Solenopsis: A Framework for the Development of Ant Algorithms

Amos Brocco
University of Fribourg
Fribourg, Switzerland
Email: amos.brocco@unifr.ch

Béat Hirsbrunner
University of Fribourg
Fribourg, Switzerland
Email: beat.hirsbrunner@unifr.ch

Michèle Courant
University of Fribourg
Fribourg, Switzerland
Email: michele.courant@unifr.ch

***Abstract*— Network resources management issues in complex and dynamic scenarios require decentralized solutions and adaptive systems to face critical and unattended situations. Bio-inspired techniques such as swarm intelligence algorithms, have proved to be robust and suitable for managing tasks like routing, load-balancing or resource discovery. In this paper we describe Solenopsis, a framework for the development, simulation and deployment of ant-algorithms, which is aimed at supporting network management middlewares. The system provides a modular and scalable environment that can be distributed over a network. Ants are coded using a simple programming language, and are able to migrate across nodes. Two basic load-balancing algorithms are presented and evaluated, as an example of how this tool works and can be used in practice.**

I. Introduction

Resource management in highly dynamic networks, mobile networks, P2P systems or pervasive computing environments, faces a number of challenges related to scalability and reliability of algorithms. Centralized or hierarchical management scales badly and is prone to creating bottlenecks. Managing dynamic heterogeneus environments requires flexible, robust and adaptive systems. Research has shown that these requirements may be fullfilled by self-organized multi-agent systems with emergent behaviors [1]–[3]. One of such mechanisms is ant colony optimization (ACO).

ACO [4] is a metaheuristic problem solving technique inspired by ant social behaviors. One of the most exploited concerns the problem of searching for food and then inform other individuals in the colony about the path followed to reach it. Basically, when ants leave their nest to look for food they wander almost randomly. As a food source is found, the ants travels back to nest, and leave a pheromone trail that can be subsequently used by other ants to reach the food. The shortest the route to food, the more frequently the trail will be reinforced, thus allowing optimal solutions to shortest-path problems or to the travelling salesman problem (TSP) to be found. Other kinds of optimization problems that can be solved by mean of ant-algorithms can be found in [5].

Another interesting behavior is the one shown by ants of the *Messor Sancta* species, where members of the colony are able to clean-up their nests by piling up dead bodies. Messor [6] implements this algorithm to perform load-balancing in a P2P network.

Despite the little solving capability of individual ants, colonies exhibit some kind of emergent behavior that is able to work out almost optimal solutions to difficult optimization problems such as finding the shortest path between two locations, balancing traffic on a track, or organizing defense against attacks from stranger colonies.

In computer science, ants are simulated by mean of autonomous agents that move across a graph (being, for example, a communication network). In [7] a distinction between intelligent and unintelligent artificial ants is made. Intelligent (or *agent-based*) species have limited individual capabilities and can interact with the environment and with other ants; communication is typically done by leaving an artificial pheromone trail (*stigmergy*). On the other hand, unintelligent ants are used solely as information conveyors, and are employed in complex adaptive systems (CAS), where some degree of fault-tolerance and adaptability to environmental changes is required. Like all other swarm intelligent systems, ACO systems are completely decentralized and self-organized: complex tasks are carried out by mean of simple local interactions between ants and the environment.

Beside the field of optimization problems, ant-based and biologically inspired systems have also found their way in complex systems management, in situations where centralized approaches are difficult or impossible to implement, or just too unreliable. In such cases, emergent behavior coupled with self-organizing capabilities provide the increased robustness and scalability typical of distributed systems along with the ability to react and adapt to situations.

In this paper we will introduce Solenopsis [1], a framework aimed at simplifying the development and simulation of ant algorithms for network management purposes. This application is part of an ongoing project called SmartGRID [8], which aims at providing a self-organizing intelligent network management middleware; nonetheless Solenopsis is developed as a standalone application that can be used in different scenarios. Section 2 presents related work in the domain of agent-based and bio-inspired network resource management middlewares. Section 3 provides the specifications of the framework, and an overview of its model with a focus on the main components.

[1] *Solenopsis Invicta*, also known as *Red imported fire ant*, is a particularly aggressive species of ant originary from South America.

Finally, section 4 describes the implementation and evaluation of two simple load balancing algorithm on our platform, as an illustration of its usage.

II. Related work

Our research on resource management in using swarm intelligence algorithms is currently focused on three domains of application: routing, network load-balancing, and resource discovery. The routing problem, which can be viewed as an instance of the TSP, has been widely investigated, thus different solutions relying on ACO exists; examples can be found in [9] and [10]. The load-balancing problem concerns optimal resource allocation in computing networks, and is mainly solved by using variations of the Messor algorithm [6], [11]. For resource discovery using ant algorithms, an approach applied to P2P networks is presented in [12].

Because many algorithms rely on empirically found parameters, simulations and the capability of fine-tuning specific values play an important role in the research and evaluation of ant-based systems. To support development of ant algorithms some specific software platforms exist: examples are Swarm [13], MASS [14] and Anthill [7].

The Swarm Simulation System allows to model multi-agent discrete simulations at different levels. The framework is object-oriented, with agents being objects. Agents can interact with each others, and the whole simulation can be synchronized. The platform itself offers different tools for algorithm profiling and data analysis.

The Multi-agent System Simulation Framework (MASS) allows accurate and controllable simulations. To ensure that results are reproducible, two simulation techniques are available: discrete time simulation and event-based simulation. Agents can sense the environment and perform a mixture of real and simulated activities.

Anthill is a Java framework that supports P2P application development. It provides runtime and simulation environments and it has been successfully used to implement the Messor load-balancing algorithm. The runtime environment is a middleware built on JXTA [15] that allows real-world deployment of applications, whereas the simulation environment allows local testing and evaluation of ant algorithms. Unfortunately the development of Anthill was stopped in year 2002.

The first two platforms aim at evaluating multi-agent coordination in distributed systems with total accuracy, by mean of a simulation environment; therefore their architecture is not well suited for real-world distributed dynamic environments. In contrast, the Anthill one is not only aimed at supporting the design and analysis of P2P systems, but at the implementation of such systems in real network environments as well.

To such an extent the Anthill framework is the most related to the work described in this paper, notwithstanding that some of the goals and the requirements are different, namely because our research is not focused on P2P networks optimization but on the study of ant-algorithms for general network management tasks.

Finally, some network middleware systems that exploit bio-inspired algorithms already exist: examples can be found in EcoMobile [16] and ARMS [17].

The EcoMobile framework has been developed in our research group, in order to provide active management for optical networks. EcoMobile approaches the problem of network resource management by using an ecosystem of mobile agents and task-objects (software entities executing a computational task). Agents perform navigational and coordination activities, whereas task-objects themselves exhibit the operational behavior. The way agents and task-objects collaborate for the management of the system is bio-inspired: while the former wander around, the latter may attach to them to be carried away.

The ARMS platform is an agent-based management middleware that implements the Messor algorithm to balance load between nodes [18]. Self-organization is achieved by mean of unintelligent ants that dispatch information between agents; thus ants just act as information conveyors.

III. Solenopsis Framework

In this section the Solenopsis framework is discussed. First, the requirements and design goals are presented. Then an overview of the framework is provided, followed by an in depth discussion of its components.

A. Design goals

Solenopis has been developed with the goal of being easy to deploy and extend. The main idea that drives this project is to provide a customizable software to simplify the development and the analysis of ant-algorithms. The required features can be divided in two groups: low-level and high-level requirements.

Low-level requirements focus on the internal details of the framework: a clean and simple object-oriented design, a distributed and multi-platform architecture, scalability and modularity.

On the other hand, high-level needs concentrate on the bare features that should be made available to the programmer when developing ant-algorithms: capability of transparently migrating ants from one node of the distributed network to another, communication facilities between ants and access to external resources.

B. Solenopsis model

Figure 1 gives an overview of the framework. Solenopsis software is distributed over a network of nodes. On each node a daemon manages one or more virtual machines, each one executing an ant-algorithm. As the virtual machine is kept simple by design, it offers only basic types management, delegating all high-level requirements to dynamic service libraries installed on the node. Such services include migration, communication, environment perception and logging facilities. Ants can access these services by mean of function calls. Virtual machines are created and destroyed dynamically: as soon as an ant tries to migrate to a node, a new virtual

machine is allocated and bound with the available services. For simulation and analysis purposes many daemons can be run on the same node, in order to ease collection of results. Unlike other frameworks, Solenopsis does not impose a specific communication layer, and can rely on custom developed network services.

The current implementations of the compiler and the virtual machine are made in Java.

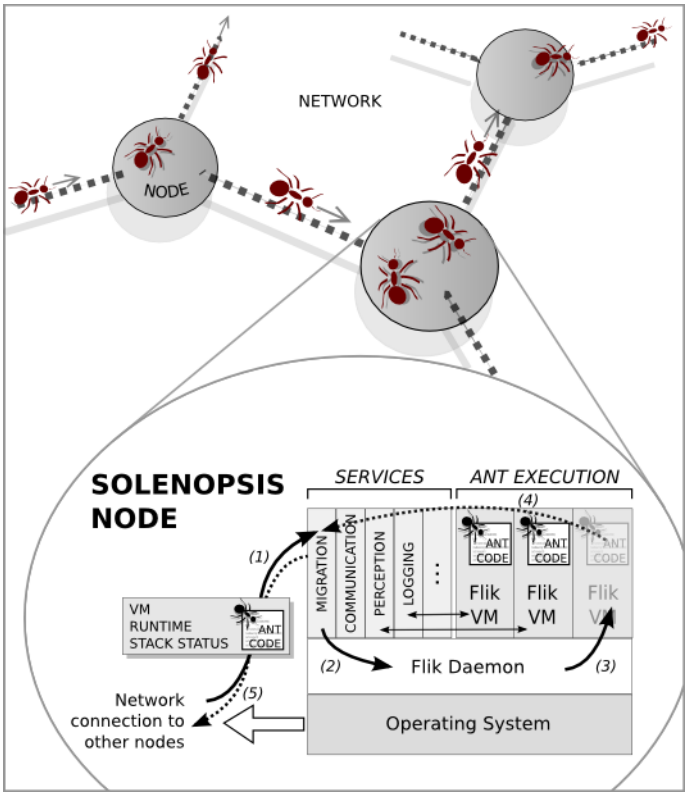

Fig. 1. Solenopsis Model and migration steps: (1) Incoming migration demand (2) Request for a new virtual-machine (3) Runtime status reestablished, execution resumed (4) Call to Migration service (5) Serialization of ant state and outgoing migration

C. Ants

Ants can be developed in a Lisp-like language then compiled to byte-code. The latter is then encoded to a platform independent Unicode string, which can be interpreted by the virtual-machine. The ant programming language uses the same syntax as Lisp, but also adds some constructs typical of imperative languages, such as **set!** (to change the value of a variable), **begin** (to define procedural blocks) and **while/for** loops. Examples of code can be found below.

Ants are typically generated by services running on a node, either in response to changes in environmental conditions or to user's requests. By mean of services available on a node, ants may have the ability to move to other nodes, to collect information about the network, to communicate with other ants (either directly or through stigmergy), or to perform specific actions such as updating routing tables.

The algorithm describing the behavior of the ant as well as its runtime status are encapsulated in the ant itself. This allows loading, executing and migrating different ant species in the system. Additionally, as the evaluation and the deployment environment are the same, there is no need to re-implement algorithms.

As the byte-code is interpreted by a VM, ants are sandboxed and the whole execution environment is protected; overall security is increased because only services made available by the node can be accessed.

D. Virtual Machine

The VM bytecode is stack-based and fully interpreted by the virtual-machine. There is support for some basic types such as *number* (integer and float), *string*, *list*, *dictionary*, *lambda*, and *nil* (the only type whose semantic value is the boolean *False*); functions to manipulate these types are available as built-in. Additionally a *blob* type is present, and typically allows exchanging data between services without using basic types. Currently 19 stack operators are available:

- LOAD: Loads a variable on the stack.
- STORE: Stores the value on the top of the stack in a variable.
- SET: Assigns the value on the top of the stack to a variable.
- CONST: Loads a constant value (number or string) on the stack.
- JUMP: Jumps to a relative offset in the byte-code.
- FJUMP: Jumps if the top of the stack is *nil*.
- TJUMP: Jumps if the top of the stack is not *nil*.
- IFCALL: Calls an internal function.
- EFCALL: Calls an external function (service).
- IPCALL: Calls an internal procedure.
- EPCALL: Calls an external procedure (service).
- BIND: Binds the n-topmost values of the stack with the given variables.
- RETURN: Returns from an internal function or procedure call.
- DROP. Drops a stack frame (used when returning from an internal call).
- END: Terminates the execution.
- LAMBDA: Defines a new lambda object.
- LFCALL: Calls a lambda function.
- LPCALL: Calls a lambda procedure.
- SYNC: Synchronizes with other ants.

The VM executes only single-threaded code, and is neither aimed at supporting object-oriented programming nor purely functional programming (despite the programming language being inspired by Lisp). The type-system is dynamic: variable type is completely determined at execution. As there is no concept of pointers or references to data, variables are always passed by-value. Finally, there is full support for recursive calls and local variables.

A drawback of keeping the VM small is that built-in functions are too simplistic and barely sufficient for real ant-algorithms development. To overcome this limitation, the

execution of additional functions is delegated to external services bound to the VM: in the current implementation each service consists of one or more Java classes that are dynamically loaded by the daemon. Each class provides a function that can be invoked from the ant's code by mean of an identifier: the daemon is responsible for mapping function identifiers with the corresponding object. An advantage of such a design is the possibility to extend and customize the framework without modifying the core components. Currently, plugging-in or removing services is simply a matter of copying or deleting the service classes from a directory.

Services can implement side-effects such as moving the ant to another node or changing some values in a local database: to provide such functionality, each service has full access to the virtual machine and its execution state. Additionally the virtual machine allows deserializing and serializing of its whole state from and to an Unicode string representation, which is essential for allowing strong migration of ants. The aim of serializing to a string instead of binary data is improving portability by allowing re-implementation of the virtual machine in other languages.

E. Services

Services available on a node are essentials for the development of ant-algorithms, and deserve a more detailed description. Services can be called in the ant code as if they where internal functions, but at byte-code level they need to be referenced explicitly. In the following subsections the main services will be discussed, with a focus on the two most important ones: migration and perception.

1) Migration: The migration service allows moving a running ant from one node to another, thus performing a strong migration. This service exports two functions: **migration::migrate** and **migration::fork**. The first one sends an ant to a given target node, the latter forks the ant so that a new ant is executed on a remote node whereas the caller ant continues its execution on the local node. An example of these services is given in Figure 2.

```
(if (migration::migrate target)
   <code executed on the target node>
 else
   <code executed if migration failed>)

(if (migration::fork target)
   <code executed on the target node>
 else
   <code executed on the local node>)
```

Fig. 2. Migration and forking

The migration service is also responsible for managing incoming migration requests, instantiate a new VM object and use it to restore ant execution. The migration process (incoming and outgoing) is depicted in Figure 1 (1)-(5):

(1) An incoming requests is received by a server run by the local migration service. This server is executed upon initialization of the service, and listens on a socket for incoming migration requests.
(2) The migration service asks the local daemon for a new VM object.
(3) The VM state is restored and the virtual-machine is set running.
(4) As the ant calls the migrate or fork function, the VM state is serialized by the migration service.
(5) Ant execution state is transmitted to the target node.

The migration and forking functions return a value representing the node where the execution is resumed: the ant knows if the migration succeeded by checking that value.

2) Perception: The perception service allows an ant to interact with the environment, by reading and writing *(key,value)* tuples. Typically it is possible to get information about the current node, such as its network address or its neighbors. Tuples are identified by their *key*, represented by a string, but associated values can be any VM recognized type. Four functions are available for this service: **perception::getEnvProperty** (to read an environment property), **perception::hasEnvProperty** (to check if a given property is defined in the environment), **perception::setEnvProperty** (to define or modify an environment property) and **perception::removeEnvProperty** (to remove a property). All information must be persistent and accessible from outside the node, and is typically stored in a database local to each node. Figure 3 shows how to read the current node identifier.

```
(perception::getEnvProperty "node.id")

(perception::setEnvProperty "ping" 143)
```

Fig. 3. Reading and writing an environment property

3) Communication: Ants communicate in an indirect way, by mean of a blackboard. Ants can store and retrieve values stored in the local node by mean of a key, which is matched using regular expressions. Information published on the blackboard is non persistent, thus it is lost in the event of a node crash.

4) Logging: The logging service allows ants to log events that can be retrieved by the user. Beside its debugging purpose, logging can also be used to track ants location in the network.

5) Other services: Additional services consisting of synchronization, mathematical or random functions are also available. Synchronization routines allow discrete-time simulations and a deterministic behavior from one simulation to the next. Synchronization methods are available both for local simulations and in a distributed environment (using a central server). By mean of random functions it is also possible to introduce non-deterministic behaviors which are typically required when implementing random wandering of ants in the network. Finally, mathematical functions can be used for statistical analysis, or for other kind of intensive calculations that would be too slow or just impractical to implement directly in ant code.

IV. Case Study

As an example of usage of this framework, this section presents the implementation and the simulation of load-balancing algorithms inspired by Messor [6]. In the first part, some background information about the Messor ant algorithm is given. In the second and third parts, details of the implementation of two algorithms are discussed. In the fourth and fifth parts, some information about the simulation environment is given. Finally simulations results and a performance evaluation is provided.

A. Messor ant algorithm

The Messor load-balancing algorithm, in all its variations, is inspired by the behavior of ants of the *Messor Sancta* species. It has been observed that ants group objects in their surroundings in order to clean-up their nest. This behavior was simulated successfully with artificial ants, and has been resumed to the following three rules [19]:

(i) Wander around randomly, until finding an object.
(ii) If an object is being carried, drop it and continue wandering.
(iii) If nothing is being carried, pick up the object and continue wandering.

To solve the load-balancing problem, these rules are actually reversed. Objects are the jobs assigned to a node, and the goal is to move jobs around between nodes to equalize the load. Instead of piling objects together, the objective of the ant is to scatter them around:

(i) Wander around randomly, until finding an overloaded node.
(ii) When an overloaded node is encountered, pick it up its jobs and continue wandering.
(iii) When no more overloaded nodes are encountered, drop the carried jobs.

B. First algorithm

The first algorithm is a straightforward implementation of the Messor rules described above; the actual code is summarized in Figure 4 (for space reasons some functions have been omitted).

Ant life is divided in two phases: an exploration phase and a balancing phase. In the first one, the ant wanders randomly in the network, collecting information about nodes load. In the second phase the ant performs a load balancing between the most and least loaded nodes found.

When an ant is created, global variables are instanced and the `__body__` procedure is invoked. The body of an ant is typically coded as an infinite loop that repeats the algorithm until a termination function is called. This algorithms requires six global variables to be defined: `MAXSTEPS` is an integer defining the maximum number of wandering steps allowed before switching the state. To store the current state, the `currentState` integer is used: 0 for *SearchMin*, 1 for *SearchMax*. Variables `maxLoadId` and `minLoadId` are used to store the identifiers of the nodes with maximum, respectively minimun, load found; `maxLoad` and `minLoad` record their values. These last four variables are initialized to the values found on the node where the ant is created.

```
(define MAXSTEPS n)
(define currentState 0)
(define maxLoadId (perception::getEnvProperty
                   "node.id"))
(define minLoadId maxLoadId)
(define maxLoad (perception::getEnvProperty
                 "node.load"))
(define minLoad maxLoad)

(define (__body__)
  (let ((steps 0)
        (target nil))
    (while 1 (begin
      (set! steps (+ steps 1))
      (updateLoad)
      (if (>= steps MAXSTEPS) (switchState))
      (set! target (getRandomElement
            (perception::getEnvProperty
            "node.neighbors")))
      (migration::migrate target)))))
```

Fig. 4. Body of the first algorithm

Beside global variables, some local variables are also defined in the body: `steps` (to count the actual wandering steps) and `target` (used to store the destination of a migration). As the algorithm begins, the ant is in the *SearchMin* state. At each loop the number of steps is incremented: when the `MAXSTEPS` value is reached the state is switched.

Depending on the current state, the `updateLoad` function is used to update the maximum or minimum load found, along with the associated node identifier. In other words, the goal of the *SearchMin* state is to find the least loaded node, whereas the *SearchMax* state looks for the most loaded node.

The algorithm determines the node where to migrate by randomly selecting a node from the neighbors list. A migration then takes place, and the following loop is actually executed on the target node.

Upon switching from the *SearchMin* to the *SearchMax* state, the ant resets the step count and continues wandering for `MAXSTEPS` steps. When switching back to the *SearchMin* state, a balancing is made between the node with the maximum load found, and the one with the minimum load. The ant then dies.

Two major constraints are set on this first algorithm. The first is the mandatory number of steps to perform in each state: this value must be defined beforehand, and determines the length of the exploration phase of the ant. The second is the limited life of the ant: as soon as the balancing is done, the ant is killed by the system.

C. Second algorithm

The second algorithm introduces two modifications to the first one: forking and anticipated balancing.

The idea behind forking is to generate additional ants in extreme load situations. Each ant keeps a mean of the loads encountered in its wandering: when this value exceeds

a certain threshold, the ant performs a fork instead of a migration. A fork results in two ants being alive: one on the current node and an exact copy on the target node. To avoid generation of too much ants by consecutive forks, after each fork the load mean on both ants is simply halved[2]. To implement this behavior, the migration call in the body is replaced by a call to the `move` function (shown in Figure 5). The `loadMean` variable is used to choose between migration or forking.

```
(define (move target)
    (if (> loadMean LOADMEANTHRESHOLD) (begin
        (set! loadMean (/ loadMean 2))
        (migration::fork target)
    ) else (begin
        (migration::migrate target)
        )))
```

Fig. 5. Fork

The second improvement, shown in Figure 6, is anticipated balancing. Small changes in the `__body__` function allows an ant to perform balancing of two nodes aforetime if some conditions are met. Balancing two nodes beforehand is permitted when the mean loads of the nodes traversed and the load on the current node are below a given threshold.

```
...
(updateLoad)
(if (and (= currentState 0)
 (< loadMean 30)
 (< (perception::getEnvProperty "node.load") 30))
 (begin
   (balancing::balance maxLoadId
     (perception::getEnvProperty "node.id"))
   (migration::kill)))
(if (>= steps MAXSTEPS) (switchState))
...
```

Fig. 6. Anticipated balancing

The load on the current node is balanced with the most loaded node found in the past cycles; balancing is followed by a call to `migration::kill`, which results in the death of the ant. This algorithm shifts some control over the population from the system to the ant: the colony itself can expand or shrink without supervision.

D. Hardware and software platform

Algorithms have been tested on a dual AMD Opteron Processor 252 server, equipped with 4GB memory. Simulations of multiple nodes have been done locally to make use of the synchronization service without incurring too much network overhead. For storage and logging purposes, an additional MySQL database server with similar hardware characteristics has been used. The operating system of choice has been GNU/Linux Debian Sarge 3.1r1, with kernel 2.6.8, compiled for AMD64 with symmetric multiprocessing support. The Java runtime environment is Java2 SE 1.5 from Sun Microsystems.

[2]Another way to avoid explosion in ant population is to act upon the forking threshold.

E. Memory footprint

The approximate static memory footprint is 1 MByte for the bare simulation environment, 10 KBytes for each virtual node/daemon, 75 KBytes for each new virtual machine created on a virtual node, and 75 KBytes for basic services (which can be shared across local daemons during simulations). Dynamic memory usage depends on the number of ants and their code size (an ant being typically 3 KBytes for the considered algorithms).

F. Simulation methodology

To evaluate both algorithms, a two-dimensional torus network topology has been chosen.

In a first run, the network is composed of 400 nodes, with random loads uniformly distributed between 0 and 100 jobs. For a second run a network of 100 nodes is used; in this case, a single node is initially loaded with 2500 jobs.

In both experiments, a node is considered as overloaded when it carries more than 70 jobs; the node load is checked every 25 iterations, and on each overloaded node a new ant is generated. The simulation has been conducted in discrete time using a global synchronization between all nodes: each iteration corresponds to a cycle of the algorithm body followed by a migration. Additional parameters used for these simulations are:

- `MAXSTEPS` set to 10.
- Forking if load mean is greater than or equal to 80 jobs per node.
- Anticipatory balancing if the load mean and the actual load are both lesser than 30 jobs.

As the upper limit of the number of steps performed by the ant in both algorithms is 20, regenerating a new population every 25 iterations (thus leaving a death phase between generations) was done for visualization reasons.

The goal of both algorithms in both runs is to remove the overload by attaining a load of 70 jobs on each node.

G. Performance evaluation

Figures 7 to 10 show the results for the two runs; depicted are the evolution of the global overload (grey area), the number of overloaded nodes (black line), the number of balancings (dotted histogram), and that of migrations (thin grey line) with respect to iterations (x-axis). The size of the population is not depicted because it corresponds to the number of migrations, as at each iteration one ant performs exactly one migration.

In the first run, the first algorithm (Figure 7) spends the first 20 iterations exploring the network, finally balancing the loads. A new generation of ants is generated by the system only at the 25th iteration, and the global overload is canceled after 98 iterations. Also, the number of migrations equals the number of nodes currently overloaded.

The second algorithm (Figure 8) quickly decreases the overall overload by exploiting its anticipated balancing capability.

The fact that network load is randomized at the beginning, provides good situations for anticipated balancing. Forking increases the ant colony size in regions of higher load; newer ants are able to better balance these regions without waiting for a new colony to be generated by the system. It is interesting to note that the number of migrations no longer corresponds to the number of overloaded nodes.

Comparing the two algorithms, the second one clearly performs better, and roughly takes 30 less iterations than the first one to completely eliminate the overload.

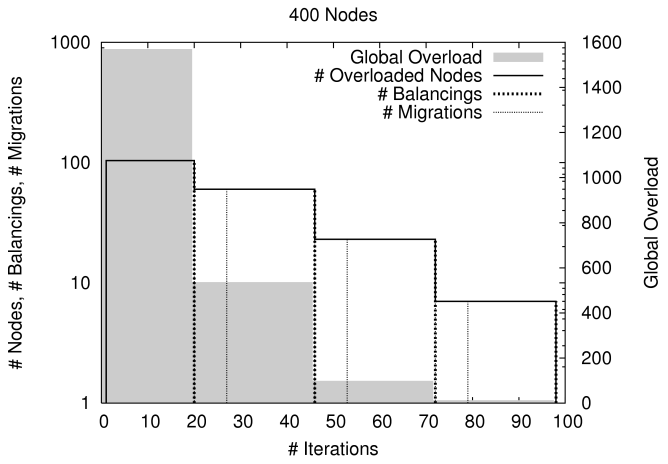

Fig. 7. First algorithm, first run (random loads)

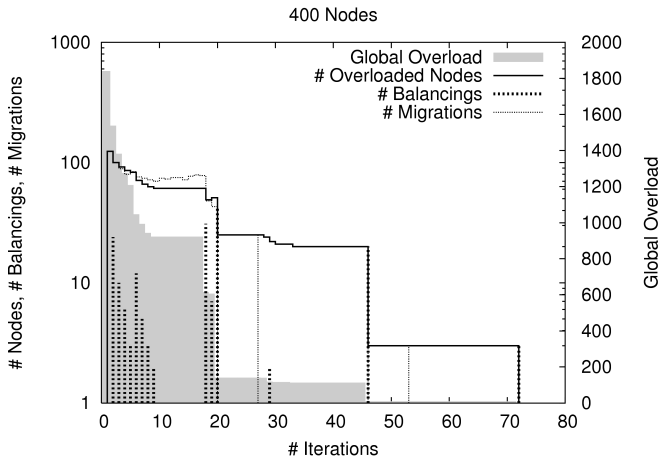

Fig. 8. Second algorithm (fork, anticipated balancing), first run (random loads)

In the second run, the first algorithm (Figure 9) struggles to get a sufficient population of ants being able to minimize the global overload. Given that free nodes are available near the loaded one, ants are forced into exploring the network until the 20th iteration. The system must then wait 5 iterations until a new generation is created.

The second algorithm (Figure 10) is able to generate many ants in the first iterations, with forking taking place in the neighborhood of the only loaded node.

For both algorithms, as soon as ants reach new parts of the network, free nodes are found and the load is then quickly balanced. Nevertheless, the second algorithm still has an edge over the first one, and manages to eliminate the overload with half iterations.

As we were expecting, Solenopsis behaves such that in both runs the second algorithm is able to balance the network with fewer iterations.

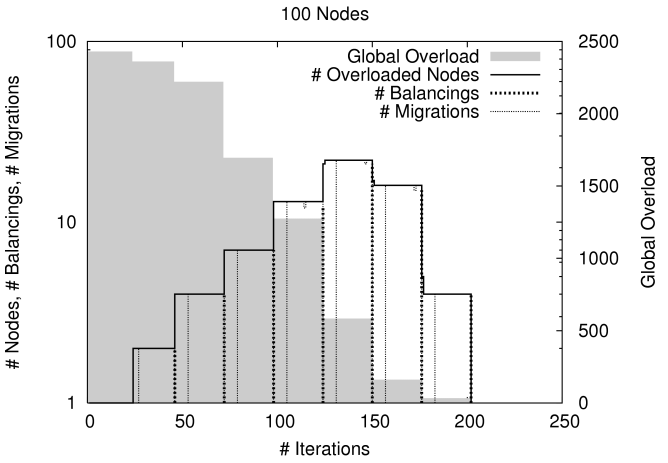

Fig. 9. First algorithm, second run (single node initially loaded)

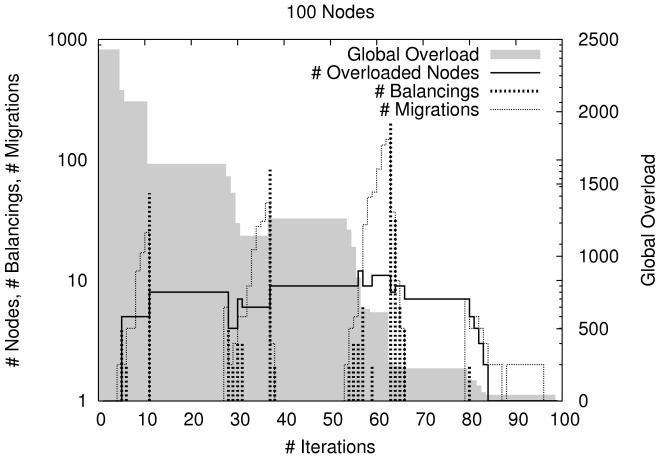

Fig. 10. Second algorithm (fork, anticipated balancing), second run (single node initially loaded)

Despite showing an improvement in the performance of both runs, forking also has a major drawback that could limit its implementation outside simulations: in high load situations, the size of the population grows almost exponentially. As the number of migrations is proportional to the number of ants, network congestion problems could arise. Increasing the threshold required for forking or upperbounding the number of forks that an ant can perform during its life cycle could probably limit this effect. On the other hand, in the first algorithm the number of ants is proportional to the number

of overloaded nodes; that way, the colony size is completely foreseeable knowing the status of the system.

V. Conclusion

In this paper we presented the Solenopsis framework for the development of ant algorithms aimed at supporting network management middlewares. This software meets the objectives of a modular and scalable framework, and fullfills the requirements for both simulation and deployment scenarios.

The model is simple by design and allows further extensions or external components to be easily integrated. Ant development is done with a Lisp-like language, which allows rapid prototyping of algorithms.

Simulation of algorithms can be performed on a local synchronized environment, whereas deployment using distributed nodes allows unsynchronized testing the system with real-world conditions.

As an example, two simple load-balancing algorithms implementations were detailed, and simulations results validate the use of this framework for its purposes. A next step will aim at developing and test more robust network management techniques, such as routing or resource discovery algorithms. Evaluation of such algorithms will then be provided in the context of large scale deployments in real world scenarios.

Next improvements will also include the creation of a development environment for ant algorithms to allow a yet more simplified simulation and evaluation, as well as additional service libraries.

To a certain extent, this work joins research done on the Anthill framework [7], although the goal of the Solenopsis framework is to support generic network management and not the development of P2P systems. In the context of the SmartGRID project [8], Solenopsis forms a low-level signaling layer meant to collect and dispatch information about the network status. On top of that, to build the SmartGRID network management middleware, an agent-based layer will use this information to coordinate and manage high-level tasks, such as scheduling or resource discovery in a computing Grid.

The complete source code of Solenopsis, which includes the bytecode compiler, the virtual machine, documentation and some example services, can be freely downloaded from [21].

Acknowledgment

Solenopsis has been developed thanks to the financial support of the Swiss Hasler Foundation [20] in the framework of the “ManCom Initiative” (ManCom for “Managing Complexity of Information and Communication Systems”), project Nr. 2122.

References

[1] G. Di M. Serugendo, M.-P. G. Irit, and A. Karageorgos, *Self-Organisation and Emergence in MAS: An Overview*. Informatica, Volume 30, Number 1, 2006

[2] Alberto Montresor, Hein Meling, Özalp Babaoglu, *Toward Self-Organizing, Self-Repairing and Resilient Distributed Systems*. Lecture notes in computer science, Springer, Berlin, Germany, 2003

[3] D. N. Legge, and P. R. Baxendale, *An Agent-Managed Ant-Based Network Control System*. Third Symposium on Adaptive Agents and Multiagent Systems (AAMAS-III) , AISB’03 Convention, University of Wales, Aberystwyth, 10rd - 11th April 2003.

[4] Marco Dorigo, Thomas Stützle, *Ant colony optimization*. The MIT Press, 2004.

[5] M. Dorigo, G. Di Caro, and L.M. Gambardella, *Ant algorithms for discrete optimization*. Artificial Life 5 137-172, 1999

[6] Alberto Montresor, Hein Meling, and Özalp Babaoglu, *Messor: Load-Balancing through a Swarm of Autonomous Agents*. AP2PC 2002: 125-137, 2002

[7] Özalp Babaoglu, Hein Meling and Alberto Montresor, *Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems*. Proceedings of the 22nd International Conference on Distributed Computing Systems, Vienna, Austria, 2002

[8] Béat Hirsbrunner, Michèle Courant, Amos Brocco, and Pierre Kuonen, *SmartGRID: Swarm Agent-Based Dynamic Scheduling for Robust, Reliable, and Reactive Grid Computing*. Working Paper 06-13, Department of Informatics, University of Fribourg, Switzerland, October 2006

[9] G. Di Caro, *Ant Colony Optimization and its application to adaptive routing in telecommunication networks* PhD thesis, Faculté des Sciences Appliquées, Université Libre de Bruxelles, Brussels, Belgium, 2004

[10] G. Di Caro, F. Ducatelle, and L.M. Gambardella, *Swarm intelligence for routing in mobile ad hoc networks*, Proceedings of Swarm Intelligence Symposium (SIS 2005), IEEE, 2005

[11] M.A. Salehi, and H. Deldari, *Grid Load Balancing using an Echo System of Intelligent Ants*, Proceeding (517) Parallel and Distributed Computing and Networks - 2006, Innsbruck, Austria, 2006

[12] Dan Wang, *A resource discovery model based on multi-agent technology in P2P system*, Proceedings of Intelligent Agent Technology (IAT 2004), IEEE, 2004

[13] N. Minar, R. Burkhart, C. Langton, and M. Askenazi, *The Swarm simulation system: a toolkit for building multi-agent simulations*, Working Paper 96-06-042, Santa Fe Institute, Santa Fe, 1996

[14] Bryan Horling and Victor Lesser and Regis Vincent, *Multi-Agent System Simulation Framework*, 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation, August 2000, EPFL, Lausanne, Switzerland, http://mas.cs.umass.edu/paper/186

[15] The JXTA Project Home Page, http://www.jxta.org

[16] D. Rossier-Ramuz, *Towards Active Network Management with Ecomobile, an Ecosystem-inspired Mobile Agent Middleware*, Department of Informatics, University of Fribourg, PhD thesis no.1392, October 2002.

[17] Junwei Cao, Stephen A. Jarvis, and Subhash Saini, *ARMS: An agent-based resource management system for grid computing*, Scientific Programming 10(2): 135-148, 2002

[18] Junwei Cao, *Self-Organizing Agents for Grid Load Balancing*, GRID 2004: 388-395, Pittsburgh, 2004

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

[20] Swiss Hasler Foundation Website, http://www.haslerstiftung.ch

[21] Solenopsis Website, http://diuf.unifr.ch/pai/solenopsis