Self-Organization Algorithms for Autonomic Systems in the SelfLet Approach

Davide Devescovi, Elisabetta Di Nitto, Daniel Dubois, Raffaela Mirandola
Dipartimento di Elettronica e Informazione
Politecnico di Milano
via Ponzio 34/5, 20133 Milano, ITALY
(devescovi,dinitto,dubois,mirandola)@elet.polimi.it

ABSTRACT

The difficulties in dealing with increasingly complex information systems that operate in dynamic operational environments ask for self-management policies able to deal intelligently and autonomously with problems and tasks. Biology has been a key source of inspiration in the definition of self-management approaches in the area of computing systems. In this paper we show how some biologically inspired self-organization algorithms have been incorporated into a framework that supports development of autonomic components called *SelfLet*s. The features of a *SelfLet* include the ability to dynamically change and adapt its internal behaviour according to modifications in the environment, to interact with other *SelfLet*s, in order to provide high-level services, and to make use of autonomic reasoning in order to enable self-* capabilities. In this context, self-organization features represent one of the *SelfLet*s autonomic abilities, and allow them to create groups of *SelfLet*s individuals able to cooperate between each other. The work is complemented with a performance study whose goal is to give insights about strengths and weaknesses of these algorithms.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Design; J.m [**Computer Applications**]: Miscellaneous

Keywords

Autonomic Computing, distributed and adaptable systems, clustering algorithms, performance analysis

1. INTRODUCTION

The today increasingly complex information systems operating in dynamic operational environment ask for management policies able to deal intelligently and autonomously with problems and tasks. To find the solution to this problem IBM has identified a new research discipline called *Autonomic Computing* [20] in which the maintenance and the decisions on a generic system occurs automatically without any human interventions.

A different line of research takes inspiration from natural adaptive systems, like bacterial colonies or insect colonies, and their intrinsic capabilities to organize global activities into highly adaptive functional patterns. These patterns emerge autonomously from simple local activity rules and local inter-component interactions. Several of these self-organization natural phenomena find a natural mapping to functional problems in modern and distributed information systems and therefore their philosophy can be adopted to build self-managing software systems.

In this paper we show how some biologically inspired self-organization algorithms [27, 29] have been incorporated into a framework that supports development of autonomic components called *SelfLet*s. A first definition of the *SelfLet* model has been presented in [11]. The features of a *SelfLet* include the ability to dynamically change and adapt its internal behaviour according to modifications in the environment, to interact with other *SelfLet*s, in order to provide high-level services, and to make use of autonomic reasoning in order to enable self-* capabilities. Autonomic applications and systems can thus be developed by deploying a number of *SelfLets*, customized and configured at a high-level according to the task they need to perform, trusting their self-managing abilities to take care of the more complex, low-level details. In this context, self-organization features represent one of the *SelfLet*s autonomic abilities, and allow them to create groups of *SelfLet*s individuals able to cooperate between each other. Input and inspiration of this work come from the *CASCADAS* European project (*Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services*) [14].

The organization of the paper is as follows. In Section 2 we present the self-organization algorithms we have adopted and highlight their possible uses. In Section 3 we present the model for the *SelfLet*, outlining the internal architecture of one of such elements and the relationships it can have with other *SelfLets*. Section 4 discusses on how we have incorporated the self-aggregation algorithms in our framework. In Section 5 we present some performance results that show whether a self-organization algorithm is good or not in a given situation. The purpose of this is to provide at the end of the simulations enough data to set up an exhaustive statistical analysis. The results of this analysis can be used to

design a *SelfLet* that can self-tune itself and behave always in the optimal way. Section 6 presents an overview of the state of the art in the fields of both Autonomic Computing and self-organization. Finally, Section 7 concludes the paper.

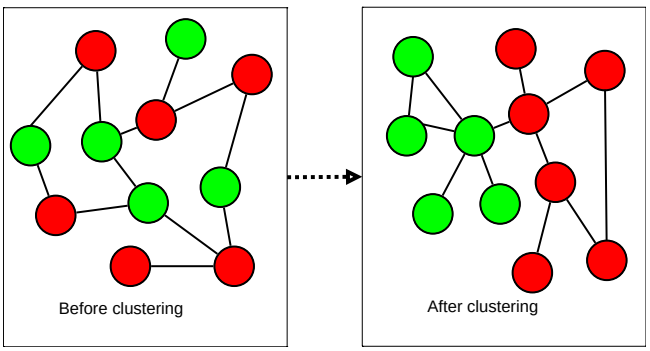
2. SELF-ORGANIZATION ALGORITHMS

In general, a self-organization algorithm is defined as an algorithm capable of making a spontaneous formation of well-organized structures, patterns and behaviors without central control [28].

In the following we consider two similar variants of self-organization algorithms, i.e., *clustering* and *reverse clustering* algorithms [27, 29] that have been defined within the CASCADAS project. The purpose of clustering is the creation of interconnected groups of nodes having the same type (see figure 1). Such groups can be useful, for instance, to achieve load balancing. In fact, the members of a group, when overloaded, can exploit the knowledge of the group and try to delegate their tasks to other members of the same group.

Vice versa, the purpose of reverse clustering is to constitute groups of nodes having different types. The constitution of this kind of groups is particularly useful when the nodes need to build communities of cooperating nodes where each node has a specific task and rely on the fact that it can delegate other tasks it is not able to execute to one of the other members.

More in detail, the self-organization (reverse) clustering algorithm applies to a situation where a number of nodes are interconnected, forming a network: each node has a type and a set of neighbours to which it is directly connected; the number of links in the network is in general supposed to be limited, so that a situation where each node is directly connected to all the other nodes cannot happen.



1: The effect of clustering on a set of nodes. Colors indicate the type of nodes.

Both clustering and reverse clustering algorithms can function in two modes: passive or active. In *passive clustering*, a node (the *initiator*) tries to create a connection between two of its neighbours; the method is called passive because the process to connect two nodes does not start from the nodes themselves. The initiator selects two neighbours having the same type, tells each of them about the existence of the other and orders them to setup a link; the initiator also removes the link to one of the two neighbours, in order to keep the overall number of links constant. This process is iterated on all nodes of the network, with random waiting times, until a stop command is received (see Algorithm 1 for a pseudo-code description of the algorithm).

Algorithm 1 Passive Clustering Algorithm

```
1 initiator = LOCALNODE
2 for i=1 to NUM_ITERATIONS
3 do
4  if ((initiator has two neighbors)
       n1 and n2 such that n1.type == n2.type
       and (n1 != n2)) then
5      add n1 to the neighbors list of n2
6      add n2 to the neighbors list of n1
7      remove initiator from the
       neighbors list of n1
8      remove n1 from the
       neighbors list of  initiator
9  fi
10 od
```

Algorithm 2 Active Clustering Algorithm

```
1 initiator = LOCALNODE
2 for i=1 to NUM_ITERATIONS
3 do
4   if (initiator has a neighbor n0
      such that initiator.
       type != n0.type) then
5      matchmaker = n0
6      if ((matchmaker has a neighbor
         n1 such that n1.type ==
          initiator.type) and
           (n1 != initiator)) then
7         add initiator to the
          neighbors list of n1
8         add n1 to the
          neighbors list of  initiator
9         remove matchmaker from the
          neighbors list of  initiator
10        remove initiator from the
          neighbors list of  matchmaker
11      fi
12   fi
13 od
```

In the *active clustering*, instead, the initiator node will ask one of its neighbours, having a type different from its own, to act as a *matchmaker*. The matchmaker will choose among its neighbours a node which is compatible with the initiator, and tell the two to establish a connection; after that, the link between the initiator and the matchmaker is removed (see Algorithm 2).

Also reverse clustering can be executed both in the active or passive mode. The modifications with respect to the already described case are minor: in passive mode the initiator needs to select two neighbours of different types, rather than of the same type; in active mode, instead, the initiator must choose a matchmaker of its same type, and the matchmaker must look for a node which has a different type with respect to the initiator.

In [29] the characteristics of the algoritms have been evalu-

ated through simulation. In particular, the degree of *homogeneity* within the network has been used as an indicator of the success of the algorithms. Homogeneity H is defined as:

$$H = \frac{\sum_{i=1}^{N} v(i)}{L}$$

where N is the number of all nodes in the network, $v(x)$ is the number of nodes of the same type which are connected to node x, and L is the number of all the links in the network. Intuitively, the clustering algorithms are required to achieve a high level of homogeneity, while those of reverse clustering are expected to have a very low level of homogeneity. Figures 2 shows an example of the results presented in [14] that highlight the performance of the active clustering algorithm in terms of this metric (see [14] for a detailed analysis).

The first noticeable property is scalability, with the 1000-strong population converging to similar or higher homogeneity values than the 100 nodes network. On the other hand, it is possible to see how augmenting the number of different types of nodes the reachable homogeneity level slows down dramatically.

Given the result of this evaluation, it emerges the opportunity to experiment with the proposed self-aggregation algorithms within a distributed context. To this end, we have integrated the algorithms in our autonomic framework and accomplished a preliminary evaluation of them.

3. SELFLET CONCEPTUAL MODEL AND ARCHITECTURE

Our proposal is founded on the concept of *SelfLet*: a *SelfLet* is a self-sufficient piece of software which is situated in some kind of logical or physical network, where it can interact and communicate with other *SelfLets*. In this section we provide a high level overview of *SelfLets* and of the framework that allows the designer to build them. A more detailed presentation of the proposed conceptual model is given in [11].

In general, *SelfLets* require or offer some kind of services to other *SelfLets* in order to comply with their specified internal behaviour. *SelfLets* dynamically modify and adapt this behaviour in reaction to changes in their internal state or in the environment, in order to accomplish their high-level goals. The conceptual model which is presented next extends and polishes the one that has been presented in [19].

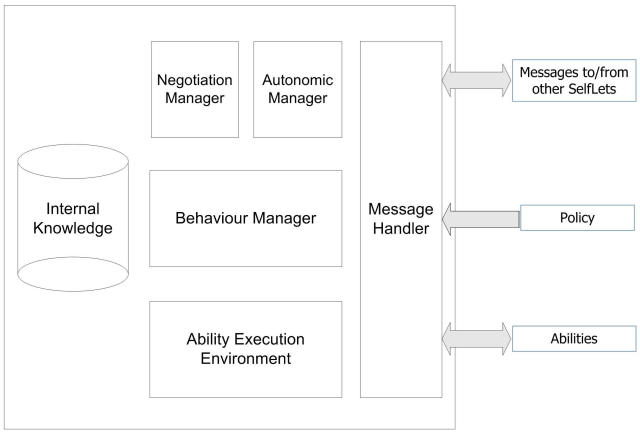
A *SelfLet* is characterized by a unique *ID* and one or more *Types*. A type is not strictly defined and is in general application dependant, for example indicating a *SelfLet* has a specific behaviour or a certain service installed: therefore, the types a *SelfLet* belongs to can change dynamically. At any given time a *SelfLet* can also belong to a *Group*, constituted by its neighbouring *SelfLets*, with which communication can take place. A *SelfLet* can belong to two or more groups, but it can also belong to a single group, which is the case when either all the *SelfLets* know each other or they form isolated islands.

A *SelfLet* can store more than one *Behaviour* and execute one at a time. Behaviors can be seen as workflows, constituted of states and transitions between states, which model the way a *SelfLet* "behaves" in the environment. The execution of behaviors can result in the invocation of one or more services; these services are called *Abilities*, which are generic applications that can perform one or more specific functions. A *SelfLet* can have different Abilities installed, can dynamically add or remove them and can pass them to other *SelfLets*.

A behavior can result in the fulfillment of one or more *Goals*. A *SelfLet* can advertise its capability to achieve a Goal, and offer this capability to *SelfLets* who need it. Thus, executing a Behaviour can involve either the local execution of one or more Abilities, or a remote request to find other *SelfLets* who can achieve a Goal needed for the execution of the Behaviour.

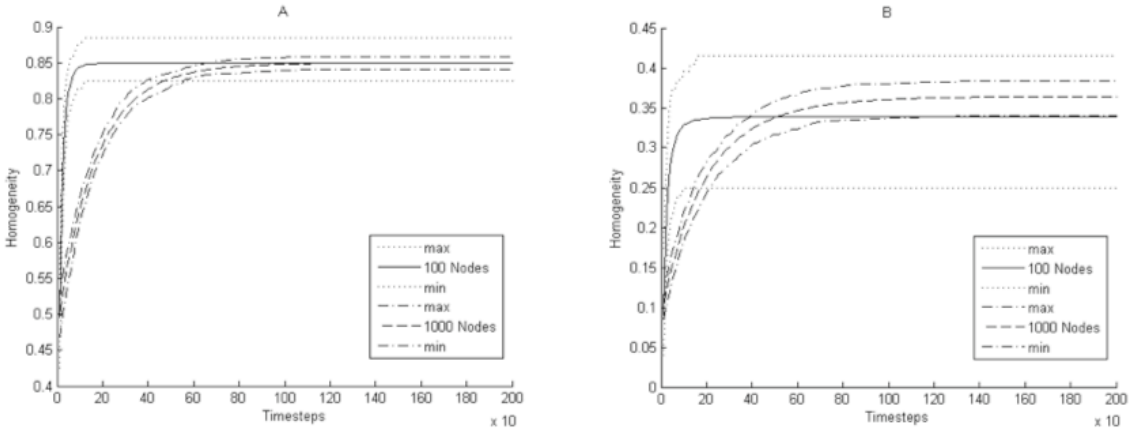
Other fundamental features of a *SelfLet* are "intelligent" rules that are executed whenever a change in the internal state of the *SelfLet* or in the environment trigger them. These are called *Autonomic Rules*. They can modify a Behaviour by adding, removing or editing states or transitions; they can also install or uninstall Abilities and decide which Goals will be offered and advertised by a *SelfLet*. For instance, in case the *SelfLet* load factor gets too high, a rule can be triggered to stop offering a certain Goal, in order to try to reduce the overall workload. Finally, Autonomic Rules can run *Autonomic Abilities*. These are specific kinds of Abilities that perform some autonomic tasks, e.g., they can create an aggregation of neighbours respecting certain properties. Collectively, we call a set of Autonomic Rules *Autonomic Policy*. In the following we present the internals of a *SelfLet* architecture and we describe the life cycle of a *SelfLet*.

3.1 Internal Architecture of a `SelfLet`



3: The internal architecture of a *SelfLet*.

The internal components which constitute a *SelfLet* are shown in Figure 3. The *Behaviour Manager* is responsible for the actual execution of the *SelfLet* Behaviours. In order to make each *SelfLet* self-adaptable, its Behaviour needs to be defined explicitly so that it can be modified at runtime. To this end, finite state machines, possibly exploiting the Statechart expressive power, can be used; and indeed our custom implementation is able to execute `Behaviour`s described in



2: Homogeneity over simulation time for two (A) and ten (B) node types. Dashed curves indicate extreme values observed in the 100 independent realisations per combination of parameter values.

terms of a StateChart coded in the XMI format generated by ArgoUML [4]. The actions that can be performed in a state or during a transition are implemented as Java code that is dynamically loaded by using *Javassist* [8]. This allows us to redefine actions while the system is running.

The Abilities of a *SelfLet* are managed by the *Ability Execution Environment*. It offers the primitives to install, store, and uninstall Abilities without restarting the *SelfLet*. Moreover, it has to take into account potential dependency constraints between Abilities and to handle versioning and updates while at the same time being as lightweight as possible, because *SelfLets* are likely to be deployed on portable devices. All these requirements has led us to the usage of an OSGi Framework [24] as the technology for building this component. OSGi, in fact, supports the dynamic installation of so called *OSGi boundles*, which are easily mappalble to our Abilities.

The *Negotiation Manager*, instead, is used to interact with other *SelfLets* by publishing the availability of certain Goals or by requesting one if needed.

The communication is performed via the *Message Handler* which is the access point for incoming and outgoing messages. The Message Handler has been implemented using the REDS Framework [9], a framework of Java classes which can be used to build publish/subscribe applications for large and dynamic networks.

The *Autonomic Manager* has the task of monitoring all the other components (and the communications between them) and of dynamically adjusting the *SelfLet* according to a given policy by firing Autonomic Rules: for example it can change the Behaviour of the *SelfLet* or uninstall unused Abilities. Thus, it needs to allow reasoning and possibly learning capabilities. To implement this component we exploited *Drools* [18], a Java implementation of a *Rules Engine*.

Finally the *Internal Knowledge* is basically an internal repository which can be used to store and retrieve any kind of information, needed by any of the *SelfLet* components: for example the Autonomic Manager can store monitoring data for future reference, or installed Abilities can record needed parameters.

For further details about the implementation of the prototype, see [10].

3.2 The SelfLet life cycle

The proposed architecture allows to develop generic autonomic applications and systems by configuring and developing only application dependent aspects.

In particular, for each type of *SelfLet* involved in the system, at least a basic Behaviour must be defined by the developer in order to define the *SelfLet* life cycle, the actions it needs to perform, the Goals it has to use or achieve and, if needed, some autonomic rules supporting application dependent self-configuration.

As Behaviours are essentially represented as finite state machines, this translates to the design of a StateChart via ArgoUML.

The next step is the definition of the Actions and the Abilities that are executed by the StateChart, and of the Goals that the *SelfLet* offers to the others. Development of both Actions and Abilities requires some coding because these are the actual services which operate according to the application's necessities. Even if these services are already existing and implemented, some code might be still needed to adapt them to the Ability interface, in order to allow interoperability with the *SelfLets*.

The last step to do is to assign an Autonomic Policy to the *SelfLets*, by defining the desired high-level guidelines for the *SelfLet* evolution: the more advanced the Autonomic Manager implementation is, the more high-level the Policy specification can be. The manager will convert the guidelines in actual Autonomic Rules, which will intervene on the *SelfLet* as needed, installing and/or executing Abilities or changing

Behaviours; technically, rules can even create a Behaviour from scratch if opportunely defined, but in general they will operate on a standard Behaviour on which to build upon.

Once these phases are performed, a *SelfLet* is ready to be run in its intended deployment environment.

4. CLUSTERING ALGORITHMS AS ABILITIES OF SELFLETS

The self-organization algorithms presented in Section 2 can constitute a fundamental block of the *SelfLet* approach. They, in fact, can be used to enable the creation of cooperating groups of *SelfLet*s starting from a condition where these *SelfLets* do not have the possibility to know all the other nodes that are present on a network, but are (physically or logically) restricted to a certain number of neighbours.

When mapping the self-aggregation terminology to our *SelfLet* model, we can say that a node corresponds to a single *SelfLet*. The identifer of the node is the *SelfLet* ID, and the type of the node is the *SelfLet* type. We know that *SelfLet*s can be organized into groups: this means that each *SelfLet* can consider the members of the same group as its "neighbors". This information is stored in the *SelfLet* internal knowledge.

Self-aggregation algorithms are implemented in our framework as Autonomic Abilities. Thus, they are encapsulated in a OSGi boundle that offers the methods to trigger the execution of the algorithm and its termination. These Abilities can be installed, either statically or at runtime in all *SelfLet*s participating into aggregation.

Aggregation can be started by a *SelfLet* Autonomic Rule for various reasons. For instance:

- The rule is triggered and activates the clustering algorithm because the *SelfLet* load has passed a certain threshold and there is not other known *SelfLet* to which the first one could delegate some of its tasks.
- The *SelfLet* already belongs to a group but the rule is triggered because it is not possible to contact some of the members of the group. In this case, a new group of neighbours has to be created.
- The rule realizes that the *SelfLet* frequently needs the execution of a Goal that it is not able to fulfill autonomously. In this case, the reverse clustering algorithm can be started to establish a stable relationship with those neighbors able to offer the required Goal.

Various other reasons for aggregation can be identified depending on the specific application the *SelfLet*s are built for. Thus, new application-dependent Autonomic Rules can be defined by the designer to trigger the execution of the aggregation algorithms.

Independently of the reason why a self-aggregation Ability of a *SelfLet* is activated, it makes the corresponding *SelfLet* behaving as an initiator that will interact with other *SelfLet*s that, in turn, following a domino approach, will start interacting with others. While in the theoretical approach described in Section 2 the algorithms are assumed to be executed for a fixed number of iterations, in the actual case they will have to stop as soon as reasonable groupings on *SelfLets* are achieved. This is obtained through an Autonomic Rule that monitors the evolution of the group a *SelfLet* is belonging to and if this remains stable in terms of the type of the participants then the *SelfLet* invokes the Ability operation for stopping the algorithm execution.

The actual integration of the self-aggregation algorithms into Abilities poses also some new issues deriving from the need of distributing their execution over a network of *SelfLet*s. More in detail, it might happen that some steps of the algorithms remain incomplete because of the failure of some participating *SelfLet*s. In this case, we have defined a rollback mechanism that allows the *SelfLet*s to go back to the situation previous to the incomplete interaction. Moreover, we have defined a locking mechanism that avoids that a *SelfLet* is contacted by two different initiators that want to start a (reverse) clustering algorithm. Finally, in a distributed setting the communication channels can be saturated by high traffic of messages. To cope with this situation, we decrease the frequency of the interaction between *SelfLets* in the case of high network and increase it again as soon as the network load. This behavior is defined as part of the algorithms implementation and, interestingly, make them behaving as a full autonomic system in miniature. This idea of nested autonomic systems is frequent in the autonomic computing as well as it is in a human body: it allows to create new levels of transparency that can simplify the system design.

5. PERFORMANCE ANALYSIS

In this section we show the first results we have obtained using our prototype to execute the *SelfLet* self-aggregation algorithms. We have previously seen that by its definition a self-organization algorithm should be able to adapt itself to the situations in which it has to be executed. After having verified that an algorithm actually works it is important to study its behavior in different situations in order to identify its applicability fields. We have performed some preliminary analyses to compare the results obtained using our implementation in the autonomic framework with the ones derived in [14]. The followed approach and the obtained results are described in the subsection 5.1 and 5.2, respectively.

5.1 Setting up the experiments

Since self-aggregation is executed by distributed *SelfLet*s, gathering global statistics is not easy at all without some kind of centralized analysis tool. In [13] a *Manager* Ability hs been defined that is executed by a single *SelfLet* which has the task of detecting all the other *SelfLet*s on the network, informing them which type of clustering algorithm to load, and creating an initial *topology*, i.e., a configuration of nodes and links; the nodes set up their neighbours according to this topology, and then they are ready to start the actual algorithm. As the nodes exchange messages, the Manager node monitors the whole network, tracks the number of messages, checks for changes in the topology and calculates the overall values of homogeneity of clusters as this is the measure of the effectiveness of the algorithms.

To execute the algorithms, we have selected different kind of starting topologies to analyze their impact on the overall algorithms performance. The simplest one is the *Random* topology, where the links between the nodes are generated in a casual way, making it possible to have various types of topologies like, for example, partitioned graphs and singletons. In this topology the number of neighbors of each node has a very high variance, making it suitable to recreate unpredictable environments. Then we have considered the *Torus* topology where the links between nodes are created in order to form a main donut-like loop. If the number of links is greater or equal to the number of nodes, the additional links are created in such a way that the single loop becomes a chain-shaped loop. The main property of this topology is that the maximum distance between two nodes is the half of the total number of nodes. Another property is that the number of neighbors of each node is almost constant. The last considered topology is the *Spiral* topology that is very similar to the torus one, but it forms a "broken" loop. All nodes can be seen as part of a spiral, that can become a chain-shaped spiral if the total number of inks is greater than the number of nodes. This topology has the same properties of the Torus one with the difference that the maximum distance between two nodes is the total number of nodes.

5.2 Analysis results

We have performed several experiments by analyzing the influence of the execution time, the number of nodes, the number of different kinds of nodes, the number of links and the initial network topology on the overall behavior of the algorithms [13]. This comparison highlighted that the selected self-organization algorithms exhibit comparable performance expressed as maximum level of homogeneity. Therefore the strengths and the weaknesses of the algorithms are confirmed also in our implementation. For example, as a positive fact, the algorithms present a good scalability with respect to the overall number of nodes (like in figure 2 A); while, on the negative side, the homogeneity level drops off when the number of node types increases (like in figure 2 B).

As an example, in the following we illustrate some of the obtained results for the clustering (and reverse clustering) algorithms, by considering both active and passive clustering and different initial network topologies. The following figures refer to a configuration with 100 nodes, 1000 links and 5 different node types. To give an idea of the algorithm performance we have collected information about the number of exchanged messages (to measure the traffic over the network) and the level of node homogeneity (defined in section 2) achieved by the algorithms. All measures have been taken assuming specific times for the termination of the aggregation. More precisely, for the clustering algorithms we show the results obtained for time intervals starting from 0 second and ending at 100 seconds with granularity of 25 seconds, since, in this case the observed results are close to an asymptote. Whereas, for the reverse clustering algorithms we have observed quite stable results also ending the execution at 20 second. For this reason we show the obtained results starting from 0 second and ending at 20 seconds with time intervals of 5 seconds only.

Tables in figures 4 and 5 show the number of messages on the network for the clustering and reverse clustering respectively.

		Active	Passive
Topology	**Time (s)**	**Msg (10^3)**	**Msg (10^3)**
Random	0	0	0
	25	19	19
	50	35	36
	75	50	52
	100	64	66
Torus	0	0	0
	25	21	19
	50	37	36
	75	55	51
	100	72	68
Spiral	0	0	0
	25	20	19
	50	36	34
	75	55	51
	100	72	67

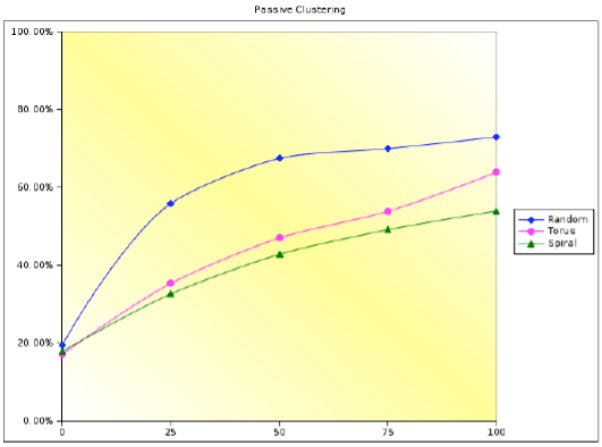
4: Clustering algorithms: Number of messages vs execution time.

		Active	Passive
Topology	**Time (s)**	**Msg (10^3)**	**Msg (10^3)**
Random	0	0,0	0,0
	1	3,0	2,9
	5	5,1	5,6
	10	5,8	9,2
	20	6,5	15,8
Torus	0	0,0	0,0
	1	2,9	2,8
	5	5,6	5,4
	10	8,9	9,0
	20	11,8	15,8
Spiral	0	0,0	0,0
	1	2,9	2,9
	5	5,7	5,3
	10	9,2	8,9
	20	13,1	15,6

5: Reverse clustering algorithms: Number of messages vs execution time.

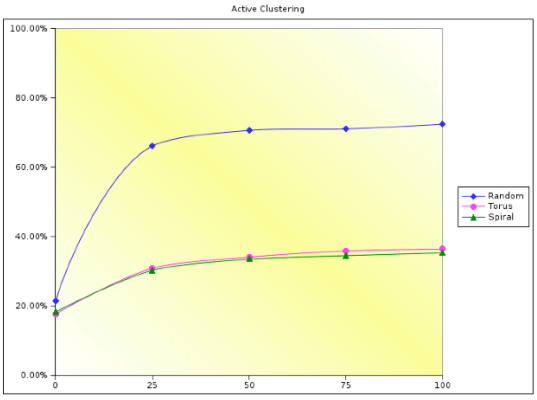
The results in Figures 4 and 6 seem to indicate that the Random topology is the best suited to achieve a high level of homogeneity (goal of the clustering) with a substantial equivalence with the other topologies in the number of exchanged messages.

Considering the active clustering (figure 7) we observe again a superiority of the Random topology with respect to the others. In particular, the Random topology for active clustering offers results that are comparable with the ones obtained for the Passive Clustering. Instead the Torus and Spiral topologies reach for active clustering a lower level of



6: Passive Clustering: Homegeneity level vs algorithm completion time.

homogeneity with respect to the case of passive clustering.

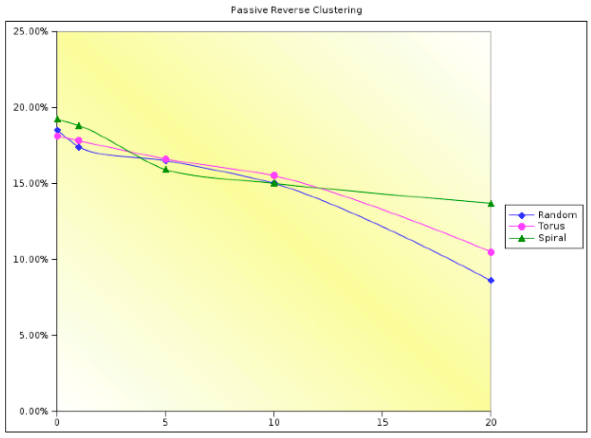


7: Active Clustering: Homegeneity level vs algorithm completion time.

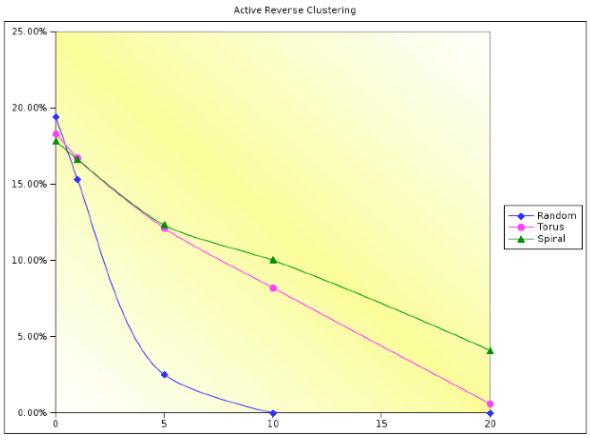
Considering the reverse passive clustering (figures 5, 8), the three different topologies present similar performance in terms of both homogeneity level (in this case it should be as low as possible) and number of exchanged messages with a slight dominance of Random topology.

The active reverse clustering algorithm (figure 9), shows a complete convergence (0% of homogeneity level) reached with the Random topology in a very short time (10 seconds). Torus and spiral topologies exhibit good performances as well (even if not optimal), which confirm a superiority of the active versus the passive version of the algorithm.

By summarizing, we can roughly say that for clustering algorithms the Random topology achieves a good level of homogeneity both with active and passive version, while torus and spiral topologies show lower performances that slightly decrease in the case of active clustering. For reverse clustering algorithms, instead, it is possible to clearly indicate the Random topology and the active version of the algorithms as the combination that shows the best performance: complete convergence with a low number of messages and with



8: Passive Reverse Clustering: Homegeneity level vs algorithm completion time.



9: Active Reverse Clustering: Homegeneity level vs algorithm completion time.

a short completion time.

This experimentation gives evidence to some critical situations that deserve further investigations. Let us consider, for example a network with two nodes of type A and one node of type B connected using three links: it has an immutable homogeneity H=33.33%, therefore H is the optimal (unique admissible) value for both clustering and reverse clustering algorithms. Another critical case is represented by network containing isolated group of nodes that cannot be reached by the (reverse) clustering steps so altering the global level of homogeneity. These examples show that the maximum level of homogeneity is often far from its bounds 0 and 1, so this index has not a fixed "goal" value (i.e. a value of 1 may be unreachable).

To overcome some of the observed shortcomings, we are investigating both the definition of new ad-hoc performance indexes and the improvement of the algorithms themselves. Some work on this direction is in progress [13]. Furthermore, we intend to use these results to write new autonomic rules and heuristics that can be used by a *SelfLet* running the clustering ability.

6. RELATED WORK

The IBM autonomic computing initiative exemplifies the applied industrial perspective on self-management [23]. This vision starts from the premise that implementing self-managing attributes involves an intelligent control loop. This loop collects information from the system, makes decisions and then adjusts the system as necessary. The academic community, on the other hand, gives raise to a project, called *AutoMate* [26], whose underlying philosophy is the same control loop. Its overall goal is to develop conceptual models and implementation architectures that can enable the development and execution of self-managing Grid applications. Other autonomic computing approaches involve or take into account GRID infrastructures [25], [22], [15], control theoretics [1], [2], negotiation theories from the economy world and even psychological or legal factors and theories [5]. All these different approaches, though, are just different facets of the same problem: there is not one approach to be preferred to the others, but all will have to be taken in consideration to build autonomic applications.

Different lines of research taking inspiration from natural adaptive systems give rise to approaches like the *emergence* [16], [3], [6] and the *multi-agent systems* ones [17], [30], [12]. Their basic idea is that a global, complex and dynamic behaviour for the whole system is very hard to define, while it is more likely to obtain such a behaviour from the interaction among the components of the system, so that the global observed behaviour emerges from these interactions. One of the most common natural examples of this is the behaviour of an ant colony, which led to several models applied to distributed computation. In general, design principles for these models are quite standard; local rules should be involved most of the time, meaning that each agent should act only according to local, low level rules, and with no need for a global view on the problem: agents usually do not have enough information to solve a high-level problem by themselves, and when cooperating to produce an emergent behaviour they are not aware that a collective choice is actually being made. Each agent should also have some independent management capabilities: the agents are often thought of as "dumb" units, with a small set of available behaviours, giving rise to something complex only by interacting; but, going back to nature, ants are not simple organisms per se, and are actually more complicated than the machines which can be built by man at the time; so artificial agents, too, should sport more complex behaviours of their own. A promising field of application is that of network-based services, which are by nature very complex, unpredictable and hard to manage in a centralized fashion or with standard technologies. Indeed several studies exist trying to use a peer-to-peer approach to produce self-organizing networks, like AntHill [7] or T-MAN [21].

Another thing to consider is that self-organization in dynamic situations evidently requires active cooperation among entities; biologically, this is an interesting problem because altruistic cooperation, which is what leads to emergent behaviours, seems to contradict Darwinian natural selection, and this translates in a similar contradiction in artificial systems. The cooperation problem has been mainly studied through the game theory, first proposed by Von Neumann and Morgenstern [31], and the well known Prisoner's Dilemma. A rational behaviour by a single agent is by nature egoistic, but this poses a problem in a cooperative environment, because the overall benefit of the system is lessened when the agents composing it "play it safe" by following rational reasonings rather than risking an altruistic behaviour: this is justifiable because in an open, unsafe environment malicious agents can exist. Since a single interaction does not allow for cooperative behaviours because of the agents' rationality, a way to overcome this is considering the concept of trust, built upon a number of previous interactions: like in nature, an agent can study the behaviour of another agent and learn which agents can be trusted and which cannot, possibly punishing "traitorous" agents but also "forgiving" them if they resume an altruistic behaviour. This approach to the cooperative problem has led to the development of reputation based mechanisms, some of which are also employed in P2P sharing programs or in the eBay system, and trust establishment schemes, which also involve the dissemination of trust information on an unsecure network [14].

In this paper we have tried to cope with the limitations of both current self-organization approaches and of autonomic computing by suggesting that an alternative approach combining the autonomic and self-organization perspectives could be the right solution to enforce effective management in complex systems. To this end we have exploited and improved the work done so far in the *CASCADAS* European project. The proposed solution is built upon existing elements that are intended to interact and aggregate on a distributed environment, giving rise to collective behaviours on a larger scale. In order to verify the feasibility of such an approach, an implementation has also been developed, trying to get to a working, albeit simplified, prototype.

7. CONCLUSIONS

In this work we have introduced the concept of self-organization algorithms in the context of autonomic systems. Specifically, the algorithms integrate into the *SelfLet* model with the role of autonomic abilities, i.e., service specialized in the

implementation of the so-called self-* proprieties.

In particular we have realized and executed a specific case of self-organization algorithms that aims at building neighborhoods of *SelfLets* on the basis of some property shared by all participating elements.

The work is complemented with a performance study whose goal is to give insights about strengths and weaknesses of these algorithms. The results of this analysis can also be used to design a *SelfLet* that can self-tune itself and behave always in the optimal way.

The work can be expanded in several directions, serving as a base upon which more refined architectures and implementations can be developed. More adaptive clustering algorithms can be defined taking into account changing environments and goals. The clustering algorithms can also be generalized so to be able to deal with domains in which the nodes have multiple types. A more ambitious objective is to provide a framework that is able to manage the input parameters for the algorithms execution by itself using some adaptable heuristic to find the critical values.

Acknowledgments

This work has been partially supported by Project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

8. REFERENCES

[1] S. Abdelwahed, N. Kandasamy and S. Neema, *"Online Control for Self-Management in Computing Systems"*. Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Toronto, Canada, May 2004.

[2] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang and Y. Lu, *"Feedback Performance Control in Software Services"*. IEEE Control Systems Magazine, Vol 23, No. 3, June 2003.

[3] R. Anthony, *"Emergence: a Paradigm for Robust and Scalable Distributed Applications"*. Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04), pp 132-139, May 2004.

[4] *ArgoUML modeling tool*, `http://argouml.tigris.org/`

[5] Autonomic Computing website: `http://www.autonomiccomputing.org/`

[6] O. Babaoglu, M. Jelasity and A. Montresor, *"Grassroots Approach to Self-management in Large-Scale Distributed Systems"*. UPP 2004, Mont Saint-Michel, France, Springer Verlag, Vol. 3566, pp. 286-296, 2005.

[7] O. Babaoglu, H. Meling and A. Montresor, *"Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems"*. In "Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)", Vienna, Austria, July 2002

[8] S. Chiba, *"Load-time Structural Reflection in Java"*. ECOOP 2000 – Object-Oriented Programming, LNCS 1850, Springer Verlag, page 313-336, 2000. Javassist website: `http://www.csg.is.titech.ac.jp/~chiba/javassist/`

[9] G. Cugola and G. Picco, *"REDS: A Reconfigurable Dispatching System"*. Technical report, Politecnico di Milano, March 2005. REDS website: `http://zeus.elet.polimi.it/reds/`

[10] D. Devescovi, *"A Conceptual Model and Architecture for Autonomic Systems Development: Design and Implementation"*. Master thesis, Politecnico di Milano, academic year 2005-2006. Available at `http://www.elet.polimi.it/upload/dinitto/papers/devescovi.pdf`

[11] D. Devescovi, E. Di Nitto, R. Mirandola *"An Infrastructure for Autonomic System Development: the `SelfLet` Approach "*. submitted for publication. Available at `http://www.elet.polimi.it/upload/dinitto/papers/selflet.pdf`

[12] D. Bonino, A. Bosca and F. Corno, *"An Agent Based Autonomic Semantic Platform"*. Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04), 2004.

[13] D. Dubois, *"Design, Development and Simulation of Self-organization Algorithms for Autonomic Systems"*. Master thesis, Politecnico di Milano, academic year 2005-2006. Available at `http://www.elet.polimi.it/upload/dinitto/papers/dubois.pdf`

[14] CASCADAS project, `http://www.cascadas-project.org/`

[15] A. J. Chakravarti, G. Baumgartner and M. Lauria *"The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network"*. Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04), 2004.

[16] T. De Wolf and T. Holvoet, *"Emergence as a General Architecture for Distributed Autonomic Computing"*. K. U. Leuven, Department of Computer Science, Report CW 384, January, 2004.

[17] Di Marzo Serugendo, G., Foukia, N., Hassas, S., Karageorgos, A. et al. *"Self-organisation: paradigms and applications"*. In: Engineering Self-Organising Systems: nature-inspired approaches to software engineering, Springer (2004)

[18] *Drools*, `http://labs.jboss.com/portal/jbossrules/`

[19] E. Hofig, B. Wust, B.K. Benko, A. Mannella, M. Mamei, E. Di Nitto *"On Cocepts for Autonomic Communication Elements"*. Proceedings of the First International workshop on Modelling Autonomic Communications Elements (MACE 2006), October 2006.

[20] P. Horn, *"Autonomic Computing: IBM's Perspective on the State of Information Technology"*. Technical Report, IBM Corporation, October 15, 2001.

[21] M. Jelasity and O. Babaoglu, *"T-Man: Fast Gossip-based Construction of Large-Scale Overlay Topologies"*. Technical Report UBLCS-2004-7, University of Bologna, Department of Computer Science, Bologna, Italy, May 2004.

[22] J. Kaufman, T. Lehman, G. Deen and J. Thomas, *"OptimalGrid – Autonomic Computing on the Grid"*. IBM article, 2003.

[23] J. Kephart and D. Chess, *"The Vision of Autonomic Computing"*. IEEE Computer 36(1): 41-50 (2003)

[24] The OSGi Alliance, *"OSGi Service Platform Core Specification"*. Release 4, Version 4.0.1, July 2006.

[25] M. Parashar and J. C. Browne, *"Conceptual and Implementation Models for the Grid"*. Proceedings of the IEEE, Special Issue on Grid Computing, IEEE Press, Vol. 93, No. 3, March 2005.

[26] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri, *"AutoMate: Enabling Autonomic Grid Applications"*. Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing, Kluwer Academic Publishers, Vol. 9, No. 1, 2006. AutoMate Project website: `http://automate.rutgers.edu/`

[27] Saffre F. and Ghanea-Hercock, R. *"Simple Laws for Complex Networks"*, BT Technology Journal, 2003, vol. 21, n.2.

[28] *"Engineering Self-Organising Applications"*, `http://esoa.unige.ch/`

[29] M Shackleton, F Saffre, R Tateson, E Bonsma and C Roadknight, *"Autonomic computing for pervasive ICT - a whole-system perspective"*, BT Technology Journal, 2004, vol. 22, n.3.

[30] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, I. Whalley, J. O. Kephart and S. R. White, *"A Multiagent Systems Approach to Autonomic Computing"*. Autonomous Agents and Multi-Agent Systems, 2004.

[31] J. von Neumann and O. Morgenstern, *"The Theory of Games and Economic Behavior"*. Princeton University Press, 1944.