2013 IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops

The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing

Philip Mayer[1], Annabelle Klarl[1], Rolf Hennicker[1], Mariachiara Puviani[2], Francesco Tiezzi[3], Rosario Pugliese[4], Jaroslav Keznikl[5], Tomáš Bureš[5]
[1]Ludwig-Maximilians-Universität München, Germany [2]Università di Modena e Reggio Emilia, Italy [3]IMT Institute for Advanced Studies Lucca, Italy [4]Università degli Studi di Firenze, Italy [5]Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic

***Abstract*—Autonomic computing — that is, the development of software and hardware systems featuring a certain degree of self-awareness and self-adaptability — is a field with many application areas and many technical difficulties. In this paper, we explore the idea of an autonomic cloud in the form of a platform-as-a-service computing infrastructure which, contrary to the usual practice, does not consist of a well-maintained set of reliable high-performance computers, but instead is formed by a loose collection of voluntarily provided heterogeneous nodes which are connected in a peer-to-peer manner. Such an infrastructure must deal with network resilience, data redundancy, and failover mechanisms for executing applications. We discuss possible solutions and methods which help developing such (and similar) systems. The described approaches are developed in the EU project ASCENS.**

I. Introduction

At the very latest, the seminal article of Kephart and Chess [1] has brought the awareness of autonomic computing ideas to the global computer science researcher community (pun intended). Autonomic systems, which work in distributed environments and react to unforeseen, dynamically evolving situations, require self-* (self-star) properties, which include self-awareness, self-expression, and self-adaptation [2]. Adapting to new situations is necessary not only on the level of individual components, but also on a collaboration level.

The EU project ASCENS [3] is one of the many initiatives contributing to the vision of autonomic computing. It advocates an approach in which autonomic systems are formed by individual building blocks called *service components* (SCs) which are combined in a dynamic manner to form *service component ensembles* (SCEs). ASCENS has the goal of developing a coherent, integrated set of methods and tools to build software for ensembles; a specific focus lies on foundational issues that arise in the development of these kinds of systems. All results produced by ASCENS can be found online [4].

In this paper, we discuss one of the case studies of the ASCENS project, which is a vision of an autonomic cloud: A cloud which is based on voluntary computing and using peer-to-peer technology to provide a platform-as-a-service. We call this cloud the *Science Cloud Platform* (SCP) since the cloud is intended to run in an academic environment (although this is not crucial for the approach). We present the idea of such a cloud system along with some of the methods of ASCENS which have been used in this context. We believe that awareness is a key enabler of the SCP, and we hope to shed more light on its role in this paper.

The remainder of this work is structured along the lines of the methods used in the case study. Firstly, we introduce the idea of the cloud itself (section II). Afterwards, three sections introduce ASCENS methods which have been used to model and develop the cloud (awareness patterns in section III, modeling in section IV and system specification in section V). A prototype implementation in Java is discussed in section VI. An excursion into the area of mobile cloud computing is presented in section VII. After a discussion of related work in section VIII, we conclude in section IX.

II. An Autonomic Cloud

The idea behind the scenario we discuss in this paper is that of an autonomic cloud computing platform; or, in other words, a distributed software system which is able to execute applications in the presence of certain difficulties such as leaving and joining nodes, fluctuating load, and different requirements of applications to be satisfied.

We integrate elements from three different computing areas to set up this vision, which will be discussed in the following three subsections; these are cloud computing, voluntary computing, and peer-to-peer computing.

A. Cloud Computing

Firstly and obviously, we deal with *cloud computing*. Cloud computing refers to provisioning resources such as virtual machines, storage space, processing power, or applications to consumers "on the net": Consumers can use these resources without having to install hardware or software themselves and can dynamically add and remove new resources.

There are three commonly accepted levels of provisioning in cloud computing, which are infrastructure, platform, and software. In the first, low-level resources such as virtual machines are offered. In the second, a platform for executing custom client software is provided. On the third level, complete applications (such as an office suite) is provided, mostly directly to end users. In any case, clouds are usually offered from one or more centrally coordinated locations; the servers providing the infrastructure run in a well-maintained data center and are under the control of a single entity.

978-1-4799-5086-7/13 $31.00 © 2013 IEEE
DOI 10.1109/SASOW.2013.16

In the ASCENS cloud computing case study, we will be concerned with a Platform-as-a-Service (PaaS) solution. The goal of the case study is providing a software system (called the Science Cloud Platform, SCP) which will, installed on multiple virtual or non-virtual machines, form a cloud providing a platform for application execution (these applications in turn providing SaaS solutions). The applications running on top of the platform are assumed to have requirements similar to Service Level Agreements (SLAs), which includes where they can and want to be run (regarding CPU speed, available memory, or even closeness in network terms such as latency to other applications or nodes).

B. Voluntary Computing

The second area is *voluntary computing*. This term usually refers to solutions in which individuals (consumers) offer part of their computing power to take part in a larger computing effort. The classic examples are the *@home* programs, of which SETI@Home [5] where personal computers are used in the search for extra-terrestrial intelligence is probably the most famous. Usually, voluntary computing is focused on computation; it depends on an agency which provides a centralized infrastructure into which people may plug-in, get their data from, perform calculations, and report back.

In the ASCENS cloud computing case study, we will adopt the voluntary computing approach insofar as we imagine individual entities (which includes natural persons, but universities as well) to voluntarily provide computing power in the form of cloud nodes which they can add or remove at any time as they see fit; i.e. nodes can come and go without warning, and their load may change outside of cloud concerns. They may include vastly different hardware, which includes CPU speed, available memory, and also specialized hardware as for example graphics processing chips.

C. Peer-to-Peer Computing

Finally, the last area is *peer-to-peer computing*. First popularized in the infamous area of file sharing, the basic idea of peer-to-peer computing is the lack of a centralized structure. There is no single node in the network on which the functionality of the overall system depends; rather, a decentralized communication approach is used which ideally is stable through the process of nodes coming and going, and offers no single point of failure, or single point of attack.

The ASCENS cloud computing case study is based on this idea; i.e. there is no centralized component in this cloud and nodes have to use some protocol to agree, in a decentralized manner, on where and what to execute. As already discussed above in the voluntary computing part, nodes may thus come and go without having to inform a central entity.

D. Bringing it all Together

Thus, all in all, we have a voluntary, peer-to-peer based platform-as-a-service solution. Such an infrastructure requires autonomic nodes which are (self-)aware of changes in load (either from cloud applications or from applications external to the cloud) and of the network structure (i.e. nodes coming and going) which requires self-healing properties (network resilience). Another issue is data redundancy in case nodes drop out of the system, which requires preparatory actions. Finally, executing applications in such an environment requires a fail-over solution, i.e. self-adaptation of the cloud to provide what we may call application execution resilience.

It is not necessary in this context to prevent participation of partially centrally-controlled entities such as IaaS providers. In fact, parts of the SCP may run on IaaS solutions which enables it to spawn new virtual machines or shut them down again. Such additional functionality can be used to balance load or to conserve energy.

To sum up in one sentence, the goal of the SCP is *to deploy and run user-defined applications on the p2p-connected web of voluntarily provided machines which form the cloud*.

III. Adaptation in the Cloud

A common approach to understanding, categorizing, and designing IT systems is the use of patterns, i.e. descriptions of characteristics which have proven to be beneficial for the implementation of a system. Within ASCENS, a catalog of architectural design patterns has been developed [6] which are intended to be used to build adaptive components and systems.

The design patterns have been studied with regard to the cloud case study [7]. In this section, we will discuss two patterns which have been used in the cloud.

Firstly, we need to discuss individual cloud nodes (which we call SCPis, for Science Cloud Platform instances). In this regard, the *proactive service component pattern* [7] best captures the behavior of such a node. This pattern enables the SCPi, which is a *Service Component* (SC) in the terms of ASCENS and the adaptation pattern itself, to have an internal feedback loop, or, in other words, implicitly contain an *Autonomic Manager* (AM) which is responsible for driving the adaptation through this feedback loop. These kinds of components are used because nodes in the cloud are goal-oriented in nature and actively try to adapt their behavior, even without an external call (e.g. for saving energy). A visualization of such a component is shown in Figure 1.

In the cloud, one such node uses its sensor to read environmental values such as CPU speed, current load, etc.; effectors may be used to configure an IaaS solution. Inputs and outputs refer to a user interacting with deployed applications. The control and emitter ports are used for ensemble adaptation (see below).

By using the proactive service component pattern, individual SCP nodes are self-aware and able to self-adapt, each following the goal of achieving best performance for deployed apps while saving energy. The internal feedback loop created through the AM part of the node is used for checking these conditions and adapting properly.

Furthermore, multiple nodes work together to execute applications. On this level, the *p2p negotiation service components ensemble pattern* [7] is a fitting description of this behavior, since each node (potentially) communicates with every other node for adaptation, there is no central coordinator, and each

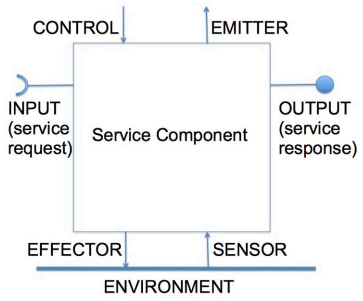

Figure 1. Proactive Service Component

node follows a goal (which in this case is the same for each node, though with different data depending on deployed apps). The use of this pattern is also possible because the components that form the ensemble are proactive and need to communicate with others to propagate adaptation. This is done, as indicated above, through the control and emitter interfaces of the service component.

Using this pattern, multiple SCP nodes work together: For each application, one ensemble consisting of a subset of the overall cloud nodes is formed which is then responsible for executing the application (which includes deployment, finding an executor, executing, and monitoring). We call such an ensemble an SCPe (Science Cloud Platform ensemble).

IV. Modeling Ensemble Behaviour

Modeling the behavior of the individual components and the ensembles which implement the cloud functionality is challenging due to the complexity and dynamics of the participating ensembles. In ASCENS, existing techniques such as component-based software engineering ([8], [9]) have thus been augmented with features that focus on the particular characteristics of ensembles. Among these are the fact that ensembles are dynamically formed on demand, realizing collective, goal-oriented behavior through communication between the individual participants; furthermore, multiple ensembles may run concurrently using the same basic resources, but dealing with different tasks on a higher level. To be able to model these issues on a first-class basis, the *Helena* approach [10] has been developed, which uses a UML-like notation for collaborations founded on a rigorous formal semantics.

A particular property of ensembles is the fact that although the platform on which ensembles run may itself be plain component-based, each component can take part in different ensembles and in the course of doing so take up different, ensemble-specific *roles* [11]. A service component may play different roles at the same time, both in one ensemble and in different, concurrently running ensembles; it may also dynamically change its role(s) in order to adapt to new situations.

The Helena approach is centered on this notion of roles and the collaboration of roles in ensembles for pursuing the ensemble goal. In the present case study, there may be multiple such ensembles; one for each of the applications which are executed within the cloud. Each ensemble has the goal of deploying the application, finding an execution target node, executing, and finally monitoring the application execution. This is illustrated in Figure 2.

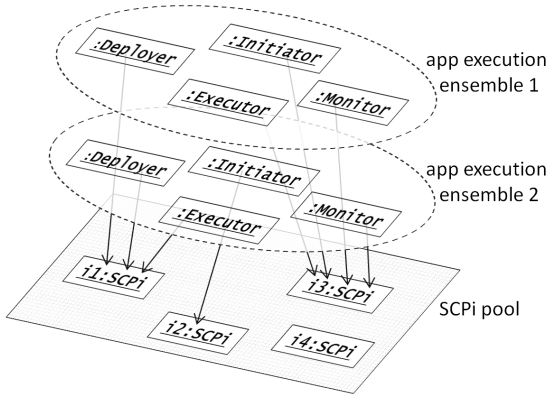

Figure 2. Ensembles in the *Helena* approach

The first or basic level (on the bottom of the figure) shows the pool of all SCPi nodes which are, in principle, able to provide resources to the cloud. In the figure, these are the four nodes labeled i1 to i4, which may be physical or virtual machines on which instances of the science cloud platform (SCPis) are running. Each of these may participate in ensembles for executing an application. As indicated in the figure, executing an application requires different responsibilities taken up by different roles in the ensemble. These are the deployer (node from which the application originates), the initiator (leading the search for an execution node), the actual executor, and a monitor which keeps tab on the executor. As an example, the figure shows two different ensembles, each executing one application, where nodes concurrently play different roles or do not participate at all.

Ongoing research in Helena currently focuses on the description of the behavior of each role as well as on the behavior on the ensemble level. These descriptions are given a rigorous formal foundation, which can then be exploited for ensuring that the ensemble behavior actually reaches the desired goal. We believe that the analysis of ensembles of collaborating roles can be beneficial to developers due to the reduction of the complexity of the models, since the combination of all roles within one service component must only be integrated into a component-based architecture in the following implementation phase. This is discussed in the next section, where a language is presented to which a systematic transition from Helena is currently being investigated.

V. System Specification in SCEL

ASCENS has been studying linguistic primitives suitable for the autonomic computing paradigm, and has developed the language SCEL (Software Component Ensemble Language) [12], [13] which is geared towards describing autonomic systems, taking into consideration the behaviors, knowledge, and aggregations involved, based on specified policies. SCEL in particular supports programming context-awareness, self-awareness, adaptation and ensemble-wide interactions.

In the following, we discuss the application of SCEL to the service components of the cloud case study. The concept of a service component – or autonomic component – lies at the heart of SCEL. This concept directly matches the notion of an SCPi, i.e. an individual node in the science cloud. Furthermore, the notion of an ensemble in SCEL matches the notion of an SCPe, since both are based on components' attributes, which in the science cloud usually take the form of participation in the management of a cloud application.

As an example, we consider here the SCEL implementation for a situation in the cloud where a node is overloaded, i.e. the CPU load exceeds a certain threshold and an application needs to be moved to a different node. This scenario includes the use of an IaaS solution, that is we include the ability to spawn a new virtual machine and moving the application there.

The full SCEL specification for the scenario of high load and moving an element to a newly created VM can be found in [13]. We will outline the general idea of the behavior here. The SCPi where the application is running initially is the SCEL component $\mathcal{I}[\mathcal{K}, \Pi, (AM[ME])]$. The interface $\mathcal{I}$ of the component encapsulates the remaining three elements. $\mathcal{K}$ represents the knowledge of the SCPi, which includes attributes relevant for adaptation. Π is the policy the component follows, which in this case is specified in SACPL, the SCEL Access Control Policy Language [13], discussed below. $AM[ME]$ is the (controlled) composition of processes AM and ME running in the component.

As we have seen in the section on adaptation patterns, an SCPi follows the proactive service component pattern. This means it contains, as in a SCEL component, internal knowledge and goals. In the above definition, the main work of the node, including the application logic, is performed in the Service Component (SC) which here is called Managed Element (ME). The component also contains its own, implicit, Adaptation Manager (AM), which specifies actions for adaptation (in particular, spawning a new machine).

The actual adaptation logic (i.e., when to adapt) is dealt with using the policy Π. The component's interface $\mathcal{I}$ exposes the attribute *CPULoad*, whose value (i.e., a percentage of load) is a context information sensed by the component from the underlying infrastructure. The policy Π detects when the attribute value is over a given threshold (e.g., 80%) and triggers the autonomic manager. More specifically, the policy says that the main application logic, which is part of ME, may only be performed as long as *CPULoad* is less than the threshold, while the spawning of a new machine (realized by means of an action **new** in AM) may not be performed until *CPULoad* is greater than the threshold.

An interesting problem in this context is that Π, ME and AM in a dynamically created VM are the same as those within the corresponding source node of the science cloud; however the application logic which is part of ME may only be executed on one machine at a time (since we assume that the application is a singleton). To ensure such behavior, multiple options have been explored with different power of expression. First, it is possible to add a new attribute to the component which keeps track of its execution status; AM is thus modified to properly set such an attribute. Second, the policy Π can be extended to include obligations that are actions executed as part of a node switch to take care of dealing with the execution status attribute (in place of AM). Finally, it is possible to use several policies instead of a single one, and dynamically switch between policies on an adaptation by means of a sort of automata where states are policies and state transitions represent adaptivity events (expressed as policy targets). The details of these three options are discussed in [13].

To summarize, the above description has shown the use of SCEL and a policy language, SACPL, to model a scenario within the science cloud where high load of a node leads to the spawning of a new virtual machine with an additional SCPi which can take over the application logic. An implementation of these abstract descriptions can be done in Java (as discussed in the following chapter) or more directly in jRESP [13], which is currently work in progress.

VI. Implementing the Science Cloud Platform

As identified in the previous sections, the cloud system will need to be implemented in a peer-to-peer manner with a heavy focus on being aware of changes in the available nodes and the load of each node. There are obviously multiple options of implementing such a system, and we are experimenting with several of them. Here, we are reporting on an implementation which is based on the existing peer-to-peer substrate Pastry [14] and accompanying protocols as well as an interpretation of the ContractNET protocol [15] used for the decision process on application execution.

The implementation is split into three layers: A network layer, which implements routing and message passing along with network self-healing properties; a data layer which handles data storage, including redundancy, and an application layer, which handles execution and fail-over of applications.

On the *network level*, the nodes which form the science cloud need to know about one another and be able to pass messages, either to single nodes (unicast), a group of nodes (multi- or anycast), or all nodes (broadcast). Given that the network can potentially become large, it is advisable that not all nodes need to know all other nodes. Furthermore, routing needs to be stable under adverse conditions (i.e. nodes that are part of the science cloud leave, or new nodes are added).

We use the existing protocol Pastry [14] in the form of the FreePastry implementation [16] as the basis of this layer, extended with the SCRIBE protocol [17] for any- and broadcast purposes. The inner workings of Pastry are similar to that of classic Distributed Hash Tables (DHTs), that is, each node is assigned a unique hash and nodes are basically organized in a ring structure, with appropriate shortcuts for faster routing. The protocol has built-in network resilience (self-healing). Efforts are under way to verify these properties formally [18].

The second layer handles *data*. When an application is deployed, the code needs to be available to all nodes which can possibly execute it; furthermore, application data needs to be stored in such a way that resuming an application, after a node which ran it failed, is possible. We thus need data storage with data redundancy, not only of immutable data (application code)

but also of mutable data (application data). Data is handled on top of Pastry using gcPAST, which is an implementation of the PAST protocol [19] with support for mutable data. PAST basically implements a DHT and includes a data redundancy mechanism which works by keeping k copies of a data package in the nodes surrounding the primary storage node (which is the one the data package hash is closest to).

The final layer, and the one implementing the actual platform-as-a-service idea, is the *application layer*. Applications can only run on some machines (based on requirements) so these must be found in the network. Every user of the cloud runs (at least) one instance of an SCPi and uses this instance both for deploying and using applications.

Deploying an application first means simply storing the executable code (as an OSGi bundle), which is based on the primary storage node idea introduced above. The primary storage node assumes the role of the initiator in the Contract-NET protocol [15] and uses a SCRIBE-based communication channel to request bids for execution. The request for bids includes the requirements of the application extracted from the stored bundle. Bids received back are evaluated and an executor node is selected.

While the executor runs the application, the initiator switches to a monitoring mode to ensure application availability on a regular basis. If the executor itself detects that it can no longer execute an application (for example, due to high load), it informs the initiator which initiates a new bidding process. The same applies if the executor node goes down, which is detected by regular checks from the initiator. If the initiator itself goes down, the hash-based node and data identification automatically leads to a new nearest node and thus initiator.

The SCP implementation is open-source and available from the ASCENS website [4].

VII. Mobile Cloud Computing

An interesting aspect of the case study is the fact that the individual nodes can be personal computers. As such, the concept also includes mobile nodes: laptops, tablets, or even smartphones. Mobile devices have some noteworthy properties in addition to standard nodes. They are devices (a) whose neighbors – in the sense of network proximity – may change, (b) whose battery capacity is limited, and (c) whose computing capacity may be (severely) limited as well.

Applications running on top of the science cloud may want to take those properties into consideration. In fact, we can imagine that applications intended to run on mobile devices be effectively split into two components, or smaller applications, communicating with one another. In one scenario, they may both run on one SCPi — if the node is powerful enough and access to power is not an issue; in another, they may be split between two SCPis, one on a mobile node (which handles UI) and another on a stationary node (which handles the computationally extensive background work). In order to keep the user interface responsive, the network latency between the two nodes may not exceed a certain threshold, which becomes problematic in the presence of (physical) node mobility.

This scenario has been investigated within ASCENS [20]. The envisioned method for this case uses a specialized adaptation architecture which, through two components, takes care of the planning and monitoring involved.

The first component involved is the *monitor*, which works within an application and can operate in one of two modes:

Running mode. In running mode, the monitor executes as part of a running application, i.e. it reflects the actual deployment. The monitor gathers data about the current node, which includes the performance and battery life. This non-functional propeties data (*NFPData*) is used by the planner (see below) to decide on adaptation.

Mock mode. A monitor may also be detached from its application and spawned on a different node where it runs in mock mode, testing the performance of the node with the performance model of the application (*MonitorDef*) in mind, but without actually moving the whole application. Again, NFPData is generated which can be used by the planner.

The second component is the *planner*. The planner provides the SCPi with the MonitorDefs for the monitors involved, which the SCPi can distribute to interesting nodes for gathering NFPData. Based on information about the application, which are included in a deployment plan, the planner is able to restrict which nodes are interesting; for example, this may include nodes which are a limit of two hops away. Based on the information in the NFPData from affected nodes, the planner instructs the underlying SCPi(s) to deploy the applications appropriately given the data.

A particular advantage of the monitor approach with mock modes is the availability of real data: The monitor deployed on remote nodes is able to report, based on its MonitorDef, precisely those measurements which are relevant for the application. As usual, the nodes which may take part in the execution of an application form an ensemble with the specific task to figure out the best configuration for all entities involved.

All in all, the adaptation architecture based on planners and (mock) monitors allows for a very flexible awareness of the network environment. While this approach is useful for all kinds of nodes the SCP may run on, it is particularly helpful in the presence of mobile nodes.

VIII. Related Work

The topic of this paper is a vision of a peer-to-peer, voluntary-computing-based cloud platform-as-a-service. Taken individually, the related work in these three areas has traditionally focused on a) routing and distributed storage of data (p2p), b) distributing workload from a central server (voluntary computing), and c) provisioning resources inside centralized data centers (cloud computing). Combining these areas has started to attract attention in recent years; we believe however that this research is far from concluded.

Voluntary clouds have been identified as a recent research trend in a state-of-the-art survey [21] in 2010. Another survey-type paper [22] by Panzieri et al. from 2011 also lists implementing cloud implementation on top of P2P networks as an open problem, and observes the usually centralized nature of voluntary computing. Panzieri et al. list the work by Babaoglu et al. from 2006 [23] as the first proposal for a "fully decentralized P2P cloud". The group has since followed

up with additional works, of which a very interesting recent one is [24] from 2012, which implements a similar system to the one presented here on the infrastructure-as-a-service level.

An approach to bridge volunteer and cloud computing, but without going for a fully decentralized organization, is the work by Cunsolo et el. [25] in 2009. There, the idea is for users to contribute additional resources to certain centralized components. Also in 2009, Chandra and Weissman [26] have come up with the term *Nebulas* instead of clouds for distributed voluntary resource use. They list three requirements for such systems, which we have addressed partially in this paper.

IX. Conclusions and Future Work

In this paper, we have explored the idea of an autonomic cloud in the sense of a voluntary computing, peer-to-peer based platform-as-a-service infrastructure which uses self-awareness and self-adaptation as the main ingredients for managing the execution of arbitrary applications.

We have shown several methods from the ASCENS project which can be helpful for discussing, modeling, and implementing such a system. Many aspects of this vision still require further research. In particular, we are interested in further exploring self-adaptation performance in the cloud, perform large-scale tests, explore alternative implementation models, and gather feedback on the methods discussed here.

Acknowledgements

This work has been partially sponsored by the EU project ASCENS, FP7 257414. We thank all partners who have contributed to the cloud case study.

References

[1] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani, "On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles," in *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*. IEEE, 2011, pp. 108–113.

[3] M. Wirsing, M. Hölzl, M. Tribastone, and F. Zambonelli, "ASCENS: Engineering Autonomic Service-Component Ensembles," in *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011*, ser. LNCS, B. Beckert, F. Damiani, M. Bonsangue, and F. de Boer, Eds. Springer, 2012.

[4] "The ASCENS Project." [Online]. Available: http://www.ascens-ist.eu

[5] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "Seti@home-massively distributed computing for seti," *Computing in Science and Engineering*, vol. 3, no. 1, pp. 78–83, 2001.

[6] G. Cabri, M. Puviani, and F. Zambonelli, "Towards a Taxonomy of Adaptive Agent-based Collaboration Patterns for Autonomic Service Ensembles," in *Proc. of CTS*. IEEE, May 2011, pp. 508–515.

[7] M. Puviani and R. Frei, "Self-management for cloud computing," in *SAI Conference, London, UK*, 2013.

[8] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2002.

[9] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, Eds., *The Common Component Modeling Example: Comparing Software Component Models*, ser. LNCS, vol. 5153. Springer, 2008.

[10] A. Klarl and R. Hennicker, "Foundations for Ensemble Modeling - The Helena Approach," *Submitted*, 2013.

[11] G. Gottlob, M. Schrefl, and B. Röck, "Extending object-oriented systems with roles," *ACM Trans. Inf. Syst.*, vol. 14, no. 3, pp. 268–296, Jul. 1996.

[12] R. Nicola, G. Ferrari, M. Loreti, and R. Pugliese, "A language-based approach to autonomic computing," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, B. Beckert, F. Damiani, F. Boer, and M. Bonsangue, Eds. Springer Berlin Heidelberg, 2013, vol. 7542, pp. 25–48.

[13] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "SCEL: a Language for Autonomic Computing," IMT Lucca, Tech. Rep., January 2013. [Online]. Available: http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf

[14] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, ser. Middleware '01. London, UK, UK: Springer-Verlag, 2001, pp. 329–350.

[15] Foundation for Intelligent Physical Agents, "FIPA Contract Net Interaction Protocol Specification," http://www.fipa.org/specs/fipa00029/SC00029H.html, March 2013.

[16] P. Druschel, A. Haeberlen, J. Hoye, S. Iyer, A. Mislove, A. Nandi, A. Post, A. Singh, M. Castro, M. Costa, A.-M. Kermarrec, A. Rowstron, S. Iyer, D. Wallach, Y. C. Hu, M. Jones, M. Theimer, A. Wolman, and R. Mahajan, "FreePastry," http://www.freepastry.org/, March 2013.

[17] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *Selected Areas in Communications, IEEE Journal on*, vol. 20, no. 8, pp. 1489–1499, 2002.

[18] T. Lu, S. Merz, and C. Weidenbach, "Towards verification of the pastry protocol using tla+," in *Formal Techniques for Distributed Systems*. Springer, 2011, pp. 244–258.

[19] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 188–201.

[20] L. Bulej, T. Burea, V. Horký, and J. Keznikl, "Adaptive deployment in ad-hoc systems using emergent component ensembles: vision paper," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, NY, USA: ACM, 2013, pp. 343–346.

[21] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *J. Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[22] F. Panzieri, Ö. Babaoglu, S. Ferretti, V. Ghini, and M. Marzolla, "Distributed computing in the 21st century: Some aspects of cloud computing," in *Dependable and Historic Computing*, ser. Lecture Notes in Computer Science, C. B. Jones and J. L. Lloyd, Eds., vol. 6875. Springer, 2011, pp. 393–412.

[23] Ö. Babaoglu, M. Jelasity, A.-M. Kermarrec, A. Montresor, and M. van Steen, "Managing clouds: a case for a fresh look at large unreliable dynamic networks," *Operating Systems Review*, vol. 40, no. 3, pp. 9–13, 2006.

[24] Ö. Babaoglu, M. Marzolla, and M. Tamburini, "Design and implementation of a p2p cloud system," in *SAC*, S. Ossowski and P. Lecca, Eds. ACM, 2012, pp. 412–417.

[25] V. D. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa, "Cloud@home: Bridging the gap between volunteer and cloud computing," in *ICIC (1)*, ser. Lecture Notes in Computer Science, D.-S. Huang, K.-H. Jo, H.-H. Lee, H.-J. Kang, and V. Bevilacqua, Eds., vol. 5754. Springer, 2009, pp. 423–432.

[26] A. Chandra and J. Weissman, "Nebulas: using distributed voluntary resources to build clouds," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009.