Modeling and Simulation of Autonomic P2P Systems Based on Altruistic Policies

Michele Amoretti, and Gianni Conte, and Marco Muro, and Marco Picone, and Francesco Zanichelli

Abstract—Autonomic systems should be able to detect, diagnose and repair failures and adapt their behavior to changes in the environment. For totally decentralized systems of this kind, robustness can be obtained by relying on peer-to-peer protocols and altruistic policies. Modeling and analyzing such complex systems can be very challenging, due to the fact they are characterized by emerging behaviors that cannot be easily determined in advance. Our research work in this area is supported by two main tools. The first one is a modeling tool called Networked Autonomic Machine (NAM), allowing for the characterization of self-* networks whose nodes are endowed with pluggable functional modules and services. The second tool is DEUS, a discrete event simulation environment that is at the same time very flexible and straightforward to use. The paper presents a sample autonomic system that performs load sharing based on altruistic policies, whose performance is evaluated with reference to different task allocation strategies.

Index Terms—modeling, simulation, peer-to-peer, autonomic computing, context awareness

1 Introduction

The main goal of a distributed system is to connect users and resources in a transparent, open, and scalable way. Ideally this arrangement is drastically more fault tolerant and powerful than many combinations of stand-alone computer systems. Nevertheless, large-scale distributed applications are becoming more and more demanding in terms of efficiency and flexibility of the technological infrastructure, for which traditional solutions based on the client/server paradigm are not always suitable.

In the last decade the peer-to-peer (P2P) paradigm has emerged as a highly appealing solution for scalable and high-throughput resource sharing among decentralized computational entities, by using appropriate information and communication systems without the necessity for central coordination [16]. In distributed systems based on the peer-to-peer paradigm all participating processes have the same importance. In contrast with the client/server approach, in which resource providers and resource consumers are clearly distinct, on the contrary peers usually play both roles. Furthermore, a peer-to-peer system is a complex system, because it is composed of several interconnected parts that as a whole exhibit one or more properties (*i.e.* behavior) which are not easily inferred from the properties of the individual parts [14].

The P2P paradigm is one of the main topic of our research. Currently we are focusing on *autonomic* P2P systems, being able to detect, diagnose and repair failures and adapt their behavior to changes in the environment. We argue that, by increasing the context-awareness of monitoring data exchanged by autonomic peers, it is possible to efficiently sense network conditions and the level of provided services and perform corrective actions. Sharing context-based information can be realized through dissemination of specific data among different nodes or through cross-module and cross-layer messages inside the same node. For example, a QoS entity responsible to allocate network resources may exchange context-aware information with other nodes in order to identify changes in the network conditions. In section 2 we discuss related work on policy-based autonomic systems.

In this paper we present the two main tools we use to model and analyze such kind of complex distributed systems. The first one (illustrated in section 3) is a modeling tool called *Networked Autonomic Machine (NAM)*, allowing for the characterization of networks whose nodes are provided with pluggable functional modules and services. The second tool (described in section 4) is DEUS, a discrete event simulation environment that is very flexible and straightforward at the same time.

In section 5 we present a sample autonomic system that performs load sharing based on altruistic policies. We depict such a system as a NAM network whose nodes are characterized by a functional module for connectivity, a functional module for task management, and a set of services that supports task migration. Then we illustrate some significant simulation experiments that we performed using DEUS, to show how the simple rules applied locally by a NAM are related to the global behavior exhibited by the whole system.

Finally, in section 6 we conclude the paper summarizing its main results and outlining future work.

2 Related Work

Autonomic computing (AC) is based on the assumption that the increasing complexity of distributed systems

• *The authors are with the Department of Information Engineering, University of Parma, Via Usberti 181/a, 43124 Parma, Italy. Contact: michele.amoretti@unipr.it*

Manuscript received May .., 2010; revised .., 2010.

is becoming a limiting factor for further development. The solution proposed by the AC research community is to provide systems with four key properties: self-configuration, self-healing, self-optimization, self-protection [**?**]. Efforts to design self-managing systems have yielded many impressive achievements, yet the original vision of AC remains largely unfulfilled. As suggested in [7], researchers should adopt a comprehensive systems engineering approach to create effective solutions for next-generation large-scale distributed systems, for example by merging networking, software engineering and artificial intelligence. In our research activity, we take into account related concepts like *context awareness*, *policies*, *ontologies*, *evolutionary algorithms*, etc. combined with the peer-to-peer paradigm.

Until now, few significant attempts have been done for defining a specification language for autonomic systems. IBM has suggested a reference model for autonomic control loops [10], which is sometimes called the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop. This model is being widely used to communicate the architectural aspects of autonomic systems. Nevertheless, in our opinion the assumption of a complex *Autonomic Manager* for each autonomic element makes MAPE-K too inflexible. A rather interesting approach is based on the *chemical programming paradigm* [3], that captures the intuition of a collection of cooperative components which evolve freely according to some predefined constraints (reaction rules). System self-management arises as a result of interactions between members, in the same way as intelligence emerges from cooperation in colonies of biological agents. Our modeling tool (NAM) follows the same approach, but uses a simpler formalism.

Several European research projects recently proposed the use of context-aware information for improving monitoring mechanisms. FOCALE architecture [9] is based on dynamic control loops supplemented by the DEN-ng context information model at the node level. MOMENT [12] proposed an ontology for network monitoring tools but did not address any autonomic principles. In CONTEXT [5], the synergy obtained between context-awareness, ontologies and autonomic networking promotes the definition of a new, extensible, and scalable knowledge platform for the integration of context information and services support. Finally, in EMANICS [8] a new concept for context models is presented following the policy based management paradigm, suitable for representing information and managing services in cross-layered environments.

A model for designing policy-based autonomic components is presented in [4]. The article envisions components guided by *condition-action* policies and proposes a strategy for conflict detection among policies. The approach (mainly based on artificial intelligence) is convincing, even though a clear modeling approach is missing.

3 Networked Autonomic Machines

A Networked Autonomic Machine (NAM) is a formal model of a hardware/software system that is able to

- communicate with other NAMs;
- execute a number of functional modules that may provide/consume services or context events;
- dynamically deploy, undeploy and migrate functional modules and services.

These features support local self-management and self-healing activities. Conversely, the achievement of global autonomicity, in a system of NAMs, depends on the policies adopted at the functional level. For example, by providing all NAMs with a peer-to-peer overlay management module, it may be possible to enable their cooperation in routing messages for discovering new services or functional modules to use or download and execute.

Several kinds of hardware platforms are considered: PCs and workstations, notebooks, PDAs, smart-phones, as well as sensors and actuators. NAMs can be of different types and complexities, depending on their resources (OS, processor type, memory, I/O type, battery, connectivity) and functionalities (*e.g.* communication protocols, processing features, sensing capabilities).

With respect to other modeling tools, NAM allows to specify the migration of functional modules and services among nodes. The NAM formalism can be used to semantically characterize and compare the elements of a self-*, highly dynamic distributed system.

3.1 Formal Description

Formally, a NAM node is a tuple

$$NAM =< R, F >$$

where R is a set of physical *resources*, such as CPU cycles, storage, bandwidth, and F is a set of *functional modules*.

Each functional module $f \in F$ plays one or more of the following roles:

- context provider,
- context consumer,
- service provider,
- service consumer.

The formal definition is

$$f =< S_f, C_{in}, C_{out}, POL >$$

where S_f is the set of provided services, C_{in} is the set of consumed context events, C_{out} is the set of provided context events, and POL is the set of policies according to which the functional module can react to the sensed environment:

$$POL = \{p : C_{in} \rightarrow C_{out} \times S_f\}$$

A service consists in a unit of work executed by a service provider to achieve the results desired by a service consumer. In general, a *service* is a tuple

$$s =< I, O, P, E >$$

where I is a set of input parameters, and O is a set of output parameters. Each I/O parameter has a type, *i.e.* a class (still using the ontological terminology). It is important that service consumers and service providers share the same domain ontologies in order to have a common understanding of shared services. Semantic descriptions of services are used to organize service advertisements in centralized or distributed repositories, allowing to efficiently retrieve and use services in the network. P and E are the precondition and effect sets, respectively. Such optional parameters are expressed in the form of logical conditions which can assume the true or false value. Preconditions must be verified in order to invoke the service, while an execution effect may become a precondition for the successive invocation in a composition scenario. For example, in an ambient intelligence scenario, a service for switching lights may be simply defined by the effect of assigning the value "ON" to the "status" property of a "living room light". The IOPE approach is adopted for example in the OWL-based Web service ontology called OWL-S [11].

An *atomic service* is defined as the minimal executable function unit, that cannot be decomposed and whose execution can transform a given state to another state. It is represented as a tuple:

$$as =< I, O, P, E, Q >$$

where Q is the set of exposed quality-of-service (QoS) properties, such as reliability, availability, performance, security and timing. Each node can provide different atomic services. The number of concurrent service instances and the quality of service (QoS) of each instance at a certain time depends on the current availability of hardware resources on the node.

Intra- or inter- NAM services can be statically or dynamically aggregated (proactively or on-demand) to realize new complex tasks. A *composite service* is a tuple:

$$cs =< I, O, P, E, Q, G_w >$$

where G_w is the rule that allows to combine atomic services; this rule is represented as a directed workflow graph

$$G_w =< S_w, L_w >$$

where S_w is a set of services (both atomic and composite) and L_w is a set of links that represent transitions (*i.e.* I-O connections) among services.

Figure 1 shows the internal structure of a NAM, with resources, functional modules, and services.

NAMs are able to dynamically reconfigure their structure, by adding new functional modules or services, or discarding those that are no more necessary. The *migration* of functional modules and services is another NAM's feature defined as

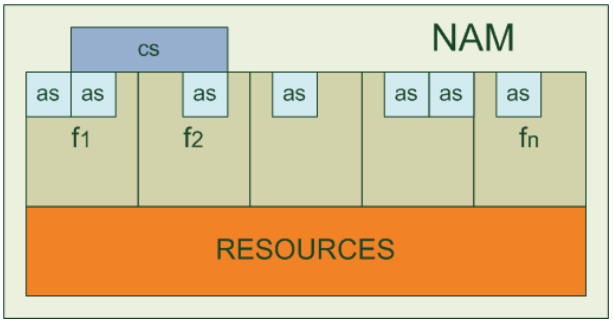

Fig. 1. The internal structure of a NAM. Functional modules use resources and provide atomic services (as), that can be aggregated in order to provide composite services (cs).

$$m_f =< NAM_u, NAM_d, f >$$

$$m_s =< NAM_u, NAM_d, s >$$

where $f \in F$ is a functional module, $s \in S$ is a service, NAM_u is the uploader and NAM_d is the downloader. A migration can start only if the resources of the downloader match with the set R_m of resources (*e.g.* running environment, libraries, CPU, RAM, etc.) required to run the functional module or the service to be migrated.

3.2 Autonomic Reasoning

Functional modules use *ontologies* to represent contextual information, services and resources. Each module may be provided with a more or less sophisticated reasoning capability, either based on statically defined or dynamically learnt policies.

Policies are the central means of control for autonomic computing systems. They consist of goals, scope, constraints according to which the system should produce behavior that users find reasonable. In the NAM perspective, a functional module may be provided with a rule engine that, using a set of rules, is able to derive new facts (output) from known facts (input).

Facts and rules should be defined according to an ontology shared among NAMs. The ontological approach tries to reduce the gap between informal language and programming language by means of formal ontology languages. Formal ontology languages have a precise semantics that allow for unambiguous specifications of domain knowledge. These languages are easy enough to learn quickly (much faster than programming languages), not just by software engineers, but also by domain experts.

Figure 2 illustrates a reasoner, *i.e.* a functional module that interact with its environment, that may include local modules and remote NAMs, but also anything

outside than the NAM network (*e.g.* operators, end users, the weather, etc.). Interaction with the environment is based on two pillars: perception, that means context-awareness, and decision, which is the (policy-based) ability to infer new context events, service calls, or output (to some users, or to a service request).

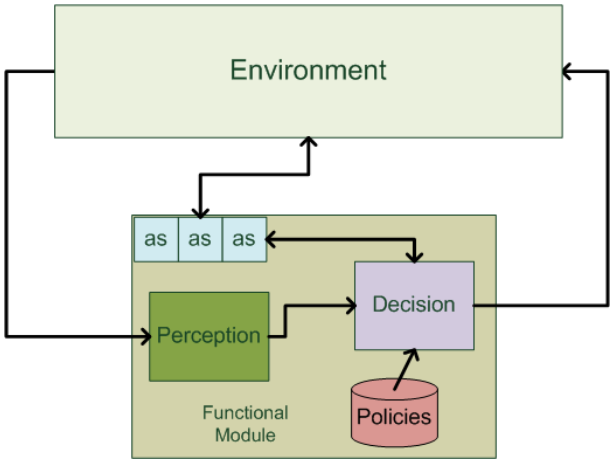

Fig. 2. A functional module that consumes context events (perceives the environment) and, based on policies, decides how to interact with the environment.

There are a variety of languages which can be used for representation of conceptual models, with varying characteristics in terms of their expressiveness, ease of use and computational complexity [17]. Normann and Putz suggest to use Prolog to define ontologies for ambient intelligence systems [13]. We argue that Prolog may be applied to any autonomic distributed system, once provided with mechanisms for checking the consistency of rules. Indeed, the set of rules may change over time, but new rules cannot be in conflict with existing rules - unless we remove such previously defined rules.

Examples of basic rules are

$$C(x) : -A(x), B(x)$$

that means: "for each x it holds $C(x)$ whenever $A(x)$ and $B(x)$ hold", and

$$C(x) : -A(x); B(x)$$

that means: "for each x it holds $C(x)$ whenever $A(x)$ or $B(x)$ hold".

More concretely, let us consider the following example that refers to a Grid Computing architecture. "Local CPU occupation is more than 60%" is a fact. "Whenever local CPU occupation is more than 60%, notify interested NAMs" is a rule. "If neighbor's CPU occupation is more than 60% and local CPU occupation is less than 30%, help neighbor" is another rule.

In our framework, two connected NAMs can interact if and only if they share the same ontology, or at least if their ontologies partially overlap. The two NAMs may use any reasoning module they like (being it Prolog-based, or not). In section 5 we illustrate an example distributed autonomic system based on altruistic policies, but before, in next section, we introduce the DEUS simulation environment which has been used to quantitatively analyze the sample system.

4 DEUS Simulation Environment

Discrete event simulation works by maintaining a list of events sorted by their scheduled event times. Executing events results in new events being scheduled and inserted into the event list as well as events being deleted and removed from the event list. The *Discrete Event System Specification (DEVS)* [18] is the most general discrete event modeling tool, which allows representing and simulating any system having a finite number of changes in a finite interval of time. In that way, systems modeled by Petri Nets, State Charts, Event Graphs, and even Difference Equations can be seen as particular cases of DEVS models.

It is possible to define a hierarchical simulator for hierarchical DEVS coupled models, consisting of devs-simulators and devs-coordinators. Shortly, each DEVS atomic model is simulated by a devs-simulator, and each DEVS coupled model is simulated by a devs-coordinator that manages the simulators of the sub-components. Since these can be themselves DEVS coupled models, a devs-coordinator is able to manage also devs-coordinators.

Our general-purpose simulation environment, called *DEUS*, follows a different approach. Its main strengths are the ease of use and flexibility: its Java API [6] allows developers to implement (by subclassing) any kind of

- nodes (*i.e.* the parts which interact in a complex system, leading to emergent behaviors: humans, pets, cells, robots, intelligent agents, etc.)
- events (*e.g.* node births/deaths, interactions among nodes, interactions with the environment, logs, etc.)
- processes (either stochastic or deterministic, constraining the timeliness of events)

Despite DEUS has been designed having in mind the three basic concepts listed above, without considering any particular modeling tool, it can be mapped on DEVS, *e.g.* a DEUS node may represent a DEVS system characterized by a set of possible states. In DEUS, transition functions may be implemented either in the source code of the events that can be associated to the node, or in the source code of the node itself.

In a previous work [1] we described the design approach pursued while developing the core modules of DEUS. Here we introduce for the first time the Visual Editor that allow to generate XML documents describing simulations, and the so-called Automator (shown in figure 3), which allows to parametrize the simulation and to automatically generate statistics in a Gnuplot-compliant format. Currently we are working on a new version of DEUS that supports parallel (multicore and/or distributed) simulations.

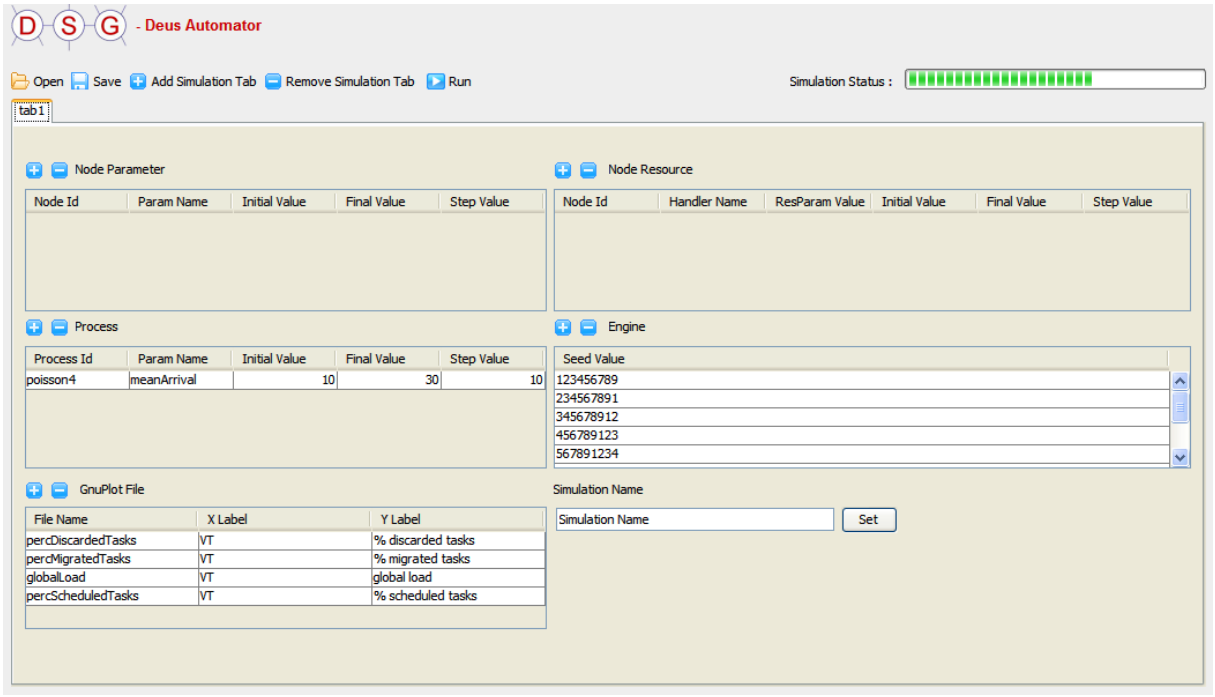

Fig. 3. The Automator tool of DEUS. The screenshot refers to one of the experiments illustrated in section 5.

5 Modeling and Simulation Example

To show the potential of NAM and DEUS, let us consider a large-scale distributed systems whose nodes provide their computational and storage capacity for the execution of tasks. Such a system may be a collection of computing nodes, a Grid, or a Cloud, depending on how resources are made available.

We model each node as a $NAM < R, F >$ with F including the following modules:

- f_{om}: an overlay management module that allows NAM to connect according to some distributed strategy
- f_{tm}: a task management module provided with a set of services that support task migration.

Due to the lack of space, we omit the details of the formal specification of such modules as NAMs, and we describe their simple load-sharing policies using natural language. The emphasis here is not on the specific load sharing strategy, but rather on the flexibility and effectiveness of DEUS as a tool to evaluate such kind of autonomic P2P systems.

In traditional peer-to-peer architectures, nodes that need a resource have to perform a query across the overlay network and wait for a positive response. Our example, on the contrary, is built around altruistic neighborhood. Given a node n, its neighbors are nodes willing to support n and will help it if n becomes overloaded, and vice versa n will help them when necessary. Thus, according to policies that are similar to the one defined in section 3.2, the f_{tm} module of n notifies neighbors if n is not able to schedule an assigned task. Such context events are catched by neighbors, and those which have enough resources offer their help (by means of task migration services) in order to achieve effective load sharing.

To simulate such a system with DEUS, a single type of node, called NetworkedAutonomicMachine, is used, together with a set of processes that manage the scheduling of events in the queue of the simulator. Scheduled events are:

- BirthEvent: a general-purpose event that creates a peer - in this case a NetworkedAutonomicMachine
- ExpTopologyConnectionEvent: a general-purpose event that connects a peer to m randomly chosen existing peers - see below for details
- TaskEvent: an application-specific event that assigns a task to a randomly chosen NAM and implements the algorithm illustrated below
- FreeResourceEvent: an application-specific event that represents the deallocation of a completed task from the peer that executed it
- LogStatsEvent: an application-specific event that logs information about the autonomic network (*e.g.* the percentage of discarded tasks)

The FreeResourceEvent is declared in the configuration, but its scheduling is managed internally to the TaskEvent's implementation, for which it is not necessary to associate it to a process and to the default DEUS *Engine*.

Although DEUS allows for simulating network churn, for simplicity we assume that the network topology is stable and completely shaped before the task assignment process can start. The overlay management module f_{om} applies a strategy that leads to a network topology with-

out preferential attachment. Starting from N_0 completely connected nodes, each other joining peer chooses m existing peers, with even probability (this function being implemented by the ExpTopologyConnectionEvent, being scheduled 10000 times in a first slot of virtual time). All connections are bidirectional, *i.e.* if node n_a has node n_b in its peerview, then n_b has n_a in its peerview. The node degree k of the resulting network has exponential distribution:

$$P(k) = (1 - e^{-\frac{1}{m}})e^{1-\frac{k}{m}} \forall k \geq m \quad (1)$$

whose expected value is

$$E(k) = \sum_{k=0}^{\infty} kP(k) = \frac{e^{(1-\frac{1}{m})}}{1 - e^{-\frac{1}{m}}} \simeq em \quad (2)$$

In these experiments, we used $m \geq 1$ as a parameter for the analysis of the load distribution strategy. Figure 4 illustrates the node degree distribution for a network with 10^4 nodes, grown with $m = 3$.

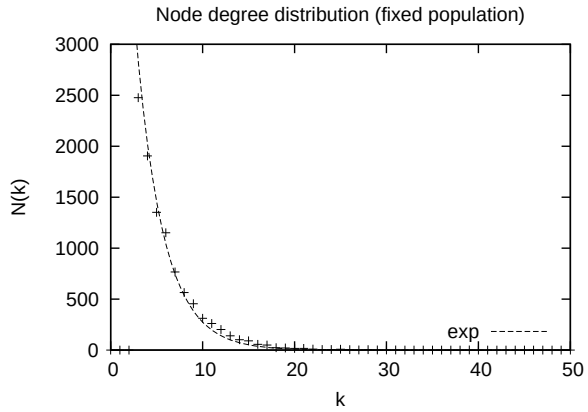

Fig. 4. Exponential node degree distribution of a network with 10^4 nodes, grown with $m = 3$.

Without loss of generality, peers are characterized by the same amount of initial free resources (capacity C), which is completely consumed when a task is assigned to the peer (task load $L = C$). When a task is assigned to a node that is already executing another task, the node sends a notification message to its neighbors. If one of the neighbors is able to help the requester, the task migrates to the helper peer. If all neighbors are busy, the task is discarded (strategy A) or locally enqueued (strategy B).

Tasks enter the system according to a Poisson process with average interarrival time t_i (which is another parameter for the analysis of the load sharing strategy). We assume that all tasks have the same duration $t_e = 10min$, and that task migration introduces an overhead $t_o = 1min$. In a future work we will investigate the effect of varying these parameters, too.

The analysis method we adopted is "repeated trials" [15]. For each configuration of the simulation we performed several runs, using different seeds for the pseudo-random number generator. Despite the length in terms of virtual time was enough to overtake the transient period and to reach steady-state values, each run lasted few (real) seconds, thanks to the highly optimized engine of DEUS.

We firstly consider strategy A, for which a task that cannot be executed neither by the target NAM nor by its neighbors, is discarded. Figure 5 illustrates the percentage of migrated tasks during the a lifetime of $10h$ of the system, a network with 10^4 nodes (grown with $m = 3$), with task interarrival time of 100, 200, $300ms$.

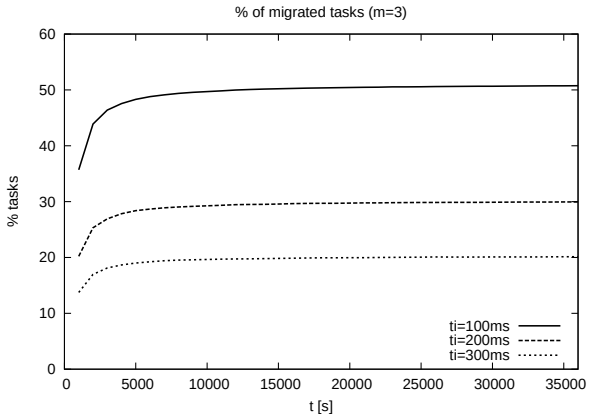

Fig. 5. Percentage of migrated tasks when the network grows with $m = 3$, with task interarrival time of 100, 200, $300ms$.

We may also have shown similar graphs illustrating the percentage of discarded tasks and the percentage of tasks that are directly scheduled (and executed) by the nodes to which they are initially assigned. In general, we observe that after the transient period, when the number of arriving tasks compensates the number of completed tasks, the percentages reach stable values.

More interestingly, figure 6 shows that the percentage of migrated tasks increases with m, while the percentages of scheduled and discarded tasks decrease with m. The observed behavior matches our expectations, because the higher m, the higher the probability of finding an unloaded neighbor. The increase of migrations allows for more nodes to be involved in the altruistic policy, for which when a new task is generated, the probability that the target node is unloaded is lower. The figure refers to the cases of $t_i = 100ms$ and $t_i = 300ms$.

If strategy B is applied, *i.e.* instead of discarding tasks, they are re-scheduled after t_e/k, where k is the number of neighbors, no task is discarded. We analyzed this strategy simulating a burst of tasks (all having the same duration $t_e = 10min$) in the time interval $100 - 120s$, with $t_i = 1ms$, and comparing the completion of the tasks when $m = 1, 3, 10$.

Since the applied load is $2 \cdot 10^4 \cdot C$, which is twice the maximal capacity of the overall network, we observe that the higher m, the sooner saturation is reached (because task migration is facilitated by higher average

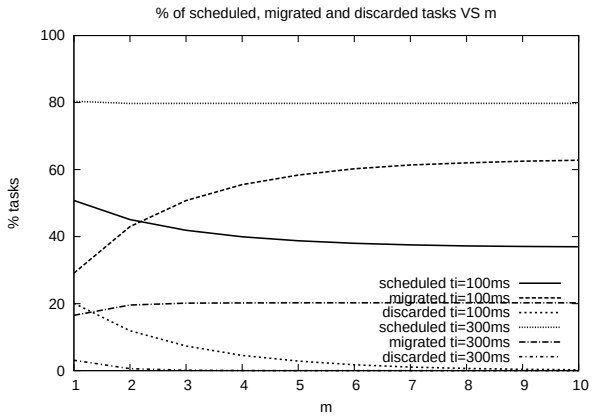

Fig. 6. Percentage of migrated and discarded tasks versus m, with task interarrival time of 100 and $300ms$.

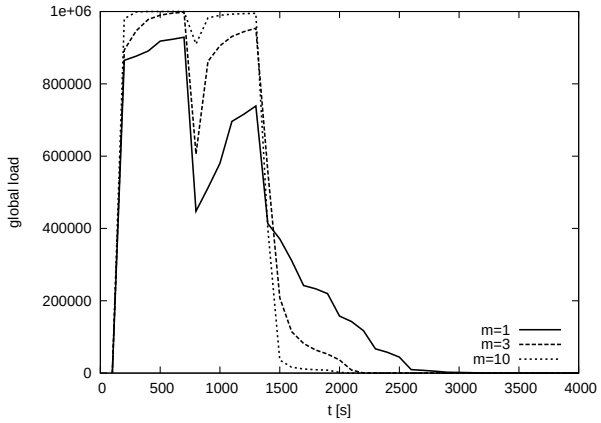

Fig. 7. Global load after a burst of tasks with $t_i = 1ms$, burst interval $100 - 120s$, fixed task duration and re-scheduling time based on the number of neighbors.

node degree). Then, having all tasks the same length, it is possible to infer that half the burst of tasks will be completed around $t = 720s$, when the second half start being scheduled or migrated. This is the reason of the two humps, whose separation decreases with m increasing (figure 7).

We performed other simulations related to strategy B, with the same burst of tasks ($t_i = 1ms$, burst interval $100-200s$), considering tasks and task migrations having random duration - with exponential distribution, and average value $t_e = 10min$ and $1min$ respectively. Figures 8, 9, 10, 11 respectively refer to the following cases: constant re-scheduling time equal to t_e (shortly, CR), re-scheduling time t_e/k (CRK), random re-scheduling time with exponential distribution and average value t_e (RR), random re-scheduling time with exponential distribution and average value t_e/k (RRK).

Looking at these results, we notice that the higher m, the better task allocation. Moreover, it appears that CRK and RRK lead to the best task allocation, with appropriate m value. But we have to consider another important variable, *i.e.* the number of re-schedules n_r,

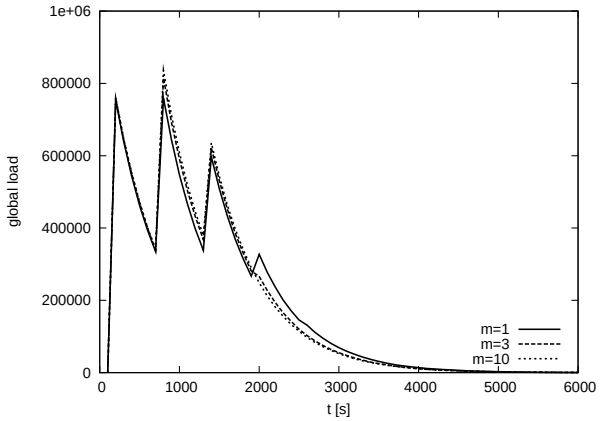

Fig. 8. Global load after a burst of tasks, with $t_i = 1ms$, burst interval $100 - 120s$, random task duration, constant re-scheduling time equal to t_e (CR strategy), and m assuming three significant values.

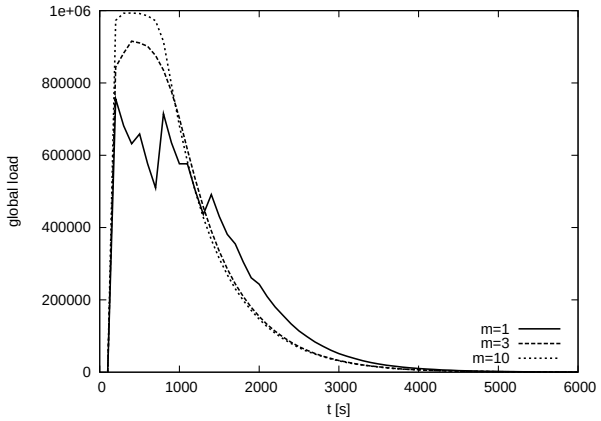

Fig. 9. Global load after a burst of tasks, with $t_i = 1ms$, burst interval $100 - 120s$, random task duration, re-scheduling time t_e/k (CRK strategy), and m assuming three significant values.

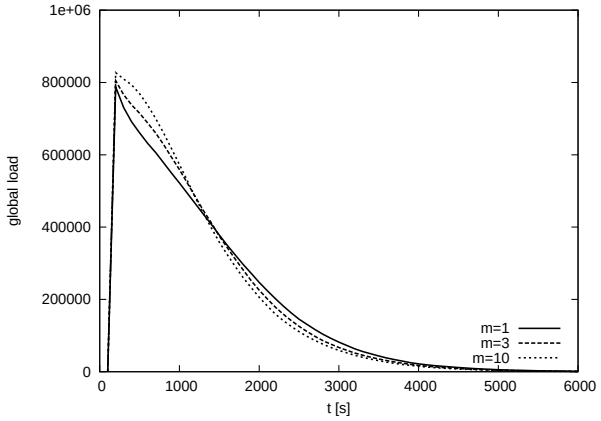

Fig. 10. Global load after a burst of tasks, with $t_i = 1ms$, burst interval $100 - 120s$, random task duration, random re-scheduling time (RR strategy), and m assuming three significant values.

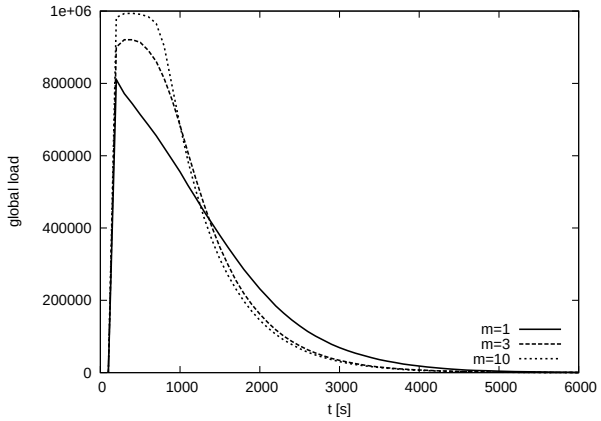

Fig. 11. Global load after a burst of tasks, with $t_i = 1ms$, burst interval $100 - 200s$, random task duration, random re-scheduling time depending on the node degree (RRK strategy), and m assuming three significant values.

	m=1	m=3	m=10
CR	16261	13905	13380
CRK	16993	29317	86272
RR	15913	13122	11234
RRK	18866	31360	86990

TABLE 1
Number of re-schedules.

which usually leads to new notification messages and consequently increases the traffic over the network. The results summarized in table 1 show that when the re-scheduling time depends on the node degree k, by increasing m the number of re-schedules dramatically increases. On the contrary, when the re-scheduling time does not depend on the node degree k, when m increases the number of re-schedules decreases.

To summarize, simulations have shown that the efficiency of strategy B depends on how we set m and the re-scheduling strategy. A re-scheduling time that is inversely proportional to k leads to good task allocation but also to increased overhead and bandwidth consumption. It is necessary to reach a trade-off between m and the dependency of the re-scheduling time on k. Such a compromise may be dynamically achieved, *e.g.* by means of adaptive evolutionary strategies like those we have described in [2].

6 Conclusion

In this paper we have emphasized the importance of policies (in particular, altruistic policies) and context-awareness for the design of autonomic P2P systems. We have presented the modeling tool and the simulation tool that we use to study such kind of systems, namely the NAM formal description and the DEUS simulation environment.

We have proposed an example of autonomic P2P system with altruistic peers, modeled with NAM and simulated by means of DEUS. Both tools showed their effectiveness, NAM by emphasizing functional modularity of nodes, DEUS by allowing for fast deployment and execution of effectual simulations.

As future work, we plan to extend the NAM formal model to make it suitable for the analytical evaluation of functional modules, and to improve the performance of DEUS (a parallel version will be released soon), in order to study more complex autonomic systems.

References

[1] M. Amoretti, M. Agosti, F. Zanichelli, *DEUS: a Discrete Event Universal Simulator*, 2nd ICST/ACM International Conference on Simulation Tools and Techniques (SIMUTools 2009), ISBN 978-963-9799-45-5, Roma, Italy, March 2009.
[2] M. Amoretti, *Fulfilling the Vision of Fully Autonomic Peer-to-Peer Systems*, Proc. of the 2010 IEEE International Conference on High Performance Computing & Simulation (HPCS 2010), Caen, France, June 2010.
[3] J.-P. Banatre, Y. Radenac, P. Fradet, *Chemical Specification of Autonomic Systems*, Proc. 13th International Conference on Intelligent and Adaptive Systems and Software Engineering, Nice, France, July 2004.
[4] G. A. L. Campos, A. L. B. de P. Barros, J. T. de Souza, J. C. Junior, *A Model for Designing Autonomic Components Guided by Condition-Action Policies*, IEEE Workshops on Network Operations and Management Symposium (NOMS), Salvador da Bahia, Brazil, April 2008.
[5] The CONTEXT project, *http://context.upc.es/index.htm*.
[6] DEUS home page, http://code.google.com/p/deus/
[7] S. Dobson, R. Sterritt, P. Nixon, M. Hinchey, *Fulfilling the Vision of Autonomic Computing*, IEEE Computer Magazine, January 2010.
[8] The EMANICS project, *http://www.emanics.org*
[9] B. Jennings, S. van der Mer, S. Balasubramanium, D. Botvich, M. O. Foghlu, and W. Donnelly, *Towards autonomic management of communications networks*, IEEE Communications Magazine, vol. 45, no. 10, pp. 112-121, 2007.
[10] IBM, *An architectural blueprint for autonomic computing*. Tech. rep., 2003.
[11] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara, *Bringing Semantics to Web Services: The OWL-S Approach*, Proc. of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC), San Diego, California, USA, July 2004.
[12] The MOMENT project, *http://www.fp7-moment.eu/*.
[13] I. Normann, W. Putz, *Ontologies and Reasoning for Ambient Assisted Living*, AALiance Conference, Malaga, March 2010.
[14] J. M. Ottino, *Engineering complex systems*, Nature, 427, 399, 2004.
[15] H. Perros, *Computer Simulation Techniques: The definitive introduction!*, NC State University, 2009.
[16] D. Schoder, K. Fischbach, *Peer-to-Peer Paradigm*, Proc. 37th IEEE Int'l Conference on System Sciences, Hawaii, USA, 2004.
[17] R. Stevens et al., *Ontology-based knowledge representation for bioinformatics*, Briefings in Bioinformatics 2000 1(4):398-414; doi:10.1093/bib/1.4.398
[18] B. P. Zeigler, H. Praehofer, T. G. Kim, *Theory of Modeling and Simulation*, 2nd Edition, Academic Press, 2000.