On Exploiting Decentralized Bio-inspired Self-organization Algorithms to Develop Real Systems

Elisabetta di Nitto, Daniel J. Dubois, Raffaela Mirandola
Dipartimento di Elettronica e Informazione
Politecnico di Milano
{dinitto, dubois, mirandola}@elet.polimi.it

Abstract

The current research trends in Software Engineering are focusing on the development of new techniques to deal intelligently and efficiently with the design of systems that are able to evolve overtime and adapt to rapid changes of their requirements. In particular, the field of Autonomic Computing has been created to study these types of systems with the ultimate aim to create systems that are able to self-configure, self-optimize, self-heal and self-protect without any external intervention. What we study in this paper is a set of the most relevant bio-inspired principles that may be applied to these systems. We discuss how to apply them to develop or adapt self-organization algorithms to real evolvable systems and we present two examples of applications that we have developed.

1. Introduction

The current research trends in Software Engineering are focusing on the development of new techniques to deal intelligently and efficiently with the design of systems that are able to evolve overtime and adapt to rapid changes of their requirements. In particular, the field of Autonomic Computing [26] has been created to study these types of systems with the ultimate aim to create systems that are able to self-configure, self-optimize, self-heal and self-protect without any external intervention. More recently the Autonomic Computing area has started very strong interactions with different related fields such as Pervasive Computing [39], Situational Computing [8], etc: all these fields have in common the fact that they need to be open in order to follow the changes that may happen in the external world [6].

To reduce the dependency on manual interventions many different software architectures have been proposed. IBM proposed a generic architecture [1] of an autonomic system composed by two entities: a manager and some managed resources. In this approach the manager communicates with the resources through a sensor/actuator mechanism and the decision is elaborated using the so-called MAPE cycle in which the manager *Monitors* the sensors, *Analyzes* the collected data, *Plans* an action, and then *Executes* it using the actuators. In addition to this preliminary research there are other different approaches such as the model-based adaptation of Garlan [19] in which the architecture of the system may change at runtime depending on what is being monitored, and a more recent work from Kramer&Magee [27], where they try to exploit the analogies between robotic systems and self-managed systems. Other self-adapting architectures that use different approaches are: Autonomia [20], which uses a two layer approach, where the first contains the execution environment and the other manages the resources; AutoMate [31] which has a multi-layered architecture optimized for scalable environments such as decentralized middleware and peer-to-peer applications; SelfLet [12] and CASCADAS [22] approaches, which have both been designed to have runtime changing goals, plans, rules, services and a behavior modeled as a final state machine that may change overtime.

While the above approaches work at the level of the application by suggesting a way to offer it some self-* properties, in literature there are other types of approaches that are specifically focusing on the middleware-level: they show self-adapting capabilities for optimizing some of middleware properties/metrics. For instance, in distributed-dispatcher publish-subscribe systems some approaches are presented that enable the reconfiguration of the underlying topology of the distributed dispatcher to minimize some cost metrics such as the network traffic [3, 25, 28], while in peer-to-peer networks based on unstructured overlays other approaches are mainly focusing on mechanisms to search/share some information [36, 33, 38].

All the described architectures are supported at run-time by an algorithmic self-organizing logic. These algorithms may be used into different levels of existing systems. An example is the distributed monitoring mechanism presented

in [5]: special architectural components called supervisors collect status data from the underlying components and decide whether to trigger or not corrective reactions.

Another class of self-organization algorithms is composed of bio-inspired algorithms. The main characteristic of these algorithms is the fact that they are based upon a set of principles inspired by the natural world and that they provide simple solutions to solve problems that would be much harder using classical approaches.

The purpose of this paper is to focus on bio-inspired self-organization algorithms and to present our initial experience in exploiting them in the development of distributed software architectures. These types of algorithms often need to be tailored to the real system in which they are executed. Finding the correct tailoring is not always trivial since it requires extensive modeling and simulations in the presence of an execution context that may be continuously changing.

We first discuss the most common principles that we may encounter in bio-inspired self-organization, we present some of the algorithms that have been proposed in the literature, and we then discuss, with the help of two examples, on the issues that arise when we apply and compose these principles and algorithms to real systems.

The organization of this paper is as follows. Section 2 presents a definition of self-organization, a classification of the most recurring bio-inspired self-organization principles, and examples of algorithms that use them. In Section 3 we discuss on the issues that arise when adapting bio-inspired self-organization algorithms to real systems. Sections 4 and 5 present the application of a bio-inspired approach to the cases of a distributed heterogeneous load balancing algorithm and of the reconfiguration of the overlay network for a publish-subscribe system. Finally, Section 6 concludes the paper and shows some future research challenges that are left open by this work.

2. Bio-inspired Self-organization

Self-organization is defined as *"The spontaneous evolution of a system into an organized form in the absence of external pressures"* [21]. Forms of self-organization that come from observed phenomena of the natural world are called *bio-inspired self-organization*.

In this section we propose a list of the most important principles of bio-inspired self-organization and some algorithms that are based upon them.

2.1. Principles

Bio-inspired Self-organization is usually based upon common principles [2] taken from the natural world that may be composed and translated into algorithms. The following is a list of some of these principles.

Noise. This principle [30] is defined as the use of some kind of perturbations that move the system away from its expected goal in the short term, but makes it possible with a certain probability to reach a better goal in the long term. In other words if the system goal is measured by a goal function that should be maximized, then the use of noise makes it possible to increase the probability to move away from the local optima of that function and then to move to the global optimum. This principle works in a similar way to what we see in genetic programming [23] in which the noise is added by using the mutation/crossover operations. Noise is usually present in most bio-inspired systems regardless their design and it is particularly useful in improving the solution in optimization problems.

Emergence. This principle [24] is defined as the capability of a system composed by multiple components to reach, with a certain probability, a global goal by achieving local goals at component level that may be apparently unrelated to the global goal. Examples of emergent behaviors are very common in nature, like the phenomenon of fireflies synchronization, in which the local goal of a firefly is to modify its blinking frequency to synchronize with another firefly in its sight range, while the global goal that emerges is having a community of fireflies that blink at the same time. Creating an emergent property in a software system is not always obvious: a possible simple method may be to decompose, in a top-down way, the global problem into smaller subproblems translated into simple rules to be run by system components, however in most of the cases components rules cannot be easily derived from the global goal and, vice-versa, global goals are gradually discovered in a bottom up way from the analysis of the characteristics of local goals.

Diffusion. This principle [11] is defined as a method for communicating information among many interconnected components. According to this principle the information produced or received by a node is sent to some other nodes regardless the destination of the message. The destination nodes may be any neighbor of the sending node (flooding) or some random nodes (gossiping). The final aim of this mechanism is to increase the probability that nodes interested to that message actually receive it. Diffusion is usually used in peer-to-peer networks; possible uses are the search or synchronization of information. This communication method usually gives self-healing/fault-tolerance properties to the system since messages are usually redundant and thus the removal (or malfunction) of some system components does not prevent the correct destinations to receive the messages.

Stigmergy. This principle [7] is defined as another method of communicating information among different moving components of a system. According to this principle the information is not sent to other nodes, but it is stored in the environment. Since components are capable of moving, when they change their location (context) they may read in the new environment the information previously left from another component, use it, and possibly update it. Environmental information may be persistent or it may expire after a timeout. This phenomenon in nature may be noticed under the form of pheromone evaporation. Since this principle relies on leaving information on the environment, the failure of some components does not compromise the communication (assuming that the probability of an environment failure is negligible).

Evolution. This principle [18] is defined as the capability of the system to improve itself using a *natural selection* process among its components: the best components tend to survive, the worst components tend to die. During this process best components may be replicated, partially mutated, and recombined with other components.

2.2. Algorithms

In this section we explain some examples of bio-inspired algorithms presented in literature that are related to the principles discussed above.

Genetic algorithms [23] are examples of application of the evolution principle. These type of algorithms consider an initial *population* of preliminary solutions to a problem and then, using a *fitting* function, they quantify the quality of each solution. Then the worst solution are discarded and the best ones are left. To avoid local optima in the solution the best solutions are combined together and changed slightly using the crossover and mutation operations. This last operation is an application of the noise principle.

A different class of algorithms is the epidemic algorithms class. These algorithms resemble the spread of a contagious disease and rely heavily on the diffusion principle. They have been proved to work in the following classes of problems: failure detection, data aggregation among internal knowledge of system components, creation of groups, etc. An example of application of such algorithms in distributed system has been proposed by Guerraoui et al [17].

An example of emergence is provided also in a work from Saffre et al [34]. In this work each system component starts as a generic worker and then, if certain conditions are met after an interaction among components, components may decide to differentiate themselves and become more efficient in performing specific tasks, with the cost of being no longer able to execute generic tasks.

Another very popular class of bio-inspired algorithms was proposed by Dorigo [16]: the Ant Colony Optimization algorithm. This algorithm has been used to solve combinatorial optimization problems by reducing them to the generic problem of ants looking for the optimal path from their anthill to the food source. This is a typical example in which we can see all the principles applied to the same problem.

A bio-inspired study in which our research group was involved in the last years include the solution of the components aggregation problem in dynamic networks [13, 35], in which each component, using simple interactions, is able to efficiently rewire the network to increase the probability that each component will eventually increase the number of similar components in its neighborhood. We will use this algorithm as an example in Section 4.

Some of other popular examples of bio-inspired self-organization focus on the problem of reorganizing the topology of the nodes in a communication network [29, 32, 38, 14]. These approaches have in common the fact that they are able to make emergent properties appear in the network topology using different metrics to decide whether to add/remove links among nodes, moreover they rely often on the concept of noise (such as what happens in Cyclon [38] when performing the periodical shuffling of neighbors nodes).

2.3. Stochasticity

A common characteristic of bio-inspired algorithms is their stochasticity. This characteristic derives from the fact that bio-inspired principles are able to apply self-organizing properties in a probabilistic way. This means that they may be used to increase the probability to self-organize the system, but do not assure that the system actually self-organize. This may be confusing at the beginning since in classical software engineering any algorithm whose preconditions are satisfied may either work (correct behavior) or not work (wrong behavior). When we use bio-inspired algorithms our question is no longer *when they actually work*, but *what the probability that they work is*. Therefore, as long as the correct-behavior probability is respected, the fact that an algorithm does not work or that the system stays in a wrong status in presence of verified preconditions is not considered a wrong behavior, but the expected behavior. The result is that all the properties of the system (both functional and non-functional) are expressed in stochastic terms. Some of the advantages of stochasticity are the fact that the system tends to keep working in presence of temporary unexpected behavior of its components. A possible drawback is the absence of guarantees on when a specific goal is reached: this may make bio-inspired approaches less suitable in solving time-constrained problems.

3. Using Bio-inspired Self-organization in Real Systems

So far, bio-inspired approaches have been studied in theory and, in some cases, developed in toy examples. In our research we are currently trying to apply them in some real systems and therefore we are trying to understand if there is any repeatable approach that we can adopt to reach this goal.

According to our experience, in some cases, the self-organization algorithms that we have presented in Section 2 can be simply adopted to address a specific problem (we will show in the following as one of these can be applied to the case of distributed load balancing). In some other cases none of the identified algorithms can be applied as they are, but, indeed, we can rely on the adoption of some of the self-organization principles (we will show in the following as some of the principles we have identified can be applied to the case of the reconfiguration of a publish/subscribe architecture). As soon as we have identified the algorithms or the principles that are most suitable to a specific case, according to any well-defined software engineering approach, we need to build a model for our system to check that it behaves as expected. This step is particularly important here because of the inherent complexity and distribution of the systems we consider, and on the consequent difficulty of building them directly without any design level check on the feasibility of the approach.

As soon as we are convinced that our model works, we can start implementing it. Even in this phase we should not be limited to a simple model-to-code approach since we have to face with non-trivial problems that are not usually faced in the theoretical definition of the algorithms and principles, and therefore could have not been captured in our model. These problems often concern synchronization among components, management of race conditions, the driving of the system toward some initial state that is suitable for the algorithm to start, the identification of the conditions under which the self-organization algorithm could start and end, and when to actually repeat an iteration of the algorithm without using too much computational time, but still achieving a good level of responsiveness. In fact, it is very common that bio-inspired self-organization algorithms are proposed and studied only as a set of rules to be executed for every iterations by abstracting away from all the details that cause the issues that we have listed above.

The solution to these problems depends not only on the problem/solution model itself, but also on the deployment scenario: a wrong implementation choice at this level may nullify the effectiveness of the algorithm at all. Thus, even in this phase, the role of models and analysis and simulation approaches is prominent in order to provide some level of guarantees on the quality of the implementation.

The last aspect to be taken into account has to do with the stochasticity of the self-organization approaches. We need to be aware of this and, in case we cannot tolerate it, we have to structure the system in such a way that, when it is not able to converge to the expected state in a time interval that is acceptable for the specific application we are developing, some other more predictable mechanisms take the lead and guarantee the expected convergence.

In the following we describe two examples in which we try to show what we did to address the aforementioned issues.

4. Example: Heterogeneous Decentralized Load Balancing

This section presents the issues that have been raised when applying self-organization to the problem of balancing load in a heterogeneous distributed system. Details on the approach that we have adopted can be found in [15].

Let us assume a system composed by interconnected generic nodes. Each node may be seen as a resource that is able to process jobs organized in a queue. Each node is associated to a type that defines which job(s) it is able to receive and process. Nodes of the same type are defined as homogeneous, nodes of different type are defined as heterogeneous. The network is characterized by high levels of churn [37] and dynamism, meaning that nodes may appear and disappear in an unpredictable way. Each node may communicate only with its direct neighbors and may transfer jobs to them provided that they are of the same type. Jobs may be generated from the environment and sent to random nodes (always respecting the type-matching between the job itself and the destination node).

In this setting the final goal of heterogeneous load balancing is to distribute the jobs evenly to all the nodes of the network, keeping into account the heterogeneity of nodes. A possible metric to measure the distance from the goal is the throughput of the system in terms of completed jobs in a fixed amount of time. The final goal is to maximize this metric.

In the literature there is plenty of load-balancing algorithms that can deal efficiently with the case of homogeneous nodes, however none of them is able to properly perform an efficient load balancing in the presence of both heterogeneity and dynamism. Our idea has been to combine one of these algorithms (we adopt the Dimension-Exchange Load Balancing Algorithm [10]) with a rewiring algorithm to cluster the nodes into several domains of homogeneous nodes and perform load balancing in these domains. The resulting algorithm works as shown in Figure 1 where the circles represent nodes, the arcs the network connections between them, the colors represent the possible types, and the numbers represent the number of jobs that each node

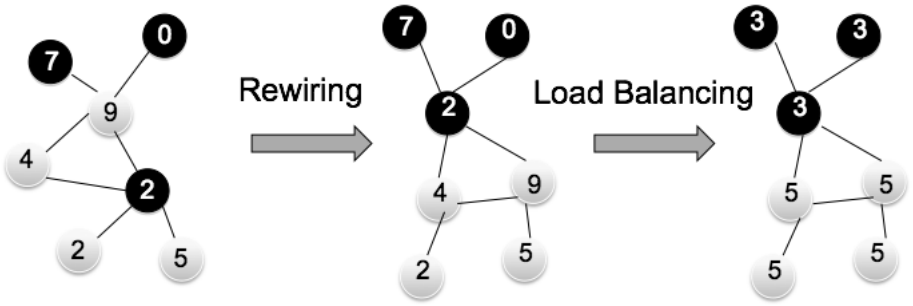

Figure 1. Combined approach for decentralized heterogeneous load balancing. Circles represent nodes, color represents their type, the number inside represents the size of the queue of each node.

has in its queue. The rewiring algorithm prepares the network by creating clusters of homogeneous nodes, while the load balancing algorithm is applied to the clusters. As discussed in Section 3 we have modeled the resulting system and in [15] we have shown a clear improvement in the performance of the system.

Satisfied of this theoretical result, we have started developing the actual implementation of the system using a toolkit to build autonomic system that has been developed in the CASCADAS project [22] and we have faced most of the issues that we have mentioned in Section 3. In particular:

- *Handling concurrent iterations*: this happens when two algorithm iterations that are executed at the same time try to modify the state of a node at the same time. The solution we have adopted has been to use a locking mechanism to prevent concurrent modifications. This has resulted in an increase of the messages exchanged between nodes, but we checked, through an accurate simulation model, that, fortunately, this increase did not impact significantly on the advantages shown by the approach in the theoretical model.
- *Activation/Termination problem*: keeping the rewiring algorithm always on has clearly an impact on the performance of the system as it increases the network traffic. A way to tackle this issue is to use the diffusion principle to diffuse the start/termination message to all the nodes when needed. In the development of our example we have chosen to maintain the system balanced all the times, therefore all nodes execute the algorithm during all their life-cycle, but, as discussed in the next item, we have worked on the calibration of the frequency of the algorithm iteration.
- *Choice of the interval between algorithm iterations*: an interval that is too short may overload the network with messages, an interval that is too long may slow down the load balancing process. The solution we have adopted has been to properly calibrate the interval that maximizes the throughput in the deployment environment. As discussed in the next paragraph, such calibration has been achieved by simulating the execution of the system in various conditions as discussed in the next paragraph.

Table 1. Preliminary results of experiments on the effects of the threshold x_c in presence of different job execution times and different values for the network overhead. Total network load is 10000 jobs and the duration of the simulation is 200s.

JobTime	Overhead	x_c	Jobs Done	Msgs
10s	0.5s	0	1700	1500
10s	0.5s	500	800	400
10s	0.5s	inf	300	50
0.1s	0.1s	0	8000	2100
0.1s	0.1s	500	10000	1600
0.1s	0.1s	inf	6000	50
0.001s	0.1s	0	5000	120
0.001s	0.1s	500	8000	100
0.001s	0.1s	inf	10000	50

Calibration of the interval between rewiring iterations. Since in our previous study we have seen that most of the network messages are caused by the rewiring iterations we need a heuristic to reduce the interval between the rewiring iterations when they are not needed. The idea is the following: if a node has not many jobs it does not require to perform many rewiring iterations, while if a node is overloaded it would ask its neighborhood for similar nodes in a much more frequent way. A possible way to implement this

idea is to decide to start the rewiring iterations with a probability that depends on the size of the local queue of jobs like, for example with the following probability function:

$$P(x) = \frac{1}{(1 - \frac{x}{x_c})^2}$$

Where x is the queue length, x_c is a special parameter that is related to the minimum queue length that is needed to have probability close to 0.5.

The system has been simulated to see if there is an actual improvement in finding an optimal value using different values for x_c and the results are reported in Table 1: the first column contains the job execution time (that includes not only the actual job processing time, but also the time needed to transfer the job to the executor node), the second column contains the network overhead (the time that is needed to initialize and execute the transfer), the third column contains the considered threshold value for x_c, the fourth and fifth columns contain the number of processed jobs and exchanged messages after 200 seconds of simulation. Simulations have been executed using a network of 100 nodes connected using a scale-free [4] topology with an average degree of 4 neighbors per node, 10 different types, and a static initial distribution of 10000 jobs. These experiments have shown that using such function to trigger the execution of the rewiring part of the load balancing algorithm may speed-up the execution of the jobs in situations in which the network overhead is a concern.

The results we have obtained may be used to self-adapt the threshold value by comparing the node service time to the network delay (evaluated – for example – by pinging remote nodes): for high service times the threshold will be gradually decreased to zero to speed-up the node rewirings; if the network latency becomes closer to the node service time, the threshold will be modified to an opportune intermediate value (the exact value for this situation will be the base for a future work); finally if the service time is significantly lower then the network latency, there will be no need for further rewiring and the threshold parameter may grow toward infinite values (thus disabling all rewiring iterations). Another important point of discussion is the message overhead: in several applications its minimization may be more important than achieving the best value for the number of completed jobs. In such a case the threshold value may be automatically increased until the desired maximum network load is reached, thus finding the best trade off between the network overhead and the algorithm performance.

5. Example: Overlay Self-organization in Publish-subscribe Systems

This section presents the application of bio-inspired self-organization to reconfigure the overlay topology of a network of brokers in a publish/subscribe system. Let us assume a system composed by interconnected entities called *brokers*. Each of these entities may have zero or more components connected. Each component may be connected only to one broker and it may publish messages and subscribe to messages produced by some other component. Each broker is in charge of collecting the subscriptions from its components and of forwarding them to all the other brokers. This way every broker can maintain a routing table that is used to route every published messages only to components that have previously subscribed to them (subscription-forwarding approach [9]). Another assumption is that the publish-subscribe system allows a broker to change its neighbors at runtime.

In this setting the final goal is to minimize the number of messages that traverse the brokers network by rewiring the connections among the brokers. To achieve this goal we could not use any of the algorithms that we have identified in Section 2.2 as the problem to be tackled depends on very specific information, that is, the subscriptions that are known by each broker and the expected pattern of traffic in the network. Instead, we could adopt the emergence principle to build a utility-function based algorithm in which the maximization of such function at local level moves the whole system toward the global goal. Moreover, we have also shown that the usage of the noise principle further improves the solution.

The preliminary results are reported in [14] where we define the utility function that, given a generic node N_0 and two of its neighbors called N_1 and N_2, is able to estimate the number of saved messages after removing the link between N_0 and N_1 and adding a link between N_1 and N_2. Therefore a generic broker N_0 decides to perform the rewiring using the two neighbors that are able to maximize the utility function.

Similarly to the previous example, implementing just the algorithm that evaluates this utility function is not enough, but we need to choose an interval between iterations and a proper locking mechanism. A possible implementation of an algorithm iteration that maximized the utility function is shown in the sequence diagram of Figure 2.

Like in the previous case we show how it is possible to improve this algorithm by adding noise to the utility function.

Using noise to refine the utility function. The easiest way to find a refinement for this solution is to try to add noise to the system by reducing the threshold for the utility

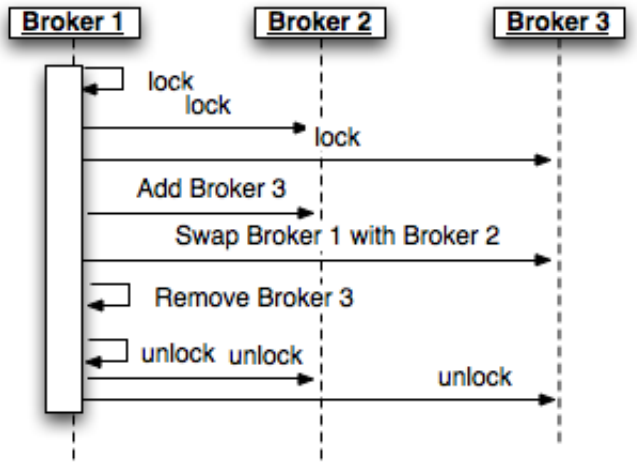

Figure 2. Sequence diagram of a rewiring iteration that maximizes the utility function.

Table 2. Comparison on the performance of our algorithm (described in [14]) in presence/absence of noise.

Noise	Traffic	Reconfigurations
No	44000	10
Yes	40000	19

function: instead of choosing neighbors that maximize it, we relax this constraint by allowing possible rewirings that increase slightly the total number of messages that traverse that broker (negative utility function). Using this approach the initial iterations of the algorithm are slowed down, but, after a while, we escape the local optima and the global solution becomes better than the one obtained using the approach without noise. A simulation of the algorithm with the presence/absence of noise has been presented in Table 2: the first column specifies the presence/absence of noise, the second one the amount of system traffic after algorithm convergence, and the last one the number of rewirings that are needed to move from the initial state to the most optimized state. Analyzing the table we can see that the total network traffic after algorithm convergence is much less in the case with noise, but the number of iterations to reach the convergence is lower in the case without noise.

6. Conclusions and Future Work

In this work we have studied the problem of applying bio-inspired self-organization techniques to real systems. We have started this study surveying different principles that describe natural self-organizing phenomena and we have presented some existing self-organizing classes of algorithms that exploit the above principles. We have seen also that existing bio-inspired algorithms tend to be presented through toy-models that are often difficult to bring as part of a real system implementation. This study aims at understanding whether it is possible to find a common approach to actually use these types of algorithms and to deal with some of the implementation issues that may arise.

The study is currently in the preliminary phase of the experimentation, but what we have learned so far is that the process of solving a problem with a self-organization algorithm is not trivial and requires extensive reasoning about which algorithms/principles to use and especially how to apply them. Differences in the real execution environments have a heavy impact on how the algorithms should be calibrated.

A possible further development of this work is to extend the proposed approach to a more complete methodology consisting in a complete list of steps and design patterns that may be used to apply bio-inspired principles to real systems in a more systematic way. Another possible future outcome is finding a way to eliminate completely the need for self-adapting the algorithm parameters (such as the interval between different iterations) without relying on extensive simulations on the real environment in which the algorithms would be used.

Acknowledgements

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] IBM Autonomic Computing Toolkit - User's Guide. `http://download.boulder.ibm.com/ibmdl/pub/software/dw/autonomic/books/fpu3mst.pdf`.

[2] Ö. Babaoglu, G. Canright, A. Deutsch, G. A. D. Caro, F. Ducatelle, L. M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes. Design patterns from biology for distributed computing. *ACM Trans. Auton. Adapt. Syst.*, 1(1):26–66, 2006.

[3] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. *Comput. J.*, 50(4):444–459, 2007.

[4] A.-L. Barabasi and A. Reka. Emergence of Scaling in Random Networks. *Science*, 286:509–512, 1999.

[5] L. Baresi, S. Guinea, and G. Tamburrelli. Towards decentralized self-adaptive component-based systems. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 57–64, New York, NY, USA, 2008. ACM.

[6] L. Baresi, E. D. Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

[7] R. Beckers, O. Holland, and J. Deneubourg. From local actions to global tasks: stigmergy and collective robotics. In *ALIFE IV*. Brooks & P. Maes, MIT Press, Cambridge (Mass), 1994.

[8] S. H. C B Anagnostopoulos, Y Ntarladimas. Situational computing: An innovative architecture with imprecise reasoning. *Journal of Systems and Software*, 80(12):1993–2014, 2007.

[9] G. Cugola, E. D. Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[10] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, 1989.

[11] A. Datta, S. Quarteroni, and K. Aberer, editors. *Autonomous Gossiping: A Self-Organizing Epidemic Algorithm for Selective Information Dissemination in Wireless Mobile Ad-Hoc Networks*, volume 3226 of *LNCS*, Boston, MA, USA, 2004. Springer Berlin / Heidelberg.

[12] D. Devescovi, E. Di Nitto, D. J. Dubois, and R. Mirandola. Self-Organization Algorithms for Autonomic Systems in the SelfLet Approach. In *Autonomics*. ICST, 2007.

[13] E. Di Nitto, D. J. Dubois, and R. Mirandola. Self-aggregation algorithms for autonomic systems. *Bio-Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd*, pages 120–128, Dec. 2007.

[14] E. Di Nitto, D. J. Dubois, and R. Mirandola. Overlay self-organization for traffic reduction in multi-broker publish-subscribe systems. In *submitted for publication*, 2009.

[15] E. Di Nitto, D. J. Dubois, R. Mirandola, F. Saffre, and R. Tateson. Applying self-aggregation to load balancing: Experimental results. *Bio-Inspired Models of Network, Information and Computing Systems, 2008. Bionetics 2008. 3rd*, Nov. 2008.

[16] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Book, 2004.

[17] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulieacute;. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004.

[18] D. B. Fogel. What is evolutionary computation? *IEEE Spectr.*, 37(2):26–32, 2000.

[19] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.

[20] X. D. Hariri, S. L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *IEEE International Performance, Computing, and Communications Conference, 2003.*, 2003.

[21] F. Heylighen and C. Gershenson. Information systems, may/june 2003. the meaning of self-organization in computing.

[22] E. Hoefig, B. Wuest, B. K. Benko, A. Mannella, M. Mamei, and E. Di Nitto. On concepts for autonomic communication elements. In *International Workshop on Modelling Autonomic Communications*, 2006.

[23] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[24] J. H. Holland. *Emergence: from chaos to order*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[25] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 543–550, New York, NY, USA, 2007. ACM.

[26] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[27] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. *Future of Software Engineering*, 0:259–268, 2007.

[28] M. Migliavacca and G. Cugola. Adapting publish-subscribe routing to traffic demands. In *DEBS '07: Proceedings of the 2007 inaugural International conference on Distributed event-based systems*, pages 91–96, New York, NY, USA, 2007. ACM.

[29] T. Nakano and T. Suda. Applying biological principles to designs of network services. *Appl. Soft Comput.*, 7(3):870–878, 2007.

[30] S. Nicolis and al. Optimality of collective choices: a stochastic approach. *Bulletin of Mathematical Biology*, pages 65, 795–808, 2003.

[31] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Cluster Computing*, 9(2):161–174, 2006.

[32] C. Prenhofer and C. Bettstetter. Self-organization in communication networks:principles and design paradigms. *IEEE communication magazine*, 2005.

[33] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, pages 329–350, 2001.

[34] F. Saffre, J. Halloy, M. Shackleton, and J. L. Deneubourg. Self-organized service orchestration through collective differentiation. *IEEE Trans Syst Man Cybern B Cybern*, 36(6):1237–46, 2006.

[35] F. Saffre, R. Tateson, J. Halloy, M. Shackleton, and J. L. Deneubourg. Aggregation Dynamics in Overlay Networks and Their Implications for Self-Organized Distributed Applications. *The Computer Journal*, 2008.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[37] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC*, pages 189–202, New York, NY, USA, 2006. ACM.

[38] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. Network Syst. Manage.*, 13(2), 2005.

[39] M. Waldrop. Pervasive computing - an overview of the concept and exploration of the public policy implications, Mar. 2003.