Apostolos Papageorgiou, Tronje Kropp, Sebastian Ahlfeld, Stefan Schulte, Julian Eckert, Ralf Steinmetz: *Enhancing Availability with Self-Organization Extensions in a SOA Platform*. In: Proceedings of the Fifth International Conference on Internet and Web Applications and Services (ICIW 2010), p. 161-166, May 2010. ISBN 978-0-7695-4022-1.

Enhancing Availability with Self-Organization Extensions in a SOA Platform

Apostolos Papageorgiou, Tronje Krop, Sebastian Ahlfeld, Stefan Schulte, Julian Eckert, Ralf Steinmetz
Multimedia Communications Lab - KOM
Technische Universität Darmstadt
Darmstadt, Germany
apostolos.papageorgiou@kom.tu-darmstadt.de, tronje.kropp@kom.tu-darmstadt.de, ahlfeld@rbg.informatik.tu-darmstadt.de, stefan.schulte@kom.tu-darmstadt.de, julian.eckert@kom.tu-darmstadt.de, ralf.steinmetz@kom.tu-darmstadt.de

Abstract—The availability and reliability of Service-oriented architectures (SOA) depends on two factors: On the one hand, the availability and reliability of the services that provide a certain business functionality and on the other hand the services that make up the underlying SOA platform. For platforms that are supposed to form the core of mission-critical service-oriented applications, this implicates the need for mechanisms that can regulate the reliability- and availability-levels of the core services in changing conditions. In this paper, we discuss open questions about what kind of monitoring functionalities and service replication mechanisms should be integrated in SOA infrastructures. Therefore, the integration of concepts from peer-to-peer (P2P) computing is proposed: We present a self-organization extension that can improve the availability of the core services of SOA infrastructures, and we provide an experiment-based evaluation, showing some of the benefits that this extension can have in a critical scenario. The concepts are prototypically implemented as extensions of Apache Tuscany, which is a realization of the Service Component Architecture (SCA) standard.

I. Introduction

Along with their established advantages, such as high flexibility, extensibility, and interoperability [11], Service-oriented Architectures (SOA) are also expected to achieve availability levels that are at least as high as these of traditional solutions. Approaches that aim at improving the availability and reliability of SOA are usually built on the assumption that a number of service alternatives can be invoked ad hoc, if a service fails. These approaches use techniques like process replanning with dynamic service substitution ([3], [13]), or dynamic enforcement of governance guidelines [10], and are usually applied at the level of service consumption or business process execution. When the availability of the applications that use these techniques is measured, there is a highest boundary that can be achieved. It is the maximum availability level that the used service platform can support. This platform can vary from a simple enabling infrastructure, i.e., a simple service registry with any accompanying components, to a complex Enterprise Service Bus (ESB).

The problem is that current service platforms can support limited availability levels, because of vulnerabilities or single points-of-failure inside their core. Such a basic vulnerability, which we try to address with our approach, is the centralized access to functions of the domain and the deployment, i.e., centralized access to interfaces that are used for address resolution, dynamic launching of services, and more. Even if the services are available, the availability experienced by the user sinks if the machines that provide these interfaces under-perform. Similar problems exist with service registries and search functions. Furthermore, current solutions use monitoring approaches (cf. also Section 2) that cannot support quick enforcement of healing mechanisms, e.g., replication of overloaded services.

Some techniques, e.g., service replication, appeared in order to face such problems. These techniques have sometimes high costs and are applied only dynamically, if necessary. These techniques, as well as the decision-making that accompanies them, are supported by monitoring mechanisms. This monitoring-supported enforcement of such techniques, as well as related research, are normally positioned under the fields of *adaptation mechanisms* and *self-organization*. How this can be optimally applied on SOA infrastructures, is still unclear, and depends on the nature of the used platform. Different service platforms (e.g., ESBs) are used in different application domains, and each of them presents different challenges concerning its enrichment with adaptation or self-organization capabilities. This work presents a concept which, in its general form, could be used for such enrichment of many SOA platforms. The concept is then implemented as an extension of the Service Component Architecture (SCA [9]). The work is presented and evaluated on the state-of-the-art SCA platform, Apache Tuscany [1].

With this regard, the paper is outlined as follows: Section 2 examines the related work and states our contributions. Section 3 identifies some extra challenges that are present in our particular scenario. Sections 4 and 5 form the core of this paper by describing our solution and its evaluation results. As the concept could enhance different platforms, the description of the idea (4.1) will be as independent of the implementation as possible. We conclude the paper in Section 6.

II. Related Work And Contributions

We look into related work in three main directions, where we also identify and position the three partial contributions of our work. First, we look at the research towards third-generation, self-adapting service platforms. Second, we see attempts of enhancing service platforms by using peer-to-peer technologies. Last, we examine monitoring aspects of up-to-date service platforms.

Traditionally, there are two approaches for building SOA infrastructures: the *point-to-point integration* and the *hub-and-spoke* approach [11]. While the first is simpler and more static, the latter includes a service bus and/or other related middleware that dynamically undertakes the routing and addressing of the used services, and the support and transformation of the used protocols. Other functionalities can also be present, letting the hub-and-spoke approach be considered as more advanced and, in essence, as the successor of the point-to-point integration (cf. also [11]). Nevertheless, research in the field of SOA self-adaptation (e.g., [6] and [15]), lets us assume that we are heading for a third generation of SOA infrastructures, in which the service platform, i.e., the service bus with the accompanying middleware, will offer even more automation and further functionalities, namely integrated monitoring, adaptation mechanisms, and more. As the enrichment of service platforms presents different challenges and opportunities depending on the exact paradigm, we contribute in these attempts towards "third-generation" service platforms by presenting an idea of what these extensions should include, and by showing how it is implemented in the case of SCA.

Main intension of the adaptation mechanisms is to keep the QoS above certain limits. A recent survey [7] already placed peer-to-peer mechanisms among the most highly suitable solutions for the substrate of future service platforms that go in the direction of QoS-guarantee and self-adaptation. Approaches that use peer-to-peer mechanisms for the enhancement of service platforms have focused until now either on special-purpose service orchestration [4], or on service discovery and group collaboration [5]. Believing that the enablement of self-adaptation dictates that these mechanisms lie deeper inside the platform and support all or most of the functionalities of a service bus, we contribute by using peer-to-peer mechanisms to distribute the service bus and enhance the availability of the services of an SCA platform. Furthermore, unlike most of such new frameworks, we provide an evaluation scenario and some measurements to demonstrate the availability enhancement.

Aspects of our integrated platform monitoring can be seen as a further contribution of this work, given that almost all state-of-the-art monitoring components of service platforms are not integrated in the platform logic and cannot serve the goal of supporting self-adaptation optimally. Instead, they normally perform centrally-controlled measurements for hardware modules or service invocations. In the next sections it will be further clarified how this differs from our decentralized, event-based, adaptation-enabling platform monitoring approach. Strengthening our argument, we mention that almost all theoretical SOA Maturity Models (e.g., [12]) define 5 possible maturity levels for a SOA and they place the feature of event-based platform monitoring in the maturity level 4. Related studies (e.g., [2]) prove that almost no current SOAs achieve that maturity level, but they rather lie between levels 2 and 3.

III. Further Challenges Of Our Scenario

The purpose of our extended platform is to serve as the SOA substrate for our project [14], a project that supports the management of disastrous events. In such a scenario, the availability of the services not only needs to be high when the disaster occurs but it is also expected to be suddenly endangered, because of an "explosion" of the system usage at that point. This system usage pattern will be reflected in the test cases of our evaluation in Section 5. We list here how these challenges were translated to *technical challenges* for our platform:

- No single point-of-failure is acceptable for any critical core service.
- Control mechanisms must provide the possibility of defining different, application- or situation-dependent algorithms that determine the minimum number of instances of particular services. These algorithms will be designed based on the needed availability levels and the expected usage patterns.
- Consistent and detailed information about the running services is needed in order to provide enhanced control. This means that all services have to be registered with the same procedure before they are started, and there must be mechanisms that find out which services, and how many instances of them are registered / active, and on which nodes.

We have found no approach that addresses exactly our needs (cf. also Section 2). As for the implementation, the extensions that will be presented were necessary also because of the following *lacking capabilities*, which are absent from many platforms other than Apache Tuscany:

- The platform enables the development of distributed applications but it is almost impossible to distribute all the core modules in the way that our challenges dictate. Normally, the Tuscany domain and deployment service is centralized and it also lacks many of the desired capabilities and functionalities that we mentioned.
- There are no service monitoring mechanisms that could support self-organization or absolute control of service instances. The monitoring modules have another meaning and a different functionality than the one we will provide. This difference will be further explained in the next sections.
- There are no replication or maintenance mechanisms for the internal application services.

IV. Our Service Platform Availability Extensions

With regard to the described challenges, we present in this section a generally applicable idea of how they could be faced inside a service platform, and then we briefly describe

how we implemented most parts of the concept by modifying the Apache Tuscany service platform.

A. Concept

We define as *core parts* of the service platform those parts that are responsible for the main platform functionalities, as we mentioned them in Section 2 (registry mechanisms, address resolution, service deployment etc.). Our main idea was to re-define the core parts so that:

- They are distributed, consisting of many co-operating instances, supporting fault-tolerance in the classical p2p manner, i.e., being able to operate despite the unavailability of some instances.
- They offer an extended set of functionalities that enable self-organization/adaptation mechanisms and support the fulfillment of our availability requirements.
- All the extended functionalities are offered through the interfaces of a p2p overlay, so that no centralized parts of the service bus have to be addressed.

The choice of p2p is driven by our striving for fault-tolerance. The failure of peer nodes, where instances are running in order to provide core mechanisms, will now not mean that the mechanisms will not be available any more. At the same time, a flexible cooperation of the core part instances is needed. Few technologies can support this fault-tolerance and this cooperation as good as p2p. On this basis we designed a platform where all participating nodes, i.e., all providers/consumers of application-level services, can also carry instances of core parts, participating in a common p2p network that connects their core part instances (Fig. 1). We re-define, extend and distribute four core parts, while a lot of accompanying platform parts/functionalities are abstracted in our concept and "borrowed" from the used platform in our implementation. A description of these four core parts follows, focusing on the features that are normally absent in current solutions, like Apache Tuscany.

Our distributed *domain service* is addressable through the overlay (so that one instance of it may be enough) and offers the extended possibility of returning multiple endpoints to service-lookups. This will support the usage of service replicas that will be generated by our self-organization mechanisms, as well as a more reliable address resolution, given that any node may be able to perform this resolution. The service registry can also be seen as part of the domain service and it has the form of a distributed database with its entries being transparently and redundantly distributed among those nodes that carry domain service instances.

Our distributed *deployment service* enables the local or remote starting/stopping of services. It is assumed that the services save their resources when they are registered in the domain and that these resources are enough in order to start/replicate them on other nodes. Nodes also use the deployment service in order to register themselves as capable of hosting particular services.

Our distributed *system manager* takes care of pre-defined numbers of instances of other core services and offers additional interfaces for system information that is important to other core parts, especially to the platform monitor.

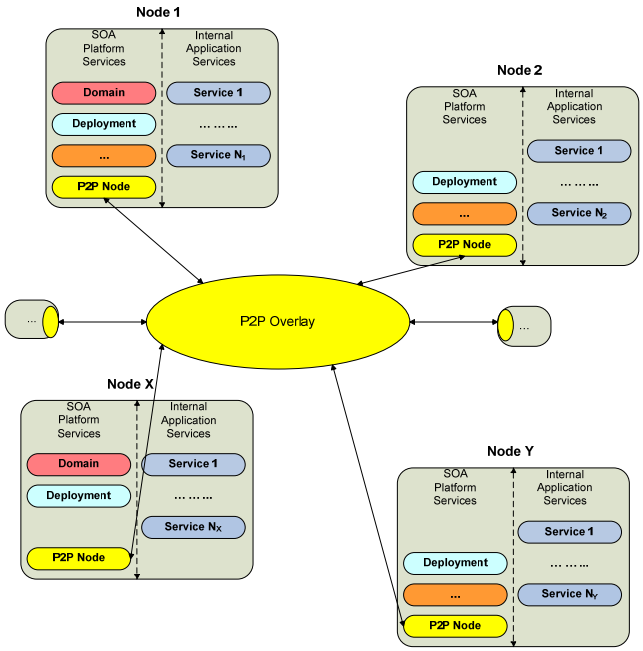

Figure 1. Overview of the p2p-based distribution of the core parts

Our distributed *platform monitor* has major differences from usual service monitoring components or tools. Its goal is to support adaptation, so it engages the event stream processing concept [8] and a push-approach for (developer-defined) monitoring events, rather than a database where simple observations are stored. Furthermore, it is integrated in the platform logic, so that no direct or indirect interaction with the monitored services or their "callers" is needed in order to gather information about the service calls. In the evaluation scenario, we will see an exemplary usage of the monitor that would not be achievable with other approaches.

B. Design and Implementation

All these conceptual extensions pose new challenges when it comes to their implementation as extensions of existing service platforms like Tuscany. For example, some features can be added "on-top" while others may present incompatibilities with existing mechanisms. We distinguish 3 approaches for enhancing the service platform with new features, which are generally valid when it comes to middleware enhancement:

- As *new platform modules*, i.e., developed and built additionally to the existing modules of the platform.
- As *external libraries*, which can be either special-purpose libraries, i.e., software developed for these extensions, or ready, possibly third-party, software.
- As *modifications in the core* of existing platform modules, when incompatibilities appear.

Before listing what we implemented in these three directions, we present in Fig. 2 a compact SCA representation of a node of our modified platform, providing a view of the interrelations of the core parts of the service platform, as well as their relation to the p2p overlay and the normal application components.

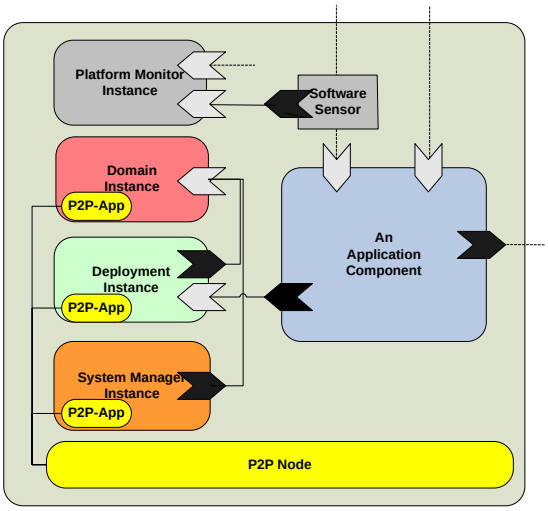

Figure 2. Component interrelations in a node of the modified platform

We had to define a new node type, the *CoreNode*, which merges an SCA node with a p2p node. While the extended domain, deployment, and system manager are based directly on the p2p node, the platform monitor is built on the (modified) service invocation mechanisms of the platform, enabling the binding of queries (posed by any monitoring component) to particular services, in order to retrieve the data that he needs about the corresponding service invocations. This is again compactly depicted in Fig. 3.

Further design and implementation details of our platform extensions, such as UML diagrams, are out of scope, while the API of each core part corresponds to the functionalities described in section IV.A. We give here just an overview of the implementation with regard to the three categories that we distinguished in this section:

- *New platform modules:* The module that defines the CoreNode and includes the implementations for the deployment and the system manager instances is the "distributed-core". It is implemented as a new module but depends on some core modifications, as well as on an external library for the p2p overlay. The "platform-monitor" is also a new module, also depending on core modifications and on an external library for the event stream processing.

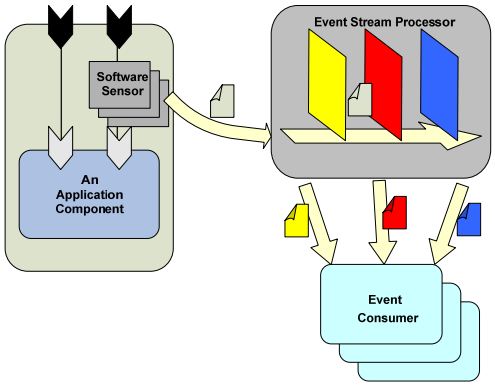

Figure 3. Adaptation-enabling event processing of the platform monitor

- *External libraries:* "freepastry" is used for the p2p overlay and "esper" for the event stream processing. Both are third-party, open-source libraries.
- *Core modifications:* The Tuscany module "core" was modified in order to implement our domain instances. Inside the "assembly" module of the core, we had to modify the runtime component implementation. Some other modules, e.g., the "java-runtime-implementation", also had to be modified in order to support dynamic invocation and other features needed by our modified platform.

V. EVALUATION

In order to evaluate our approach, we define a specific scenario that was related to our project, and compare our approach with a release version of the used platform. Of course, specific adaptation mechanisms should be compared to related approaches that could potentially enrich the same service platforms. Unfortunately, such general comparisons do not seem to be applicable at the moment, and remain subject of future work. Still, Apache Tuscany is a state-of-the-art SCA platform, and comparisons with it appear to be in our case more interesting than any other scenario.

The experiments that are based on our modified platform are such that as many new features as possible can be evaluated. Nevertheless, they are limited to include only some capabilities. We condense many functions into two main capabilities that we will use in our experiments. It is necessary to describe now these two capabilities:

- *Interest Registration*: Any component can register itself as "interested" in an SCA service, saving at the same time its queries, determining this way what kind of data the software sensors will be sending to it and when. Such components contain "actors", which enforce reactions under certain circumstances.
- *Service Instance Control Mechanism (SICM)*: The deployment instances offer to other components the possibility of retrieving the number of running instances of a particular SCA service, as well as the addresses of the nodes that could host further instances. The SICM builds on these capabilities and can be used by any component in order to define a minimum number of instances of a service that should be running. This "requirement" is saved, so that failures of hosting nodes lead to the starting of instances of the service on other candidate nodes.

A. Evaluation Scenario

Internal services of our application are expected to be suddenly invoked with an increasing frequency when a disaster occurs or later when the emergency level of the situation is set higher by the involved organizations. With this regard, we chose an example service, and implemented external clients that invoke it with the pattern shown in Fig. 4. There, we see also how a linear increase of users leads to an exponential increase of erroneous service invocations, i.e., to decreased availability levels. The test-clients record errors when no response is received or when a timeout is overridden. More details will be understood in section 5-B.

With $N_t(x)$ denoting the number of occurrences of x in the last t seconds, we define as *availability* of S for our scenario the value

$$A = \frac{N_{10}(\text{successful invocations of S})}{N_{10}(\text{invocations of S})} \times 100\%,$$

and we measure it over time for the following four experimental cases:

- *Exp1*: An instance of S is running on the Apache Tuscany release platform.
- *Exp2*: Three instances of S were running on the Apache Tuscany release platform and the invocations were equally distributed to them. The number of instances (3) was chosen empirically, so that it could almost always satisfy the given invocations' curve (Fig.4). For this case, as well as for the next two cases, the distribution of the invocations among the instances was simulated, and not automated. This is safe because the load balancing is irrelevant to the results that we present, though it would, of course, be interesting to test with different balancing of the invocations.
- *Exp3*: An instance of S is running on our extended platform, the deployment instance of a node (more nodes could be used for fail-safety) registers itself as interested in S, with a query for retrieving the number of users of S each second. The deployment instance (more precisely its "actor" upon the retrieved data) has the following simple logic: use the SICM to add an instance every time that the load of S exceeds a limit. This limit was chosen in our case so that, for the given input of fig. 4, the mechanism is started almost every minute.
- *Exp4*: As in *Exp3*, with the difference that the SICM now doubles the number of instances every time it is triggered. With these two different configurations, we show the flexibility of the freely defined adaptation logic, indicating how our framework can easily integrate application-dependent logic in order to be optimally exploited in different systems. Obviously, the choice of this logic affects the results.

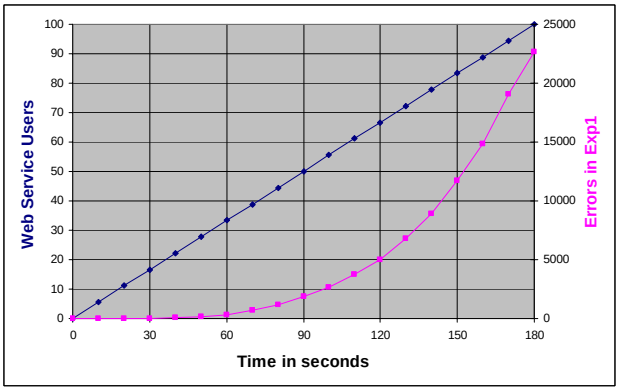

Figure 4. Experimental service invocation pattern

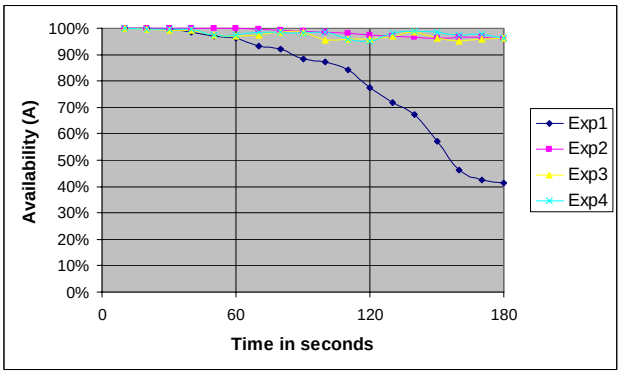

Figure 5. Measured availability

B. *Evaluation Results*

Fig. 5 and Fig. 6 present the evaluation results based on the four experiments that we described. Although the results have been obtained from an example service, which can be either an internal application service or a core platform service (e.g., an instance of the deployment service), it is obvious that this does not harm generality. Similar effects would be noticed for almost any service, maybe with a slightly modified invocation pattern. These evaluation results intend to show some enhancements of a platform in particular scenarios and are not to be seen as a direct and complete comparison. Furthermore, the results only show the benefits of the mechanisms described in 5-A, which are based on our extended concept. Further benefits of our solution that we described earlier and relate to the p2p-based fault-tolerance of the core parts are not included in these experiments and are not mirrored in the results.

The results for Exp1 prove that the availability of a service sinks when the number of users increases rapidly. The same effect is slightly noticeable even in the case of the second experiment that is based on the original Tuscany platform, namely Exp2, although the number of service instances was manually chosen in order to satisfy the given input. The decrease of the availability level is in that case slow and trivial, though steady. If the number of users would grow further, then the number of service instances would not be able to satisfy them any more, and an effect similar to that observed in the case of Exp1 would appear. Even if the maximum load that can be expected for a service is known from the beginning, excluding this way the possibility of such effects to appear, the usage of many instances from the beginning can lead to a big waste of resources. In scenarios like ours, where the service usage explosion is expected to happen suddenly but also rarely, this waste will be ongoing during most of the time.

Contrary to Exp1 and Exp2, the number of service instances during the experiments Exp3 and Exp4 is adapted to the service load, maintaining high availability levels without wasting resources. Fig. 6 shows the effect of service instance control. The component that uses the extended mechanisms in order to perform this control is (implicitly) informed – in this case every ca. 1 minute – by the platform monitor that the availability is sinking. Accordingly, further

service instances are deployed and the service invocations are again distributed among them. So, with an appropriate configuration at the side of the monitoring (and acting) component, the availability can be maintained at the wished levels, as long as this is allowed by the total resources that are available in the system. In a similar manner, the service instances can be adapted to a sinking number of users, though this is not shown with the present experiments.

During the last minute of the evaluation, Exp4 presents a higher availability, because the number of service instances is increased there more abruptly. With the difference between Exp3 and Exp4, we can understand the configurability of the used mechanisms. The fact that different logics can be used inside these mechanisms offers flexibility in the regulation of the availability levels, and their trade-off with costs. For example, a logic like the one used in Exp3 would be used in a scenario where service instance adaptations can be performed often, while the logic of Exp4 would rather be applied in scenarios where the frequent adaptation is either impossible or not desired.

VI. Conclusion

We presented a concept, along with its prototypical implementation and evaluation, for distributing the core parts of a service platform and enriching them with adaptation mechanisms in order to offer fault-tolerance and higher service availability. Concluding, we mention some limitations, which can be also seen as subject of future work.

First, security aspects become more critical, because of the further capabilities that simple nodes have now. Lack of control upon them is more dangerous when they carry platform instances than when they simply host applications services. Moreover, the complexity of the distributed implementation, as well as the fact that statefull services cannot be easily replicated or migrated, lead to some limitations concerning the applicability of our mechanisms.

Acknowledgment

This work is supported in part by the E-Finance Lab e. V. (www.efinancelab.de) and the BMBF-sponsored project SoKNOS (www.soknos.de). We would also like to thank Steffen Lortz for his participation in our implementation.

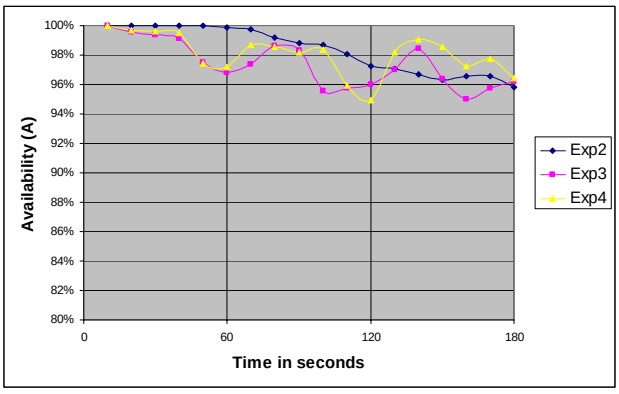

Figure 6. Adaptation effects

References

[1] Apache Tuscany project, Apache Software Foundation (ASF), “http://tuscany.apache.org/”, last updated on November 2009, last accessed on February 2010.

[2] M. Bachhuber, J. Eckert, A. Miede, and R. Steinmetz, “Readiness and Maturity of Service-oriented Architectures in the German Banking Industry – A Multi-Participant Case Study,” E-Finance Lab quarterly, vol. 4, November 2009, pp. 6-8.

[3] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, “Heuristics for QoS-aware Web Service Composition,” Proc. International Conference on Web Services (ICWS 2006), September 2006, pp. 72-82, doi:10.1109/ICWS.2006.69.

[4] W. B. Bradley and D. P. Maher, “The NEMO P2P service orchestration framework,” Proc. 37th Annual Hawaii International Conference on System Sciences (HICSS 2004), Track 9, vol. 9, page 90290.3, IEEE Computer Society, January 2004, ISBN:0-7695-2056-1.

[5] D. G. Galatopoulos, D. N. Kalofonos, and E. S. Manolakos, “A P2P SOA Enabling Group Collaboration through Service Composition,” Proc. Fifth International Conference on Pervasive Services (ICPS 2008), ACM Press, July 2008, pp. 111-120, doi:10.1145/1387269.1387289.

[6] E. Gjorven, R. Rouvoy, and F. Eliassen, “Cross-layer Self-adaptation of Service-oriented Architectures,” Proc. Of the third Workshop on Midleware for Service Oriented Computing (MW4SOC 2008), December 2008, pp. 37-42, doi:10.1145/1462802.1462809..

[7] V. Issarny, M. Caporuscio, and N. Georgantas, “A Perspective on the Future of Middleware-based Software Engineering,” IEEE International Conference on Software Engineering (ICSE 2007), Proc. Workshop on the Future of Software Engineering (FOSE 2007), IEEE Computer Society, May 2007, pp. 244-258, doi:10.1109/FOSE.2007.2.

[8] D. C. Luckham, “The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems,” Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN:0201727897.

[9] OASIS openCSA Specifications for the Service Component Architecture (SCA), “http://www.oasis-opencsa.org/sca”, last updated on August 2007, last accessed on February 2010.

[10] A. Papageorgiou, S. Schulte, D. Schuller, M. Niemann, N. Repp, and R. Steinmetz, “Governance of a Service-Oriented Architecture for Environmental and Public Security,” Proc. Fourth International ICSC Symposium on Information Technologies in Environmental Engineering (ITEE 2009), May 2009, pp. 39-52, doi:10.1007/978-3-540-88351-7_3.

[11] M. P. Papazoglou and W. J. Heuvel, “Service oriented architectures: approaches, technologies and research issues,” The VLDB Journal, vol. 16, No. 3, 2007, pp. 389-415, doi:10.1007/s00778-007-0044-3.

[12] C. Rathfelder and H. Groenda, “iSOAMM: An Independent SOA Maturity Model,” Proc. 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08), vol. 5053/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg , 2008, pp. 1-15, doi:10.1007/978-3-540-68642-2_1.

[13] D. Schuller, A. Papageorgiou, S. Schulte, J. Eckert, N. Repp, and R. Steinmetz, “Process Reliability in Service-Oriented Architectures,” Proc. Third IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2009), IEEE Computer Society, June 2009, pp. 640-645, ISBN 978-1-4244-2346-0.

[14] The SoKNOS project, “Service-oriented Architectures Supporting Networks of Public Security, “http://www.soknos.de”, last updated on October 2009, last accessed on February 2010.

[15] G. Tosi, G. Denaro, and M. Pezze, “Towards Autonomic Service-Oriented Applications,” International Journal of Autonomic Computing, vol. 1, issue 1, April 2009, pp. 58-80, ISSN:1741-8569