Messor: Load-Balancing through a Swarm of Autonomous Agents

Alberto Montresor **Hein Meling** **Özalp Babaoğlu**

Technical Report UBLCS-2002-11

September 2002

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

Recent Titles from the UBLCS Technical Report Series

2001-2 *Formalization, Analysis and Prototyping of Mobile Code Systems* (Ph.D. Thesis), Mascolo, C., Janaury 2001.

2001-3 *Nature-Inspired Search Techniques for Combinatorial Optimization Problems* (Ph.D. Thesis), Rossi, C., Janaury 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.

2001-9 *Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems*, Babaoglu, O., Meling, H., Montresor, A., November 2001 (Revised September 2002).

2002-1 *A Timed Join Calculus*, Bünzli, D. C., Laneve, C., February 2002.

2002-2 *A Process Algebraic Approach for the Analysis of Probabilistic Non-interference*, Aldini, A., Bravetti, M., Gorrieri, R., March 2002.

2002-3 *Quality of Service and Resources' Optimization in Wireless Networks with Mobile Hosts* (Ph.D Thesis), Bononi, L., March 2002.

2002-4 *Specification and Analysis of Stochastic Real-Time Systems* (Ph.D. Thesis), Bravetti, M., March 2002.

2002-5 *QoS-Adaptive Middleware Services* (Ph.D. Thesis), Ghini, V., March 2002.

2002-6 *Towards a Semantic Web for Formal Mathematics* (Ph.D. Thesis), Schena, I., March 2002.

2002-7 *Revisiting Interactive Markov Chains*, Bravetti, M., June 2002.

2002-8 *User Untraceability in the Next-Generation Internet: a Proposal*, Tortonesi, M., Davoli, R., August 2002.

2002-9 *Towards Adaptive, Resilient and Self-Organizing Peer-to-Peer Systems*, Montresor, A., Meling, H., Babaoglu, O., September 2002.

2002-10 *Towards Self-Organizing, Self-Repairing and Resilient Large-Scale Distributed Systems*, Montresor, A., Babaoglu, O., Meling, H., September 2002.

2002-11 *Messor: Load-Balancing through a Swarm of Autonomous Agents*, Montresor, A., Meling, H., Babaoglu, O., September 2002.

Messor: Load-Balancing through a Swarm of Autonomous Agents

Alberto Montresor [1] **Hein Meling** [2] **Özalp Babaoğlu** [1]

Technical Report UBLCS-2002-11

September 2002

Abstract

Peer-to-peer (P2P) systems are characterized by decentralized control, large-scale and extreme dynamism of their environment. Developing applications that can cope with these characteristics requires a paradigm shift that puts adaptation, resilience and self-organization as primary concerns. Complex adaptive systems *(CAS), commonly used to explain the behavior of many biological and social systems, could be an appropriate response to these requirements. In order to pursue these ideas, this paper presents Messor, a decentralized load-balancing algorithm based on techniques such as multi-agent systems drawn from CAS. A novel P2P grid computing system has been designed using the Messor algorithm, allowing arbitrary users to initiate computational tasks.*

1. Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy), Email: {montresor,babaoglu}@CS.UniBO.IT
2. Department of Telematics, Norwegian University of Science and Technology, O.S. Bragstadsplass 2A, N-7491 Trondheim (Norway), Email: meling@item.ntnu.no

1 Introduction

Informally, *peer-to-peer* (P2P) systems are distributed systems based on the concept of resource sharing by direct exchange between *peer* nodes (i.e., nodes having the same role and equal responsibility). Exchanged resources include content, as in popular P2P file sharing applications [26, 13, 15], and storage capacity or CPU cycles, as in computational and storage grid systems [1, 23, 14].

Distributed computing was intended to be synonymous with P2P 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 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, P2P by definition, excludes any form of centralized structure, requiring control to be completely decentralized. Finally, and most importantly, the environments in which P2P applications are deployed exhibit extreme dynamism in structure and load. The topology of the system typically changes rapidly due to nodes voluntarily coming and going or due to involuntary events such as crashes and partitions. The load in the system may also shift rapidly from one region to another, for example, as certain files become "hot" in a file sharing system; or the computing needs of a node suddenly increase in a distributed computing system.

In order to deal with the scale and dynamism that characterize P2P systems, a paradigm shift is required that includes self-organization, adaptation and resilience as fundamental properties. We believe that *complex adaptive systems* (CAS), commonly used to explain the behavior of certain biological and social systems, can be the basis of a new programming paradigm for P2P applications. In the CAS framework, a system consists of a large number of relatively simple autonomous computing units, or *agents*. CAS typically exhibit what is called *emergent behavior*: agents, taken individually, may be easily understood, while the behavior of the system as a whole defies simple explanation. In other words, the interactions among agents, in spite of their simplicity, can give rise to richer and more complex patterns than those generated by single agents in isolation.

As an instance of CAS drawn from nature, consider an ant colony. Several species of ants, in particular those belonging to the *Messor Sancta* [5] species, are known to group objects in their environment (e.g., dead corpses) into piles so as to clean up their nests. Observing this behavior, one could be mislead into thinking that the cleanup operation is being coordinated by some "leader" ants. Resnick [22] describes an artificial ant colony exhibiting this very same behavior in a simulated environment. Resnick's artificial ant follows three simple rules: (i) wanders around randomly, until it encounters an object; (ii) if it was carrying an object, it drops the object and continues to wander randomly; (iii) if it was not carrying an object, it picks the object up and continues to wander. Despite their simplicity, a colony of these "unintelligent" ants is able to group objects into large clusters, independent of their initial distribution.

What renders CAS particularly attractive from a P2P perspective is the fact that global properties like adaptation, self-organization and resilience are achieved without explicitly embedding them into the individual agents. In the above example, there are no rules specific to initial conditions, unforeseen scenarios, variations in the environment or presence of failures. Yet, given large enough colonies, the global behavior is surprisingly adaptive and resilient.

In order to pursue these ideas, we have developed *Anthill* [3], a novel framework for P2P application development based on ideas such as multi-agent systems and evolutionary programming borrowed from CAS [27, 19]. The goals of Anthill are to provide an environment that simplifies the design and deployment of P2P systems based on these paradigms, and to provide a "testbed" for studying and experimenting with CAS-based P2P systems in order to understand their properties and evaluate their performance. An Anthill system is composed of a collection of interconnected *nests*. Each nest is a peer entity that makes its storage and computational resources available to swarms of *ants* – autonomous agents that travel across the network trying to satisfy user requests. During their life, ants interact with services provided by visited nests, such as storage management and ant scheduling.

Details of the design and implementation of Anthill can be found in a companion paper [3]. After having developed a prototype of Anthill, we are now in the process of testing the viability of our ideas regarding P2P as CAS by developing common P2P applications like file sharing [20] and grid computing over Anthill. In this paper, we present one of such application, called Messor. Messor is a grid computing system aimed at supporting the concurrent execution of highly-parallel, time-intensive computations, in which the workload may be decomposed into a large number of independent jobs. The computational power offered by a network of Anthill nests is exploited by Messor by assigning a set of jobs comprising a computation to a dispersed set of nests. To determine how to balance the load among the computing nodes, Messor use an algorithm inspired by the behavior of the artificial ant described above: Messor ants drop objects they are carrying only after having wandered about randomly "for a while" without encountering object concentrations. Colonies of such Messor ants try to disperse objects (more specifically, jobs) uniformly over their environment, rather than clustering them into piles.

Several computations can be profitably supported by Messor [1, 8, 2]. For example, in the Seti@Home project [1], the enormous amount of radio signals registered by radio telescopes are subdivided into a large number of data sets, that can be independently analyzed in the search for evidence of extra-terrestrial intelligence; Distributed.net [8] is an umbrella for several distributed computing projects, including cryptography challenges in which brute-force attacks are performed by subdividing key spaces into independent portions; the Anthrax Project [2] is an effort designed to help scientists to find a treatment for the Anthrax toxin, by performing screening analysis of large sets of molecules.

All these projects are based on a master-slave architecture, in which only the master node is enabled to generate and assign new jobs. Slave machines are relegated to a role of mere executors, thus in some sense betraying the peer-to-peer philosophy. Messor is completely decentralized, allowing every node in the system to generate new jobs and submit them to the network. An application designed in this way may be interesting for groups of entities that want to share their resources in order to exploit the resulting computing power cost effectively.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the Anthill framework. In Section 3 we provide an informal system model and specification for Messor followed by an architecture description and the details of the Messor load-balancing algorithm. Section 4 present preliminary simulation results, and Section 5 concludes the paper.

2 Anthill

Anthill uses terminology derived from the ant colony metaphor. An Anthill system is composed of a self-organizing overlay network of interconnected *nests*, as illustrated in Figure 1. Each nest is a middleware layer capable of hosting resources and performing computations. The network is characterized by the absence of any fixed structure, as nests come and go and discover each other on top of a communication substrate. Nests interact with local instances of one or more *applications* and provide them with a set of *services*. Applications are the interface between the user and the P2P network, while services have a distributed nature and are based on the collaboration among nests. An example application may be a file-sharing system, while a service could be a distributed indexing service used by the file-sharing application to locate files.

An application performs *requests* and listens for *replies* through its local nest. Requests and replies constitute the interface between applications and services. 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. This set is dynamic, as new services may be installed by the user. Services are implemented by means of *ants*, autonomous agents able to travel across the nest network. In response to a request, one or more ants are generated and assigned to a particular task. Ants may explore the network and interact with the nests that they visit in order to accomplish their goal. Anthill does not specify which services a nest should provide, nor impose any particular format on requests. The provision of services and the interpretation of requests are delegated to ants.

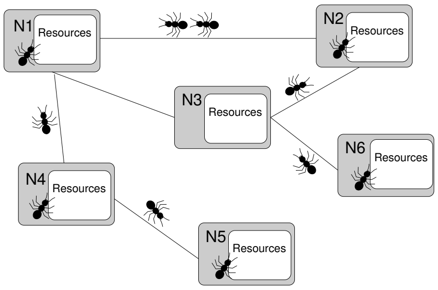

Figure 1: Overview of a nest network.

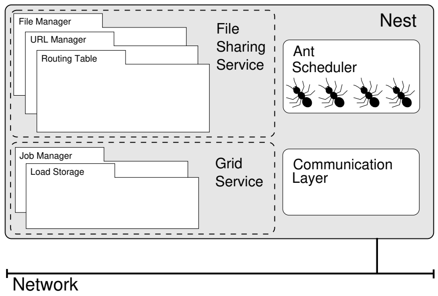

Figure 2: The architecture of a nest.

2.1 Nests

Figure 2 illustrates the architecture of a nest that is composed of three logical modules: ant scheduler, communication layer and resource managers. The *ant scheduler* module multiplexes the nest computation resource among visiting ants. It is also responsible for enforcing nest security by providing a "sandbox" for ants in order to limit the resources available to ants and prohibit ants from performing potentially dangerous actions (e.g., local file access).

The *communication layer* is responsible for network topology (neighbor) management and for ant movement between nests. In the network, each node has a unique identifier. In order to communicate with a remote node, its identifier must be known. The set of remote nests known to a node are called *neighbors* of that node. Note that the concept of neighborhood does not involve any distance metrics, since such metrics are application dependent and is more appropriately chosen by developers. The collection of neighbor sets defines the nest network that might be highly dynamic. For example, the communication layer may discover a new neighbor, or it may forget about a known neighbor if it is considered unreachable. Both the discovery and the removal processes may be either mediated by ants, or performed directly by the communication layer. In the former case, ants may report about new remote nodes they visited, or may fail to move to a neighbor because of a communication problem. In the latter case, the exact implementation of discovery and removal depends on the underlying communication substrate.

Nests offer their resources to visiting ants through one or more *resource managers*. Example resources could be files in a file-sharing system or CPU cycles in a computational grid, while the respective resource managers could be a disk-based storage manager or a job scheduler. Resource managers typically enforce a set of policies for managing the (inherently limited) resource. For example, a *least-recently-used* (LRU) policy may be used to discard items managed by a file manager when space is needed for new files. Each service installed by a nest is associated with a set of resource managers. For example, the nest in Figure 2 provides two distinct services: a file-sharing service based on a distributed index for file retrieval, in which a routing table is used by ants in making routing decisions, a file manager is used for maintaining shared files and a URL manager contains the distributed index; and a computational grid service, in which a job manager executes jobs assigned to it.

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 task 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. The ant algorithm is contained in a *run*() method, that is invoked at each visited nest. The ant code may be transmitted together with the ant state, if needed; appropriate code caching mechanisms are used to avoid to download the same algorithm more than once, and to update it when new versions are available.

Ants do not communicate directly with each other; instead, they communicate indirectly by leaving information related to the service they are implementing in the appropriate resource manager found in the visited nests. For example, an ant implementing a distributed lookup service may leave routing information that helps subsequent ants to direct themselves toward the region of the network that more likely contains the searched key. This form of indirect communication, used also by real ants, is known as *stigmergy* [11].

The behavior of an ant is determined by its current state, its interaction with resource managers 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.

2.3 The Anthill Framework

A Java prototype of the Anthill runtime environment [24] has been developed, based on JXTA [12], an open-source P2P project promoted by Sun Microsystems. JXTA is aimed at establishing a 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 services. The benefits of basing our implementation on JXTA are several. For example, JXTA allows the use of different transport layers for communication, including TCP/IP and HTTP, and deals with issues related to firewalls and NAT.

In addition to the runtime environment, Anthill includes a simulation environment to help developers analyze and evaluate the behavior of their P2P systems. Simulating different P2P applications require developing appropriate ant algorithms and a corresponding request generator characterizing user interactions with the application. Each simulation study is specified using XML by defining a collection of component classes and a set of parameters for component initialization. For example, component classes to be specified include the simulated nest network, the request generator to be used, and the ant algorithm to be simulated. Initialization parameters include the duration of the simulation, the network size, the failure probability, the number of requests to be generated, and the type and capacity of the resource managers to be used by ants. This flexible configuration mechanism enable developers to build simulations by assembling a collection of pre-defined and customized component classes, thus simplifying the process of evaluating ant algorithms.

Unlike other toolkits for multi-agent simulation [18], Anthill uses a single ant implementation in both the simulation and actual run-time environments, thus avoiding the cost of re-implementing ant algorithms before deploying them. This important feature has been achieved by a careful design of the Anthill API and by providing two distinct implementations of it for simulation and deployment.

3 Load Balancing in Messor

In this section, we present the Messor application and the services on which it relies. Messor is aimed at supporting the concurrent execution of highly-parallel, time-intensive computations, in which the workload can be decomposed into a large number of independent jobs.

3.1 System Model and Messor Specification

Messor nests may fail by *halting* whereby they stop prematurely to execute their algorithm. Halted nests may subsequently perform a *recovery* action, after which they lose any information stored in volatile memory, while they regain access to all data maintained in stable storage. We say that a nests is *crashed* if it halts and never perform a recovery action later. We say that a nest is *correct* if it is not crashed. All nests in the system may communicate directly with each other by using an underlying communication layer. Communication between nests may be disrupted by the presence of *partitionings*, that subdivide the network into a set of disjoint *partitions*. Any

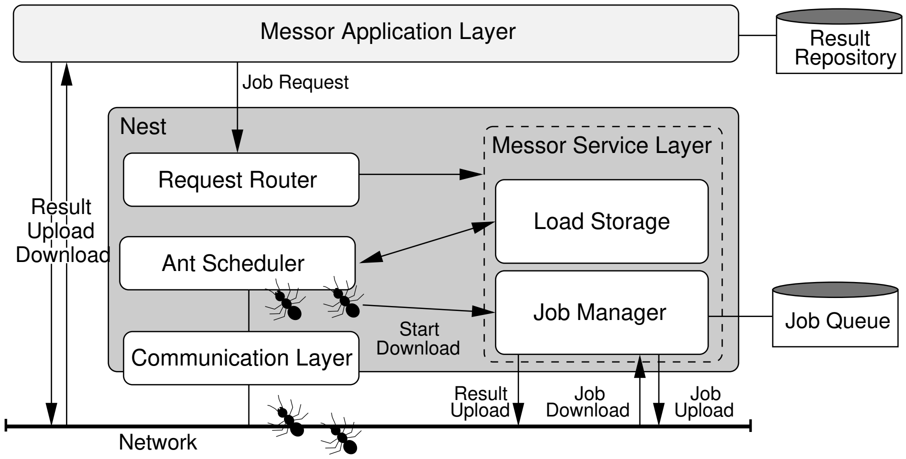

Figure 3. The architecture of a node supporting Messor

two nests belonging to the same partition may communicate reliably, while two nests in distinct partitions cannot communicate. Partitionings may be temporary due to subsequent repairs.

A Messor system is composed of a collection of interconnected Anthill nests configured to run the Messor software. Every such nest can submit *jobs* to the nest network, where each job is composed of some input data and the algorithm to be computed over these data. Jobs are scheduled and executed by the nest on which the job resides, by invoking the job algorithm. We say that a job is *completed* when its associated algorithm has been executed to completion. A completed job outputs a *result*, i.e. some data obtained from the job computation.

At each nest, Messor offers a very simple API to its users, enabling them to submit new jobs to be computed and collecting results once the jobs have been computed. The *originator* nest of a job is the nest where the job has been submitted. Once submitted, jobs may remain in the originator, or may be transferred to other nests in order to exploit their unused computational power. When a job is completed, the result is sent back to the originator. Once there, the user is either notified of the job result, or the result is stored locally; in the latter case, the user may periodically poll the nest to obtain the collected results. Messor guarantees that all jobs submitted to a *correct* originator will eventually be completed and their results delivered to the originator itself. Although this property may be satisfied by simply letting the correct originator compute all jobs, Messor attempts to disperse the load uniformly among cooperating nodes.

3.2 Messor Architecture

The architecture of a node supporting the Messor application is shown in Figure 3. Messor nodes are composed of two main layers:

- the Messor Application Layer is responsible for interacting with the local user by accepting requests and collecting computed results on her behalf; furthermore, it is also responsible for keeping track of job assignments, in order to re-insert in the system, jobs assigned to nodes that may have crashed.
- the Messor Service Layer is responsible for job execution and load balancing.

The Application Layer receives jobs from the user, and delivers them as job requests to the Request Router contained in the nest. This module analyze the request and routes it to the appropriate service among those installed in the nest. In the case of Messor, job requests are delivered to the Messor Service Layer.

In order to achieve its goals, the Application Layer maintains a database of jobs originated by the local user and their status with respect to the computation. The status may corresponds to the computed results, if available, or to the identifier of the nest to which the job has been assigned. Results computed by remote nests are downloaded by the Application Layer out-of-band, i.e. outside the ant communication mechanism offered by nests, for efficiency reasons. A lease mechanism is used to keep track of operational nodes, in order to identify crashed nodes and opportunely re-insert jobs assigned to them.

The Service Layer exploits the ant communication and scheduling facilities provided by nests. Two main resource managers are employed: the Load Storage contains information about the estimated load of remote nests, while the Job Manager is responsible for executing the jobs assigned to the local nest. The Load Storage implementation is memory-based; it is the main data structure maintained by visiting ants, and its utilization is explained in the next section. The Job Manager maintains a database of jobs to be computed by the local nest, implemented as a queue, and acts as a scheduler that selects the next job from the queue and executes it. Once computed, the Job Manager is responsible for uploading the result to the Application Layer of the job originator. Jobs are inserted in the job queue either after the Messor Service Layer have received a local request through the Request Router, or by downloading them from other nests. The download process is triggered by Messor ants, that are responsible for load balancing, while the actual download is performed out-of-band, without the mediation of ants.

3.3 The Messor Ant Algorithm

The most interesting component of Messor is its ant algorithm. In order to understand the basic idea behind Messor, consider the following variation of the artificial ant algorithm described in the introduction: (i) when an ant is not carrying any object, it wanders about randomly until it encounters an object and picks it up; (ii) when an ant is carrying an object, the ant drops it only after having wandered about randomly "for a while" without encountering other objects. Colonies of such ants try to disperse objects uniformly over their environment rather than clustering them into piles.

The algorithm of Messor ants is inspired by the rules described above. The environment in which Messor ants live is given by the network of nests. The objects to pick-up and drop-off correspond to the actual jobs, existing within the nest network. During its life-time, a Messor ant may assume two different states: SearchMax and SearchMin. While in SearchMax state, the ant wanders about in the network until it finds an "overloaded" nest; at that point, the ant records the identifier of this nest and switch to the SearchMin state. While in SearchMin state, the ant wanders about looking for an "underloaded" nest. When such a nest is found, the ant requests the local Job Manager to transfer jobs from the overloaded nest to the underloaded one, and then switches back to the SearchMax state again; and the process repeats. The transfer process is performed by direct downloading between the two nests; this to avoid carrying potentially large amounts of data representing jobs from one node to another while wandering about, searching for underloaded nodes.

The *load* of a nest is defined as the number of jobs currently in the job queue of that nest; alternatively, if information about the potential computing power needed to perform jobs is available, the load of a nest may be defined based on this information. The concepts of overloaded and underloaded nests are relative to the average load of the nests recently visited by an ant. This definition enable ants to make decisions about job transfers between nests with unbalanced loads on the basis of local information only, i.e. without global knowledge.

The SearchMax and SearchMin walks are not performed completely at random. When wandering, ants collect information about the load of the last visited nests. This information is then stored in the Load Storage component in each nest, and is used by subsequent ants to drive their SearchMax and SearchMin phases: at each step, the ant randomly selects the next node to visit among those that are believed to be more overloaded (in SearchMax) or underloaded (in SearchMin). In this way, ants move faster towards those regions of the network in which they are more interested. To avoid that the system become biased toward a subset of nests (those believed to be more over- or underloaded), ants may occasionally, based on an *exploration probability*, select the next nest using a uniform distribution, enabling the exploration of the entire network.

The Messor algorithm is shown in Figure 4. The state of each ant is represented by the set of

```
integer state = MAX;
Queue visits = new Queue(N);
integer maxLoad, minLoad;
NestId maxNest, minNest;

method doSearchMax() {
  if ((minLoad/maxLoad) ≤ TargetRatio[MAX]
    and not tossCoin(KeepSearchProb[MAX])) {
    state = MIN;
    doSearchMin();
  } else
    goNextNest();
}

method doSearchMin() {
  if ((minLoad/maxLoad) ≤ TargetRatio[MIN]
    and not tossCoin(KeepSearchProb[MIN])) {
    state = MAX;
    mng.forceTransfer(maxNest);
    clearMaxMin();
    doSearchMax();
  } else
    goNextNest();
}
```

```
method run(AntView view) {
  JobManager mng =
    view.getManager(JOBMANAGER);
  LoadStorage strg =
    view.getManager(LOADSTORAGE);
  initMaxMin(mng.getLoad(), view.getId());
  if (state == MAX)
    doSearchMax();
  elseif (state == MIN)
    doSearchMin();
}

method goNextNest() {
  if (tossCoin(ExplorationProb[state])) {
    list = strg.getNeighborList(view);
    nextNest= uniformRandom(list);
  } else {
    list = strg.getOrderedList(state, view);
    nextNest= normalRandom(list, Dev[state]);
  }
  strg.addLoads(visits);
  visits.add(mng.getLoad(), view.getId());
  view.move(nextNest);
}
```

Figure 4. Pseudo-code description of the algorithm

variables listed in the preamble. The current state (SearchMax vs SearchMin) of the ant is stored in variable state. A circular queue, visits contains nest identifiers and load information of the last N visited nests; this information is used to update the load information stored in Load Storages. Variables maxLoad and maxNest (minLoad, minNest) contain the load and the identifier of the nest with maximum (minimum) load among those recently visited.

Whenever an ant reaches a nest, its *run*() method is executed. The *AntView* parameter passed to *run*() is a proxy object used by ants to communicate with the nest. The first action of *run*() is to obtain references to the local Job Manager and Load Storage; then, variables maxLoad, minLoad, maxNest and minNest are initialized by the *initMaxMin*() method (not shown in the figure), simply by substituting, the load value and the identifier of the nest with maximum or minimum load. Finally, method *run* invokes methods *doSearchMax*() or *doSearchMin*(), depending on its current state.

The first step of method *doSearchMax*() is to decide whether to keep traveling through the network, searching for nodes with higher loads, or to switch to the SearchMin state. An ant will explore the network until the ratio between the maximum and the minimum load values stored in the ant state reaches a target value (represented by TargetRatio). Furthermore, each ant has a probability KeepSearchProb to keep searching even when the target ratio has been reached, providing a way for ants to continue their search for overloaded nests.

If the ant decides to keep searching, method *goNextNest*() is invoked. This method selects the next nest and moves there by invoking method *move*() on the *AntView*. The selection of the next nest depends on whether the ant decides to explore the network completely at random, or to direct itself towards a region of the network that is expected to be more overloaded. This decision is made by tossing a coin with probability ExplorationProb. If the decision is not to explore, the next nest is selected according to a normal distribution among the nests contained in the local Load Storage that are believed to be more overloaded. Before moving to the next nest, the ant updates the local Load Storage with its current content of visits, and then updates the visits variable with the load value and the identifier of its current nest.

Method *doSearchMin* is similar to *doSearchMax*; the only difference is when the ant decides to switch again to the SearchMax state, in which case the balancing operation (mediated by the involved Job Manager) is started and the variables are re-initialized.

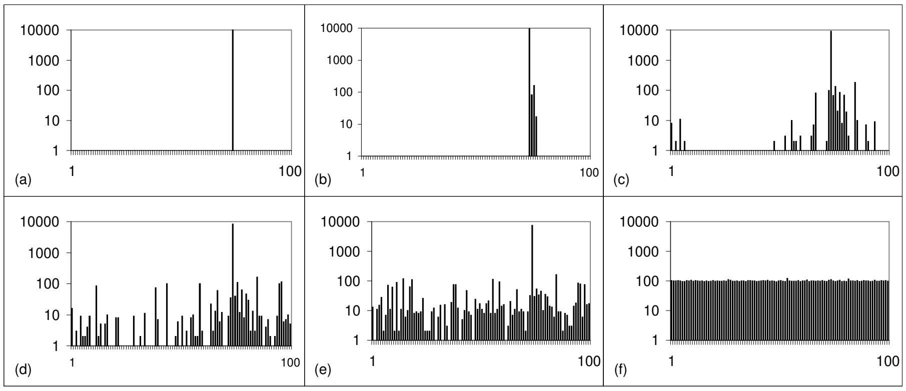

Figure 5. Load distribution after 0, 5, 10, 15, 20, 50 iterations.

4 Performance Evaluation

In this section, we present preliminary results for Messor, obtained through the Anthill simulator. Further details can be found in a companion paper [21]. Figure 5 illustrate how the load balancing process performed by Messor evolves over time. The results were obtained in a network of 100 idle nests, initially connected to form a ring (for visualization reasons). Initially, 10,000 jobs are generated in a single node. The different histograms depict the load observed in all the nests (x-axis) after 0, 5, 10, 15, 20, and 50 iterations of the algorithm. At each iteration, a set of 20 ants perform a single step, i.e. executes its run method and moves to the next nest. In each iteration, a node is limited to at most 200 job transfers to other nodes. As the figure illustrate, only 15-20 iterations are required to transfer jobs to all other nodes in the network, and after 50 iterations, the load is perfectly balanced. The first iterations are spent exploring the neighborhood in the ring network. After a few iterations, new connections are created and used to transfer jobs to remote parts of the network.

5 Discussion and Conclusions

We have argued that techniques borrowed from CAS could form the basis for a new paradigm for building P2P systems that are adaptive, resilient and self-organizing. To prove the viability of this idea, we have used Anthill to develope a P2P load-balancing algorithm that exhibit the above properties. Messor ants adapt their behavior to the load conditions, wandering about randomly when the load is uniformly balanced and moving rapidly towards regions of the network with highly unbalanced loads when these exist. The system is resilient to failures, as jobs assigned to crashed nodes are simply re-inserted in the network by the nest that generated them. And finally, Messor is self-organizing, as new nests may join the network, and their computing power is rapidly exploited to carry on the computation, as soon as ants discover the nest and start to assign it jobs transferred from other nests.

Our work may be compared with existing architectures for distributed computing. Several distributed computing projects [1, 8] are based on the master-slave paradigm, in which a well-known centralized master is responsible for supplying slave machines with jobs. This kind of approach is often called *hybrid P2P*, since although Seti@Home and Distributed.net is capable to take advantage of the huge amount of computing resources available in the Internet, its design is centralized. In Messor, every node of the network is capable of producing new jobs and introduce them in the network for computation.

Messor may be compared with so-called grid computing projects [10], such as Globus [9] and Legion [7]. The goals of Messor are more simplistic than those of these projects, that present complex architectures, capable to organize computations based on the memory, storage and computing requirements of jobs, as well as on the relationships between jobs. Nevertheless, Messor is interesting because, unlike these projects, presents a completely decentralized architecture.

The agent cloning approach [25], also facilitate load-balancing through the use of matchmaking agents to advertise capabilities of e.g., underloaded agents. This allows overloaded agents to find underloaded once to which they may clone (migrate) themselves. Using matchmaking agents is inappropriate for P2P and grid computing systems, since it impose a certain degree of centralization.

Many systems already exist for achieving dynamic load distribution, in particular process migration systems [17] like MOSIX, Sprite, Mach and LSF. However, none of these apply a CAS based approach to this problem, and many of them employ a centralized load-balancing algorithm, making them unsuitable for deployment in grid computing applications. The MOSIX [4] system use a decentralized and probabilistic load balancing algorithm similar to the one adopted by Messor. However, MOSIX requires a prior knowledge of the distributed sytstems, while Messor is a self-configuring P2P system. Moreover, MOSIX is a kernel level process migration system, and thus unsuitable in heterogenous environments.

Anthill share its goals with project DIET (Decentralised Information Ecosystem Technologies) [16, 6], aimed at the development of a robust, adaptable and scalable software platform for multi-agent systems applied to information processing.

We conclude by highlighting the fact that Messor is still a prototype. Many important features needed by distributed computing systems have not been implemented yet. For example, we have not considered issues related to security, apart from enclosing visiting ants in "sandboxes" that limit the set of actions performed by them. Mechanisms to authenticate users and to keep account over the number of jobs submitted and computed by nests are needed; these mechanisms may also prove useful as a defense against denial-of-service attacks. Although important, we believe that these lacks are hortogonal to the Messor algorithm presented in this paper, whose novelty resides in the way in which the load of the system is balanced. In this sense, the preliminary results obtained by the Messor simulations are interesting. Nevertheless, Further studies are needed to improve our understanding of the behavior of Messor ants. In particular, we are interested in obtaining an evaluation of the number of ants needed to manage a network. We plan to implement a mechanism to bound the number of ants present in the system simultaneously, by adding a module at each nest that kills ants when they are in excess, and creates new ants when the nest has not been visited recently. This module will also have an adaptive behavior, increasing the number of ants when the load is highly unbalanced.

References

[1] D. Anderson. SETI@home. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 5. O'Reilly, Mar. 2001.

[2] The Anthrax Project. http://www.chem.ox.uk/anthrax.

[3] Ö. Babaoğlu, H. Meling, and A. Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. In *Proc. of the 22th Int. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.

[4] A. Barak and A. Shiloh. A Distributed Load Balancing Policy for a Multicomputer. *Software Practice and Experience*, 15(9):901–913, Sept. 1985.

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[6] E. Bonsma and C. Hoile. A Distributed Implementation of the SWAN Peer-to-Peer Lookup System Using Mobile Agents. In *Proc. of the 1st Workshop on Agent and Peer-to-Peer Systems*, Bologna, Italy, July 2002.

[7] S. Chapin, J. Karpovich, and A. Grimshaw. The Legion Resource Management System. In *Proc. of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 1999.

[8] Distributed.net Home Page. http://www.distributed.net.

[9] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure,*. Morgan Kaufmann, 1999.

[11] P. Grasse. La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. *Insectes Sociaux*, 6:41–81, 1959.

[12] Project JXTA. http://www.jxta.org.

[13] G. Kan. Gnutella. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 8. O'Reilly, Mar. 2001.

[14] J. Kubiatowicz 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.

[15] A. Langley. Freenet. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 8. O'Reilly, Mar. 2001.

[16] P. Marrow, M. Koubarakis, et al. Agents in Decentralised Information Ecosystems: The DIET Approach. In M. Schroeder and K. Stathis, editors, *Proc. of the Symposium on Information Agentsfor Electronic Commerce (AISB'01)*, pages 109–117, University of York, United Kingdom, March 2001.

[17] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, Sept. 2000.

[18] 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.

[19] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Apr. 1998.

[20] A. Montresor, Ö. Babaoğlu, and H. Meling. Gnutant: Free-Text Searching in Peer-to-Peer Systems. Technical Report UBLCS-02-07, Dept. of Computer Science, University of Bologna, May 2002.

[21] A. Montresor, H. Meling, and Ö. Babaoğlu. Messor: Load-Balancing through a Swarm of Autonomous Agents. Technical Report UBLCS-02-08, Dept. of Computer Science, University of Bologna, May 2002. In preparation.

[22] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1994.

[23] 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.

[24] F. Russo. Design and Implementation of a Framework for Agent-based Peer-to-Peer Systems. Master's thesis, University of Bologna, July 2002.

[25] O. Shehory, K. Sycara, P. Chalasani, and S. Jha. Agent Cloning: An Approach to Agent Mobility and Resource Allocation. *IEEE Communications Magazine*, 36(7):58–67, July 1998.

[26] C. Shirky. Listening to Napster. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 2. O'Reilly, Mar. 2001.

[27] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.