of this interface is `request()`, that 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.

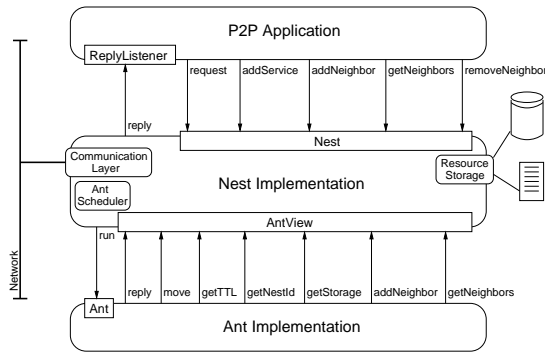


Figure 5: Overview of the Anthill interaction model.

2.2 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 (if the algorithm requires this) 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, that 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.

Ants never communicate with each other directly; instead, they communicate indirectly by leaving information related to the service they are implementing in the appropriate resource storages found in the visited nests. For example, an ant implementing a distributed lookup service may leave routing information that help subsequent ants to direct themselves toward the zone of the network that more likely contains the searched key.

The behavior of an ant is determined by its current state, the content of the resource storages found in its current nest and its algorithm, that 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 the Ant interface shown in Figure 5. The `run()` method contains the ant algorithm and is executed at each nest visited during the ant's trip.

The set of actions available to ants is limited to those included in interface `AntView`, shown in Figure 4. Among these actions, ants are enabled to move to other nests, obtain access to local resource storages, obtain information about the identifiers of the local nest and its neighbors, modify the list of neighbors nests of the nest and the identifiers of the neighbor nests, and finally notify the nest about a reply for a request originated in this nest.

Figure 5 gives a structural overview of the interactions between the various Anthill components. In addition to building a P2P application using the nest interface and the reply listener, the developer must also implement a set of ant algorithms facilitating services needed by that particular P2P application.

3 The Anthill Framework

In this section, we discuss the characteristics of the *runtime* and the *simulation* environments included in Anthill. These two distinct implementations of the Anthill model are used for real network deployment and for evaluation purposes, respectively. Unlike other toolkits for multi-agent simulation [10], Anthill uses a single ant implementation in both the simulation and real network environments, thus avoiding the cost of re-implementing ant algorithms

when deploying them and promoting the rapid deployment of prototype ant algorithms in the Internet.

3.1 The Runtime Environment

A prototype of the runtime environment is currently under development. This section contains a brief overview of the planned system. The runtime environment is a distributed implementation of the Anthill model, that provides low-level functions such as communication, security, resource management and ant scheduling. The runtime environment is based on JXTA [7], that is an open-source P2P project promoted by Sun Microsystems. JXTA is aimed at establishing a network programming platform for P2P systems, by identifying a small set of basic facilities necessary to support P2P applications and providing them as building blocks for higher-level functions. Although we are interested only in the Java version of JXTA, there are other implementations ("bindings") for different programming languages. Interoperability between the different implementations are guaranteed by the use of XML as the low-level formatting systems for messages exchanged between JXTA peers.

The benefits of founding our implementation on JXTA are several. For example, JXTA provides the possibility of using different transport layers for communication, including TCP/IP and HTTP, and is capable of handling the problems related to firewalls and NAT. This will spare our implementation from these low-level details. Furthermore, we will exploit the complex security architecture that is being developed for JXTA.

The JXTA middleware is composed of three layers. At the bottom is the *JXTA core*, that deals with low-level functions such as peer establishment, peer discovery, communication management and routing. The *JXTA services* are built on top of the core and deals with higher-level concepts, such as indexing, searching, and file sharing. These services, although useful by themselves, are used by *JXTA applications* to build high-level applications like emailing, auctioning and persistent storage.

The runtime environment of Anthill is designed as a JXTA service and exploits the facilities offered by the JXTA core to provide an infrastructure for the construction of ant-based P2P distributed applications. It implements the Nest interface, providing methods for performing generic requests to Anthill applications. This nest implementation includes an ant scheduler capable of multiplexing the Java virtual machine among multiple visiting ants. Using the security model of Java, the execution of ants is confined to a controlled environment ("sandbox") by limiting their interactions with the local nest to those included in the AntView interface. A number of disk- and memory-based resource storage implementations are provided, enabling the ant algorithms to make use of predefined and pre-installed classes in the visited nests. Nevertheless, it is also possible for ants to construct their own specialized resource storages.

The communication layer is based on some of the fundamental primitives offered by the JXTA core, namely pipes, peer groups and advertisements. *Pipes* are communication channels for sending and receiving messages, and are used in the communication layer to move ants between nests. A *peer group* is a collection of cooperating peers providing a common set of services and speaking the same set of protocols. Peers may participate in several groups at the same time, thus offering several services. In Anthill, there will be a general peer group constituted by all of the peers that are executing the nest service, and several peer groups constituted by the set of nests that accept to execute the algorithm of a particular ant. Owners of peers will be able to decide which kind of services their machines are going to offer, by accepting or rejecting the installation of new ant algorithms. Using the features of the Java virtual machine, we are implementing a simple class loader capable to download the code of unknown ants from remote sites and cache ant classes on local disks so as to avoid repeated downloads. Finally, *advertisements* are XML structured documents that name, describe and

publish the existence of a resource, such as a peer, a peer group, a pipe, or a service. Advertisements are used by the JXTA discovery protocol to locate services. In Anthill, they are used to advertise peer groups of nests offering a particular service.

3.2 The Simulation Environment

To evaluate ant algorithms, Anthill includes a simulation environment through which the behavior of a particular ant implementation may be simulated and assessed. Simulating different P2P applications requires developing appropriate ant algorithms and a corresponding request generator characterizing user interactions with the application. Each simulation study, called an *experiment*, is specified using XML by defining parameters for the nest network, the request generator and the ant algorithm. Figure 6 shows an excerpt from a simulation configuration file. By specifying the simulation parameters in this way, experiments can be built at run-time by assembling a collection of component classes, thus simplifying the process of evaluating ant algorithms.

Several experiment implementations may be written to assess the behavior of ant algorithms in different scenarios. The configuration file shown in Figure 6, the generic experiment interface is implemented by the *TimeExperiment* class, that enables to monitor how the behavior of an ant algorithm evolves with time. In the example, we have specified the number of iterations of the algorithm, the number of rounds per iterations, and how many times the experiment should be repeated to obtain average values. Other experiment implementations may test the algorithms in different conditions, for example trying to determine the behavior of the system after an initialization phase. The actual sequence of requests that are to be satisfied are generated on-the-fly during the simulation through the provided request generator. In the example, we use the *GnutantGenerator* class that is specific for the Gnutant algorithm presented in the next section.

The nest network is simulated by class *PeernetImpl* and it 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 inside a single Java virtual machine 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 build other simulated network implementation with dynamic behavior by specifying the total number of nests and the neighbor degree as *probability distributions* rather than static values. In addition, we are also investigating how we can distribute the simulated nest network on a cluster of Java virtual machines, to remedy problems with scaling the simulation environment to millions of nodes. The scalability problems of the simulation environment are due to the relatively large memory footprint required by nest implementation.

Resource storages are implemented as simple data structures with a maximum capacity and associated replacement policies. Finally, we may also specify additional parameters, that may depend on the particular ant algorithm being simulated. In the example, we have modified the TTL parameter of to adjust the reach of each ant.

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. The simulation environment enable programmers to evaluate several different experiments and obtain average figures for the collected statistics. Monitoring network traffic is not performed at the packet level, but rather at the ant level. That is, measuring the number of ants sent between various nests in the system. This approach is reasonable since the size of the typical ant will be less than the nominal packet size.

In the simulation environment, the nest's communication layer is simpler to implement than the one included in the runtime environment, since remote communication can be

```

<Create interface="antsim.Experiment" class="antsim.impl.TimeExperiment">
  <Argument name="iterations" type="int" value="100"/>
  <Argument name="rounds" type="int" value="20"/>
  <Argument name="repetitions" type="int" value="20"/>
</Create>
<Create interface="antsim.Generator" class="gnutant.GnutantGenerator">
  <Argument name="nrequests" type="int" value="2"/>
  <Argument name="keywordFile" type="String" value="keywords.txt"/>
</Create>
<Create interface="UrlMemoryStorage" class="storage.impl.MemoryStorage">
  <Argument name="capacity" type="int" value="64"/>
</Create>
<Create interface="DocMemoryStorage" class="storage.impl.MemoryStorage">
  <Argument name="capacity" type="int" value="8"/>
</Create>
<Create interface="antsim.Peernet" class="antsim.impl.PeernetImpl"/>
  <Argument name="size" type="int" value="2000"/>
  <Argument name="degree" type="int" value="6"/>
</Create>
<Config name="NestTTL" type="int" value="100"/>

```

Figure 6: Example XML configuration.

achieved through local interactions instead. 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. The simulation environment runs a centralized scheduler that uses the provided request generator to create requests and in a round-robin fashion invokes the ants run() method.

4 Using Genetic Techniques to Evolve Ant Algorithms

In Anthill, we further exploit the “nature” metaphor by using evolutionary techniques such as genetic algorithms in designing ant algorithms [11]. Ant algorithms, which govern the behavior of ants, can be parameterized in various ways. For example, an ant may deterministically choose to follow what is reputed to be the best path to a resource, or probabilistically select any of the neighbors of its current nest, depending on some exploration parameter. The Anthill simulation environment has been extended to enable the definition of a collection of such parameters and the selection of the fittest set of parameters for a particular task. The fitness function can be configured using the XML file defined in the previous section, by defining the class responsible for the evaluation.

In addition to the off-line use of genetic techniques, we plan to investigate whether genetic techniques can also be applied at run-time. For example, in order to satisfy a request, a nest could launch several ants, each characterized by a different chromosome, and then rate them using a local fitness criterion. Subsequent requests could be delegated to ants derived genetically from those that were deemed fittest in previous requests. Nests could also “steal” the algorithms and chromosomes of visiting ants and use them in crossover and mutation techniques for generating new ants. It is interesting to note that the on-line evolutionary selection mechanism itself can be viewed as a P2P system whose task is to tune the ants of the original P2P application.

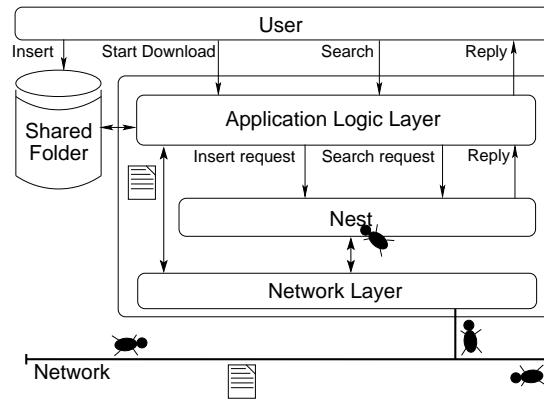


Figure 7: Gnutant Application Overview.

5 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 7 gives an overview of the Gnutant application. 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.

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.

5.1 The Gnutant Ant Algorithms

The Gnutant application is implemented using 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.

5.1.1 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*, that is similar to the routing technique used in Freenet [2]. 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.

5.1.2 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.

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);
  }
}

```

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

5.2 Preliminary 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

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}
      }
    }
  }
}
```

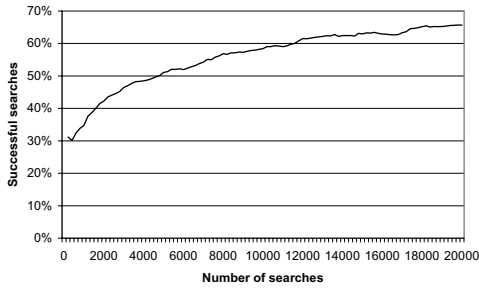


Figure 8: Search success rate.

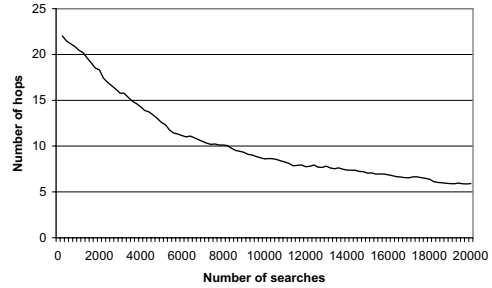


Figure 9: Number of hops until first reply.

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 capacity of the routing, document and URL storages were set to 16, 16, and 64 entries, respectively. All resource storages use the LRU replacement policy. Upon receiving a set of matching documents to a query, we simulated the download of 10% of the matching documents, followed by execution of an InsertAnt for each downloaded document. The simulation was repeated ten times in order to obtain average values.

The simulation results are shown in Figures 8 and 9. Figure 8 shows the success rate for search requests, while Figure 9 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. Although our simulation results are preliminary, they appear to be roughly comparable to the routing performance of Freenet [2], even though Freenet cannot deal with free search queries.

6 Related Work

The importance of the P2P distributed computing model was recently recognized by industry, leading to several standardization and infrastructure efforts, including JXTA and the Peer-to-Peer Working Group [13, 7]. Anthill differs from these industrial initiatives because one of its main goal is to support the scientific investigation of the properties of P2P systems, by providing a simulation testbed for prototyping and tuning their P2P algorithms. On the other hand, we are exploiting the rich facilities offered by JXTA [7] as a basis for the runtime implementation of the Anthill model.

Agent-based technology has been a very active area of research and development for the past decade, and has lead to the development of several commercial agent platforms, such as

Voyager [12]. Anthill's simulation environment can, to some extent, be compared with agent simulators such as Swarm [10] and MASS [6]. Swarm is a general purpose software package for simulating distributed artificial worlds. It provides a general architecture for problems that arise in a wide variety of disciplines and is particularly suitable for problems involving a large number of autonomous entities "living" in an environment. MASS (Multi-Agent System Simulator) is an agent simulator developed with the aim of accurately measuring the influence of different multi-agent coordination strategies in an unpredictable environment. Anthill differs from these systems, as they are only focused on simulation, and do not support deployment in a real network environment.

Our Gnutant application can be compared with existing document sharing systems. In Gnutella [8], 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 [2], 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.

Despite the fact that Gnutant was designed for document sharing, the underlying mechanism could be used to implement generic distributed lookup services like CAN [14] and Chord [3]. Our lookup services 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 key-value associations, rather than uniform replication as provided by Chord and CAN. 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.

7 Conclusions

In this paper we have presented Anthill, a framework supporting a new approach for building P2P applications based on the MAS paradigm, in which societies of autonomous agents cooperate in order to accomplish tasks assigned to them.

The Anthill project is still in its early stages. So far, we have implemented the simulation environment and we have used it to develop Gnutant, an ant algorithm implementing a document sharing application. The simulation environment includes also an evolutionary environment that enables programmers to genetically select the fittest ant algorithm for a particular task. We are currently building a prototype implementation of the runtime environment, based on JXTA [7]. In addition, work is under way to improve the simulation environment of Anthill by augmenting it with an interface for the graphical visualization of

the properties of the simulated ant algorithms.

We also plan to use Anthill to study and evaluate properties of several existing P2P algorithms, including Pastry and Tapestry [15, 19], Past and OceanStore [16, 9], Chord [3] and CAN [14]. 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. Finally, we plan to exploit evolutionary programming techniques to improve the performance of the resulting algorithms.

References

- [1] G. D. Caro and M. Dorigo. AntNet: Distributed Stigmergetic Control for Communications Networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.
- [2] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In H. Federrath, editor, *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
- [3] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society.
- [4] FIPA. FIPA Peer-to-Peer Positioning Paper. Technical Report F-OUT-00076, Foundation for Intelligent Pyhsical Agents, Dec. 2000.
- [5] S. P. Franklin and A. C. Graesser. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *Proc. of the 3rd Int. Workshop on Agent Theories, Architectures, and Languages*, pages 21–35, 1996.
- [6] B. Horling, V. Lesser, and R. Regis. Multi-Agent System Simulation Framework. In *Proc. of the 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, Aug. 2000.
- [7] Project JXTA. <http://www.jxta.org>.
- [8] G. Kan. Gnutella. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 8. O’Reilly & Associates, Mar. 2001.
- [9] J. Kubiawicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of the 9th International Conference on Architectural support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.
- [10] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm Simulation System, A Toolkit for Building Multi-Agent Simulations. Technical report, Swarm Development Group, June 1996. <http://www.swarm.org>.
- [11] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Apr. 1998.
- [12] ObjectSpace, Inc. Voyager – Mobile Java Agents. <http://www.objectspace.com/Voyager/>.
- [13] Peer-to-Peer Working Group. <http://www.p2pwg.org>.
- [14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM’01*, San Diego, CA, 2001.

- [15] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the 18th International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001. IFIP/ACM.
- [16] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Canada, Nov. 2001.
- [17] C. Shirky. Listening to Napster. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 2. O'Reilly & Associates, Mar. 2001.
- [18] T. White and B. Pagurek. Emergent Behavior and Mobile Agents. In *Proc. of the Workshop on Mobile Agents in the Context of Competition and Cooperation at Autonomous Agents*, 1999.
- [19] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, U.C. Berkeley, Apr. 2001.