Engineering Distributed Adaptive Systems using Components

Keznikl, Jaroslav
2012

Engineering Distributed Adaptive Systems Using Components

Post-Graduate Student:
MGR. JAROSLAV KEZNIKL
Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ
keznikl@cs.cas.cz

Supervisor:
RNDR. TOMÁŠ BUREŠ, PH.D.
Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ
bures@cs.cas.cz

Field of Study:
Software Systems

The work was partially supported by the EU project ASCENS 257414, the Grant Agency of the Czech Republic project P202/11/0312. The work was partially supported by Charles University institutional funding SVV-2012-265312.

Abstract

One of the major issues in the domain of dynamically evolving distributed systems composed of autonomous and (self-) adaptive components is the task of systematically addressing the design complexity of their communication and composition. This is caused mainly by the inherent dynamism of such systems, where components may appear and disappear without anticipation. Addressing this issue, we employ separation of concerns by introducing a mechanism of implicit communication over implicit bindings, enabling components to dynamically form implicitly interacting groups – ensembles. Specifically, we present the DEECo component model, which based on this mechanism.

1. Introduction

Traditional software engineering methodologies together with related programming paradigms have long been guiding the procedure of building software systems through the requirements and design phase to testing and deployment. In particular, engineering paradigms based on the notion of components [1] have gained a lot of popularity as they support separation of concerns – extremely valuable when dealing with systems of high complexity.

It seems, though, that these traditional methodologies and paradigms are not sufficient when exploited in the domain of continuously changing, massively distributed and dynamic systems, such as the ones we explore in the ASCENS project [2]. These systems need to adjust to changes in their architecture and environment seamlessly or, even better, acknowledge the absence of absolute certainty over their (constantly changing) architecture and environment. An appealing research direction seems to be the decomposition of such systems into components able to operate upon temporary and volatile information in an autonomous [3] and self-adaptive fashion [4]. From the software engineering perspective, two main challenges arise:

- What are the correct *low-level abstractions* (models, resp. paradigms) that will allow for separation of concerns?
- How can we devise a systematic approach for *designing* such systems, exploiting the above abstractions?

In response, we propose the *DEECo component model* (stands for Dependable Emergent Ensembles of Components) [5]. The goal of the component model is to allow for designing systems consisting of autonomous, self-aware, and adaptable components, which are implicitly organized in groups called *ensembles*. To this end, we propose a slightly different way of perceiving a component; i.e., as a self-aware unit of computation, relying solely on its local data that are subject to external modification during the execution time. The whole communication process relies on automatic data exchange among components, entirely externalized and automated within the DEECo runtime framework. This way, the components have to be programmed as autonomous units, without relying on whether/how the distributed communication is performed, which makes them very robust and suitable for rapidly-changing environments.

The rest of the paper is organized as follows. In Section 2 the main concepts of the DEECo component model are presented. Section 3 evaluates the presented concepts by giving an example based on the ASCENS

cloud case study. Section 4 discusses the related work, while Section 5 concludes the paper and presents future work ideas.

2. DEECo Component Model

DEECo is based on two concepts: *component* and *ensemble*. Stemming from the ASCENS project, these concepts closely reflect fundamentals of the SCEL specification language [6] and are in detail elaborated in the rest of this section.

2.1. Component

A component is an autonomous unit of deployment and computation. Similar to SCEL, it consists of:

- Knowledge
- Processes

Knowledge contains all the data and functions of the component. It is a hierarchical data structure mapping identifiers to (potentially structured) values. Values are either statically typed data or functions. Thus DEECo employs statically-typed data and functions as first-class entities. We assume pure functions without side effects.

Processes, each of them being essentially a "thread", operate upon the knowledge of the component. A process employs a function from the knowledge of the component to perform its task. As any function is assumed to have no side effects, a process defines mapping of the knowledge to the actual parameters of the employed function (*input knowledge*), as well as mapping of the return value back to the knowledge (*output knowledge*). A process can be either periodic or triggered. A process can be triggered when its input knowledge changes or when a given condition on the component's knowledge (*guard*) is satisfied.

2.2. Component Composition

In DEECo, component composition is captured by means of ensembles. Composition is flat, expressed implicitly via a dynamic involvement in an ensemble. An ensemble consists of multiple member components and a single coordinator component. The only allowed form of communication among components is communication between a member and the coordinator in an ensemble. This allows the coordinator to apply various communication policies.

Thus, an ensemble is described pair-wise, defining the couples coordinator – member. An ensemble definition consists of:

- Required interface of the coordinator and a member
- Membership function
- Mapping function

Interface is a structural prescription for a view on a part of the component's knowledge. An interface is associated with a component's knowledge by means of *duck typing*; i.e., if the component's knowledge has the structure prescribed by the interface, then the component reifies the interface. In other words, an interface represents a partial view on a component's knowledge.

Membership function declaratively expresses the condition, under which two components represent the pair coordinator-member of an ensemble. The condition is defined upon the knowledge of the components. In the situation where a component satisfies the membership functions of multiple ensembles, we envision a mechanism for deciding whether all or only a subset of the candidate ensembles should be applied. Currently, we employ a simple mechanism of a partial order over the ensembles for this purpose (the "maximal" ensemble of the comparable ones is selected, the ensembles which are incomparable are applied simultaneously).

Mapping function expresses the implicit distributed knowledge exchange between the coordinator and a member of an ensemble. It ensures that the relevant changes in knowledge of one component get propagated to the other component. However, it is up to the DEECo runtime framework when/how often the mapping function is invoked. We assume a separate mapping for each of the directions coordinator-member, member-coordinator.

The important idea is that the components do not perceive the existence of ensembles (including their membership in an ensemble). They operate only upon their own local knowledge, which might get implicitly updated by the DEECo runtime framework whenever the component is part of an ensemble.

2.3. Execution Model

The DEECo execution model is based on asynchronous knowledge exchange and process execution, stemming from the asynchronous nature of the target dynamic distributed systems. Specifically, the component processes execute in parallel as independent threads either periodically, when triggered by modification of (a part of) their input knowledge, or whenever the process guard is satisfied. Similarly, a component binding of component forming an ensemble is accomplished by a separate

activity, evaluating the mapping function (again either periodically or when triggered).

Due to the asynchrony, it is necessary to ensure that knowledge is accessed consistently. Thus, at its start, a process is atomically provided with a copy of its input knowledge so that its computation is not affected by later-occurring knowledge modifications. When finishing, the process atomically updates its output knowledge. The same atomic copy-on-start and update-on-return semantics also applies to the membership and mapping functions of ensembles. Technically, this semantics can be implemented for instance via messaging.

Consequently, based on the computational model, an ensemble is created when the ensemble condition starts to hold, and is discarded when the condition gets violated. Technically, as the whole system is asynchronous and potentially distributed, techniques for handling inherent delays, while creating/discarding ensembles, have to be carefully chosen.

3. Evaluation

To evaluate and illustrate the above-described concepts, we'll give an example from the Science Cloud case-study [12]. In this scenario, several interconnected heterogeneous network nodes (execution nodes, storage nodes) run a cloud platform, on which 3rd-party services are being executed. Moreover, the nodes can dynamically enter/leave the network. Provided an external mechanism for migrating a service from one (execution) node to another, the goal is to "cooperatively distribute the load of the overloaded (execution) nodes in the network".

3.1. Solution in a Nutshell

Before describing the solution in DEECo concepts, we will give an outline of the final result. Basically, for the purpose of this evaluation, we consider a simple solution, where each of the nodes tracks its own load and if the load is higher than a fixed threshold, it selects a set of services to be migrated out. Consequently, all the nodes with low-enough load (determined by another fixed threshold) are given information about the services selected for migration, pick some of them and migrate them in using the external migration mechanism.

The challenge here is to decide, which of the nodes the service information should be given to and when, since the nodes join and leave the network dynamically. In DEECo, this is solved by describing such a node interaction declaratively, so that it can be carried out in an automated way by the runtime framework when appropriate.

3.2. Realization in DEECo

Specifically, we first identify the components in the system and their internal knowledge. In this example, the components will be all the different nodes (execution/storage nodes) running the cloud platform (Figure 1). The inherent knowledge of execution nodes is their current `load`, information about running services (`serviceInfo`), etc. We expect an execution node component to have a process, which determines the services to be migrated in case of overload. Similarly, the inherent knowledge of the storage nodes is their current `capacity`, `filesystem`, etc.

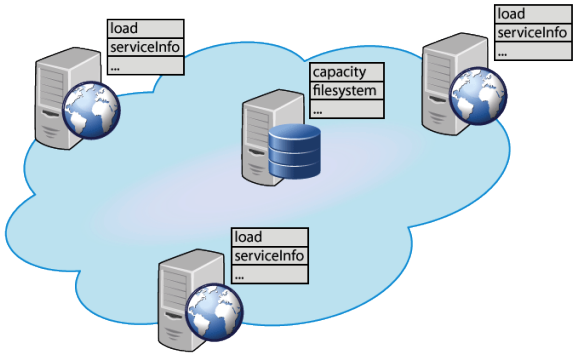

Figure 1: Components representing the cloud nodes and their inherent knowledge.

The second step is to define the actual component interaction and exchange of their knowledge. In this example, only the transfer of the information about services to be migrated from the overloaded nodes to the idle nodes is to be defined. The interaction is captured in a form of an ensemble definition (Figure 2), thus representing

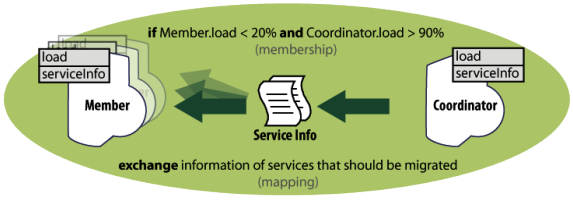

Figure 2: Definition of an ensemble that ensures exchange of the services to be migrated.

a "template" for interaction. Here, the coordinator, as well as the members, has to be an execution node providing the `load` and `serviceInfo` knowledge entries.

Having such nodes, whenever the (potential) coordinator has the load above 90% and a (potential) member has the load below 20% (i.e., the membership function returns true), the ensemble is established and its mapping function is executed (possibly in a periodic manner). The mapping function in this case ensures exchanging the information about the services to be migrated from the coordinator to the members of the ensemble.

When applied to the current state of the components in the system, an ensemble – established according to the above-described definition – ensures an exchange of the service-to-be-migrated information among exactly the pairs of components meeting the membership condition of the ensemble (Figure 3).

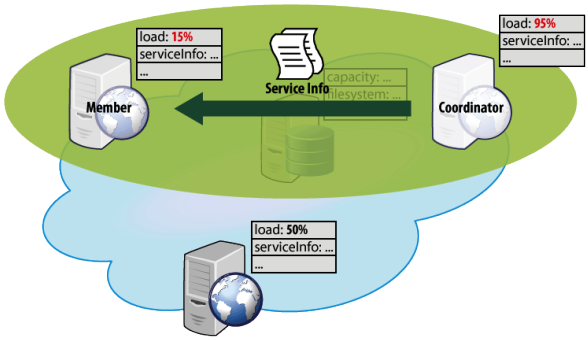

Figure 3: Application of the ensemble definition from Figure 2 to the components from Figure 1.

According to the exchange of service information, the member nodes then individually perform service migration via the external (i.e., outside of DEECo) migration mechanism.

3.3. Runtime Framework Implementation

Currently, we work on a prototype of the DEECo runtime framework implementation, based on distributed tuple spaces and implemented in Java. The sources, as well as documentation and examples, can be found at https://github.com/d3scomp/JDEECo.

4. Related Work

The task of achieving autonomy and (self-) adaptation has been partially addressed by agent-based approaches [7, 8], where actors leveraging on messaging establish explicit bindings for data and code exchange. Jade [7] is a complete framework for building, running and managing distributed multi-agent systems. Similarly, X-Klaim [8] is a complete framework based on a domain-specific language for capturing agent-based systems, where both data and processes are subjects to mobility. In general, agent-based frameworks themselves do not provide any higher abstractions for implicit grouping of components; however, due to their relative maturity, such frameworks could represent a suitable middleware for implementing the knowledge exchange in the DEECo runtime framework (as a replacement of distributed tuple spaces).

As for coping with dynamism of component bindings, techniques utilizing implicit bindings while focusing on explicit communication have been proposed. In iPojo [9] – a component system built upon the Felix OSGi implementation – each component binding is determined by a declarative specification associated with component interfaces. In [10], the idea of agent self-organization based on declarative conditions is presented. Specifically, the agents organize themselves into groups according to their spatial distribution and reorganization rules, communicating explicitly via a shared tuple space.

Finally, separation of concerns was to some extent achieved by introducing implicit communication (driven by a third-party entity) [11]. However, the communication is usually carried-out via explicit bindings.

5. Conclusion and Future Work

We assume that DEECo will be employed in the design of systems of autonomous self-adaptive components, such as a self-managing cloud platform and self-organizing car sharing [12], where it aims at simplifying the design process. Specifically, we expect DEECo to effectively handle knowledge exchange among the components, emphasizing separation of concerns. Although similar to software connectors [13], DEECo ensembles capture component composition implicitly and thus allow for handling of dynamic changes in an automated way. Similar benefits result from the implicit knowledge exchange.

We envision that the component model outlined here will serve as the basis for a design methodology that will exploit the presented abstractions and help in building long-lasting systems of autonomous components and component ensembles. Further, in order to support controlled architecture evolution, we aim to incorporate mechanisms for dynamic addition, modification, and removal of ensemble prescriptions. In addition, we envision supporting formal verification of DEECo applications. As for model checking of temporal properties, we assume a mapping of applications to SCEL [6] and intend to exploit its means [14] for this purpose. Moreover, we anticipate also employing stochastic model

checking [15, 16] for quantitative verification. Finally, inspired by the cloud and e-mobility case studies, we intend to introduce, in addition to abstractions for performance awareness, other forms of implicit knowledge-based communication such as distributed consensus.

References

[1] C. Szyperski, "Component Software: Beyond Object-Oriented Programming" (2nd Edition) (Hardcover), Addison-Wesley Professional, 2002.

[2] ASCENS [Online], http://www.ascens-ist.eu.

[3] J. O. Kephart, and D. M. Chess, "The vision of autonomic computing", *Computer*, vol. 36, IEEE CS, 2003, pp. 41–50.

[4] R. N. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation", *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2009)*, 2009, pp. 171–180.

[5] J. Keznikl, T. Bures, F. Plasil, and M. Kit, "Towards Dependable Emergent Ensembles of Components: The DEECo Component Model", *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2012)*, Aug, 2012.

[6] R. De Nicola, G. Ferrari, M. Loreti , and R.Pugliese, "Languages primitives for coordination, resource negotiation, and task description", ASCENS Deliv. D1.1, 2011, .

[7] F. Bellifemine, G. Caire, and D. Greenwood, "Developing multi-agent systems with Jade", John Wiley & Sons, 2007.

[8] E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi, "Modeling adaptation with a tuple-based coordination language", Proc. of *27th Symposium on Applied Computing (SAC 2012)*, 2012.

[9] C. Escoffier and R. S. Hall, "Dynamically adaptable applications with iPOJO service", *Software Composition*, 2007.

[10] C. Villalba, M. Mamei, and F. Zambonelli, "A self-organizing architecture for pervasive ecosystems", *Self-Organizing Architectures*, volume 6090 of LNCS, pp. 275–300, 2010.

[11] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP", Proc. of *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, 2006, pp. 3-12.

[12] N Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther, "Requirement specification and scenario description", ASCENS Deliv. D7.1, November 2011.

[13] R.N. Taylor, N. Medvidovic, and E.M. Dashofy: "Software architecture: foundations, theory, and practice", Wiley, 2010.

[14] L. Bettini et al., In global computing. Programming Environments, Languages, Security, and Analysis of Systems, volume 2874 of LNCS, 2003, pp. 88–150. "The Klaim project: theory and practice", *Global Computing: Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of LNCS, 2003, pp. 88–150.

[15] M. Z. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems", Proc. of *Tools and Algorithms for Construction and Analysis of Systems (TACAS 2010)*, Springer, 2010, pp. 23–37.

[16] J. Barnat, L. Brim, I. Cerna, M. Ceska, and J. Tumova: "ProbDiVinE, a parallel qualitative LTL model checker", *Quantitative Evaluation of Systems (QEST 07)*, IEEE, 2007.