Enhancing Availability through Dynamic Monitoring and Management in a Self-Adaptive SOA Platform

Apostolos Papageorgiou, Tronje Krop, Sebastian Ahlfeld, Stefan Schulte, Julian Eckert, Ralf Steinmetz
Technische Universität Darmstadt
Multimedia Communications Lab - KOM
Darmstadt, Germany
apapa@kom.tu-darmstadt.de, tronje.krop@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 of Service-oriented Architectures (SOA) depends on two factors. These are firstly the availability of the services that provide a certain business functionality and, secondly, the availability of the components or 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 availability levels of the core services in changing conditions. In this paper, we handle open issues about the kind of monitoring functionalities and adaptation mechanisms that should be integrated in SOA infrastructures. In our proposed solution, we integrate concepts of event-based systems to enhance the dynamicity of the SOA platform monitoring, as well as concepts from peer-to-peer computing to achieve an efficient distribution of the SOA platform core. By prototypically implementing the concepts as extensions of Apache Tuscany, which is a realization of the Service Component Architecture standard, we show in an experiment-based evaluation how the availability of the core services of SOA infrastructures has been improved. Additionally, we explain further benefits that can be achieved with adaptation mechanisms other than replication, which are also enabled by our extensions.**

***Keywords*-Service Platform, Adaptation, SOA, Web Services, Availability**

I. Introduction

As has been also described in [1], the success of Service-oriented Architectures (SOA) is normally not credited to the strict technical features or Quality of Service (QoS) levels offered by the underlying technologies. However, along with their established advantages, such as high flexibility, extensibility, and interoperability [2], Service-oriented Architectures are now also expected to achieve performance and availability levels that are as high as these of traditional, platform-dependent solutions. Approaches that aim at improving the availability 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 (as in [3], [4], and [5]), or dynamic enforcement of governance guidelines [6], and are usually applied at the level of service consumption or business process execution. Still, when the availability of the applications that use these techniques is measured, there is an upper bound 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) [7].

The challenge 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 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 declines if the machines that provide these interfaces under-perform. Similar problems exist with service registries and search functions. Furthermore, current solutions use static monitoring approaches (cf. also Section II), which cannot support quick enforcement of healing mechanisms, e.g., replication of overloaded services.

Some techniques, such as Web service replication [8], appeared in order to solve some of the aforementioned problems. These techniques have sometimes high costs and must be supported by monitoring mechanisms and by an adequate decision logic. 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 has not been thoroughly examined from a technical perspective, 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, can be used for such enrichment of many different SOA platforms. Its main ideas are the distribution of the core parts of a SOA platform and the employment of event-based monitoring in the platform core for supporting self-adaptation. This general 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 [10].

With this regard, the paper is outlined as follows: Section II examines the related work and states our contributions. Section III identifies some additional challenges that are present in our particular scenario of a mission-critical SCA platform. Sections IV and V 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 (Subsection IV.A) will be as independent of the implementation as possible. Still, the detailed desription of the different service lifecycle phases (Subsection IV.C) sometimes needs to refer to implementation details in order to better support the reader's understanding. Section V presents the results of a well defined evaluation scenario with our extended Apache Tuscany platform and Section VI offers implementation-related examples of further adaptation mechanisms that can be integrated due to our extensions. Our conclusions and plans for future work are summarized in Section VII.

II. Related Work and Contributions

The decision to use concepts from peer-to-peer (p2p) computing and event-based systems were taken after a careful analysis of all the phases that a self-adaptive SOA platform has to go through. Therefore, the best way to present the related work is to explain where these concepts have been already used or proposed in order to enhance SOA platforms. This way we come to new ideas and propose their usage for further possible enhancements. So, 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.

In accordance to the nature of service-oriented software, some tasks exist, which must be fulfilled in almost all SOA solutions. The most important of them are:

- The service registry mechanisms (service advertisement, service look-up, etc.).
- The address resolution (mapping of name-based service calls to exact addresses/endpoints).
- The service deployment (loading, configuration, starting, and stopping of services).
- The management and monitoring, usually in the form of auditing and logging, with focus on QoS parameters such as service response times or hardware metrics such as CPU load.

Depending on the scale at which these tasks are automated or undertaken by middleware components, there are traditionally two approaches for building SOA infrastructures: the point-to-point integration and the hub-and-spoke approach [2]. While the first is simpler and more static, the latter includes a service bus and/or other related middleware that dynamically undertakes the aforementioned tasks, as well as their subtasks, such as the routing and addressing of the used services, or 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 [2]. Nevertheless, research in the field of SOA self-adaptation ([11], [12]), 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 more sophisticated, 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. As the SCA paradigm dictates the existence of certain components in the service platform, our contribution is the identification of the exact points where these SCA-specific components could be enriched with self-adaptiveness, as well as our corresponding implementation, performed as an extension of Apache Tuscany.

Main intension of the adaptation mechanisms is to keep the QoS above certain limits. A recent survey [13] 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 [14], or on service discovery and group collaboration [15]. 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., [16]) define five 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., [17]) prove that almost no current SOAs achieve that maturity level, but they rather lie on lower levels, usually levels 2 and 3. The referenced study was done among software departments of the german banking industry, which are supposed to be among the leaders of SOA adoption.

III. Further Challenges of our Scenario

The purpose of our extended platform is to serve as the SOA substrate for our project [18], which 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 V. The use of our platform in such a scenario indicated a long list of requirements. In the following, we list how these requirements can be summarized or 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 service bus, but also no conceptual or architectural approach, which addresses these needs (cf. Section II). As for the implementation, the extensions that will be presented were necessary also because of the following lacking capabilities, which are absent from Apache Tuscany, but also from all other examined platforms:

- 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 can only support static logging and not the dynamic monitoring logic that we will describe in more detail in the next sections.
- There are no replication or maintenance mechanisms for the internal application services.

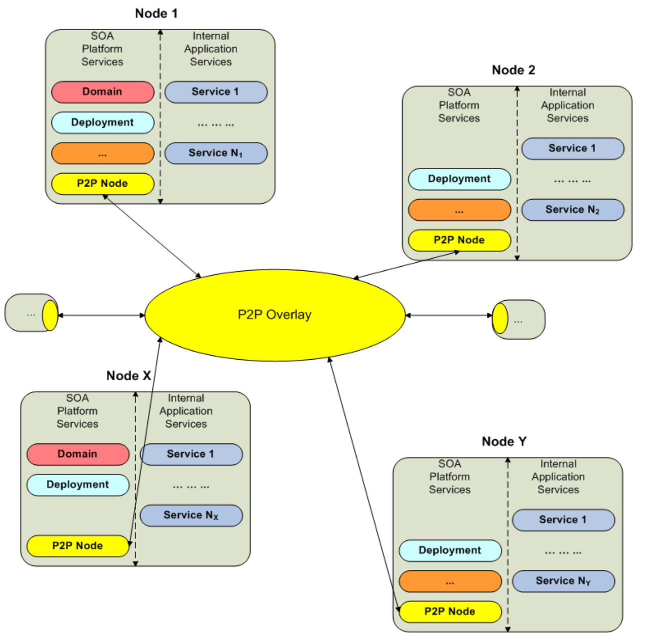


Figure 1. Overview of the p2p-based distribution of the core parts

IV. Service Platform Availability Extensions

With regard to the described challenges, we present in this section a generally applicable idea of how they could be handled inside a service platform, and then we briefly describe how we implemented most parts of the concept by modifying the Apache Tuscany service platform. In the third part, we go into more detail in order to explain how our extensions work. This part mentions implementation details only when this is helpful for the understanding.

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 II (registry mechanisms, address resolution, service deployment, and monitoring). Our main idea was to re-define these 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

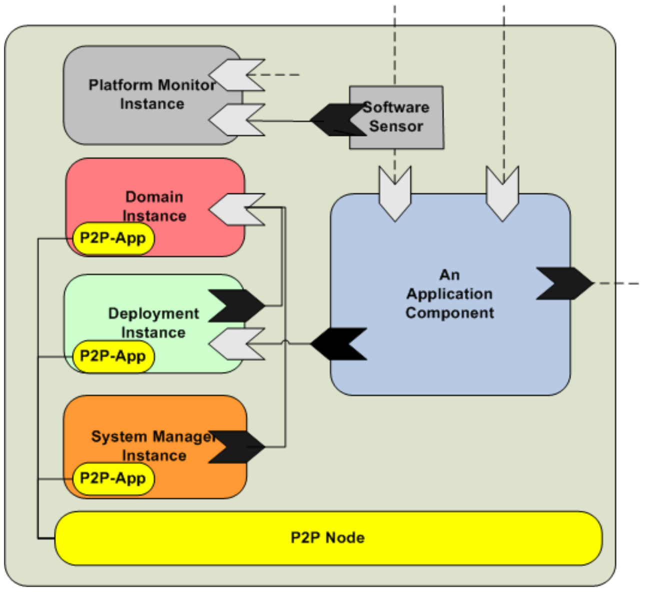


Figure 2. Component interrelations in a node of the modified platform

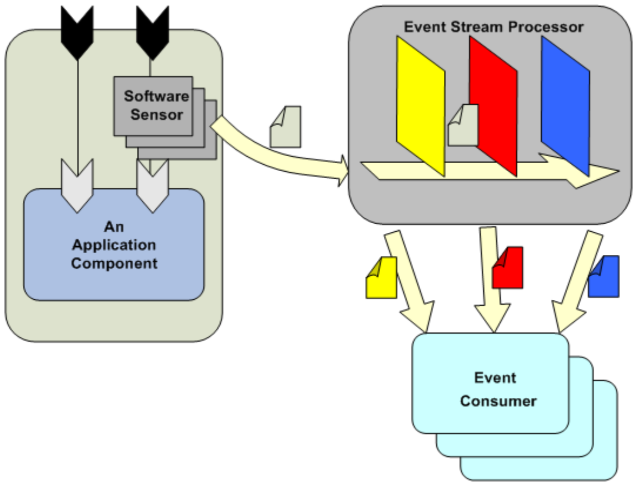


Figure 3. Adaptation-enabling event processing of the platform monitor

of the service bus have to be addressed when core functionalities are requested by any node of the system.

The choice of p2p is driven by our striving for fault-tolerance. The failure of peer nodes, on which 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 the p2p technology does. 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 (Figure 1). We re-define, extend and distribute four core parts, while a lot of accompanying platform parts/functionalities are abstracted in our concept and taken 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 might be enough) and offers the extended possibility of returning multiple endpoints to service-lookups. This supports the usage of service replicas that are 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 store their resources when they are registered in the domain and that these resources are sufficient 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.

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 (ESP) concept [19] 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 three 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 Figure 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.

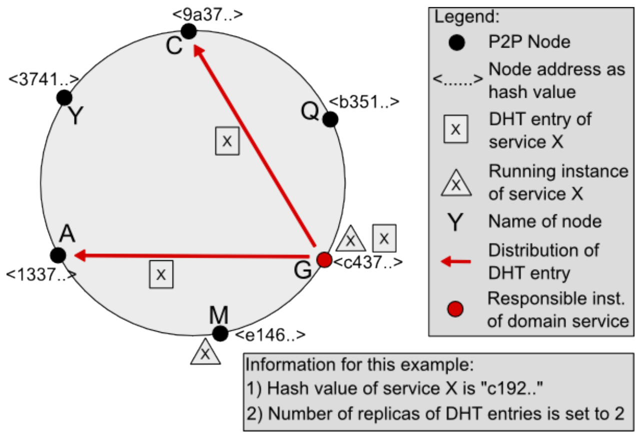
We had to define a new node type, the *CoreNode*, which merges an SCA node with a p2p node. While the extended domain, deployment, and system manager are based directly on the p2p node, the platform monitor is built on the (modified) service invocation mechanisms of the platform, enabling the binding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

The API of each core part corresponds to the functionalities described in Section IV-A. We provide here 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.
- *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.

C. Service lifecycle phases for redundancy and self-adaptation

With regard to the presented concept, design and implementation of our service platform extension, we present a more detailed view of the lifecycle of a service with the focus on self-adapting and self-organization.



<9a37..>
C
<3741..>
Y
<b351..>
Q
A
<1337..>
G
<c437..>
M
<e146..>
X
Legend:
P2P Node
Node address as hash value
DHT entry of service X
Running instance of service X
Name of node
Distribution of DHT entry
Responsible inst. of domain service
Information for this example:
1) Hash value of service X is "c192.."
2) Number of replicas of DHT entries is set to 2

Figure 4. DHT entry distribution

Phase 1 - Service Registration and Distribution of Service Resources:

The most important aspect of the domain service is the transparent distribution of its database, which holds redundantly all the information needed for a service, e.g., its name, its configuration, and its resources. In order to coordinate the self-organization tasks, there is only one instance of the domain service which is responsible for the registration and for any modifications of a particular service X. The same instance is also responsible for returning multiple endpoints to service look-ups of the service X.

To understand the nature of the responsibilty of an instance of the domain service for a service X, it is important to know the structure of the used p2p overlay. Our service platform extension uses *freepastry*, which is an open-source implementation of Pastry [20]. Pastry uses a ring structure for the peer-to-peer overlay, where every node of the ring is identified by an unique address. The address of a node is a hash value randomly allocated by the *freepastry* library during the bootstrap process, i.e., when the node enters the overlay. An example ring structure is presented in Figure 4, where the name and the address of every node are illustrated among other information.

To address a node of the ring it is not necessary to know the accurate address of it, but one hash value part of the range of addresses which "belongs" to the node. For the example of Figure 4, this means that node C with the hash value $\langle 9a37..\rangle$ is addressable by any hash value from $\langle 3741..\rangle + 1$ to $\langle 9a37..\rangle$. Thus, every node of the ring is responsible for the hash values from the address of its predecessor (node Y in the example) to its own. For more details about this kind of hash-based addressing in p2p networks, we refer to [21].

The responsibility of a domain service instance is the range of hash values which belong to the node where the instance is located. To find out which domain service

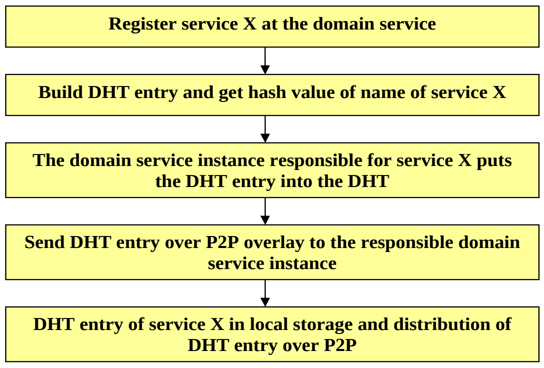

Register service X at the domain service
Build DHT entry and get hash value of name of service X
The domain service instance responsible for service X puts the DHT entry into the DHT
Send DHT entry over P2P overlay to the responsible domain service instance
DHT entry of service X in local storage and distribution of DHT entry over P2P

Figure 5. Registration of a service X

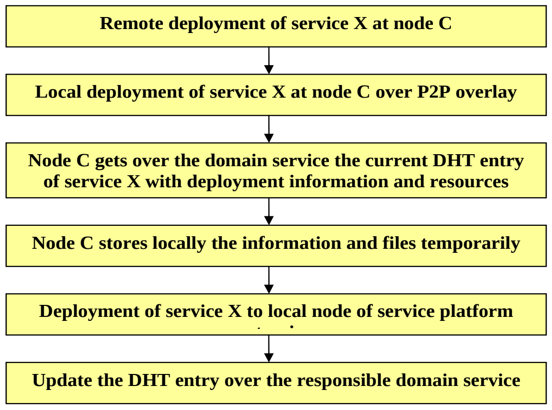

Remote deployment of service X at node C
Local deployment of service X at node C over P2P overlay
Node C gets over the domain service the current DHT entry of service X with deployment information and resources
Node C stores locally the information and files temporarily
Deployment of service X to local node of service platform
Update the DHT entry over the responsible domain service

Figure 6. Remote deployment of a service X

instance is responsible for the deployment of a service X, the name of the service will be transformed to a hash value as well. The domain service instance responsible for the hash value obtained from the service name is then addressed in order to deploy the service.

For the registration of an internal service, i.e., for a service that has been created within our platform and is running on it, it is important to know which information must be saved and how the domain service will distribute this information inside the platform. Before explaining the sequence of this information distribution, the Distributed Hash Table (DHT) entry is introduced. A DHT entry contains the name of the service, the data files needed in order to deploy the service on the extended service platform, and information about the nodes where the service is already deployed. In our implementation, the mentioned data files are stored as a JAR file of the service X in its DHT entry. This offers the possibility of remote deployment of the service. This feature will be explained in more detail in the next phase.

The information of the DHT entry includes the status of the service on each node as well. This status can have the values “started”, “running” and “stopped”. “started” stands for the phase when a service is registered on the domain service but not yet deployed. After a successful deployment, the status is changed to “running”, while an undeployment of the service switches the status to “stopped”, until the information about the node where the service was undeployed is deleted.

The different steps of the registration process are presented in Figure 5. The process starts with the registration of service X at the instance of the domain service where the DHT entry of the service was built and the hash value of the service name was calculated. Then the hash value is used to address the responsible instance of the domain service over the peer-to-peer overlay and the DHT entry is transmitted to it. The responsible instance puts the entry into the DHT, which stores the entry locally and also distributes it over the peer-to-peer overlay to other nodes. The number of replicas of the entry is predefined at startup of the peer-to-peer overlay.

The distribution of the DHT entries is illustrated in Figure 4, where the domain service instance which is responsible for the example service X is running on node G. The service itself is running on nodes G and M, while the DHT entry is stored locally on node G and transmitted to nodes C and A as replicates. It is important that the information about a service must not be stored locally where the service is running. The information can be somewhere else inside the peer-to-peer network.

Through this redundant storing of the DHT entries, there is no single point of failure for getting information about services whenever this information is needed. Even more important, the resources that are needed to start the service are also available on more than one node, together with the rest of the information.

Phase 2 - Local or Remote Service Deployment:

The nodes of the extended service platform have an instance of the deployment service running, thus providing the possibility of deploying a service locally and remotely. With the remote deployment function, it is possible to deploy a service X on any other node inside the service platform. This requires the cooperation of the deployment service instance that “wants” to start service X with the domain service instance that is responsible for service X. However, this is performed seamlessly, as the deployment service only addresses its local domain service instance. The latter locates then the domain service instance which is responsible for modifying and re-registering service X.

The procedure of a *local deployment* includes the registration process described in Phase 1. If the service to be deployed is already registered, the information that other nodes have about the status of the service is updated. After the registration, the deployment continues by adding the service to the platform, using the locally stored data. In the case of Apache Tuscany, the only resource needed for the deployment of a service is a JAR file. Then, the status of the service in the DHT entry is updated to "running". Again, for this update, the deployment service seamlessly addresses the responsible domain service instance.

The detailed procedure of a *remote deployment* for a pre-registered service X is presented in Figure 6. To deploy service X on another node C of the platform, a deploy message will be sent over the peer-to-peer overlay. The node C receiving this message gets the DHT entry from the domain service and saves the information and the resources locally. Then the local resources are used to deploy service X with the local deploy method, including the update of the status of service X over the responsible instance of the domain service.

Phase 3 - Static or Dynamic Service Replication:

Another important additional feature of the extended service platform with regard to self-adaptation is the static or dynamic replication of a service. The part that mainly enables this feature is the system manager. This is performed by a mechanism called Service Instance Control Mechanism (SICM), which works in strong cooperation with the domain service and the deployment service. The dynamic replication, i.e., the replication as a self-adaptation action, is, of course, also supported by the platform monitor.

The SICM can be started for a service X directly after its successful deployment (static replication), or it can be called later by any other node of the extended service platform (dynamic replication). The number of deployed instances of service X with the status "running" is read from the DHT entry. This number is compared to a threshold (minimum number of needed service instances). This threshold has been passed to the SICM as a parameter.

Depending on the result of the comparison, the SICM terminates if enough instances are running. Otherwise, it uses the domain service in order to search for nodes where additional instances of service X could be deployed. If there are not enough nodes, the SICM will be idle for a predefined time and then start again. Otherwise it will start to deploy more instances of service X on nodes with enough resources that had no running instances of service X.

Additional to the SICM, the system manager provides interfaces for getting system information, or other information that support self-adaptation.

Phase 4 - Maintenance and Monitoring of a Deployed Service:

As already mentioned, a distributed, event-driven platform monitor has been added. The modules used by such monitors in order to capture and forward specific information, are called software sensors. The main features of the platform monitor, which will be described in more detail in the following, are the possibility of adding software sensors with a developer-defined focus, as well as the possibility to trigger different adaption actions for different "captured events".

The platform monitor implements the ESP concept. This means that it can gather and preprocess events from remote software sensors. Additionally, it offers the developer the possibility to implement a monitoring logic that may be different for each software sensor. This monitoring logic is defined by writing ESP queries with an SQL-like language, called Event Processing Language (EPL) [19]. With the following example, we give an idea of how such a query looks like:

```
select sender, count(sender)
as sentPackets from
Event.win:time(5 sec)
group by sender
```

The node that submits the query is called *actor*, because it will act (or better, react) upon the event defined in the query. For example, an actor registered at the platform monitor with the above query is interested in all services which transmit packets inside the network of the platform. The platform monitor will store the information that matches the query and will send back to the actor an event for every service that sends packets. This event will contain the packets transmitted by the service within the last five seconds.

To collect the queried information, a software sensor has to be registered at the platform monitor. Then, it sends the collected information back to the platform monitor. The procedure of a registration of a software sensor for service X and the calling of this service by a node B is presented in Figure 7. We refer to this software sensor as sensor S.

After the registration of S at the platform monitor, its creation is registered at the system manager. The system manager inserts S to a proxy of the service that is to be monitored, and not to the service itself.

After this step, S is successfully deployed and will send events to the platform monitor according to the query with which it has been created. In order to present the monitoring process when service X is called by node B, Figure 7 shows the corresponding sequence of actions. Node B perceives the call of service X as a direct call, but inside the extended platform the call is redirected to a proxy. At the proxy, all registered sensors observe the call and act according to their logic. For example, there is the possibility of sending an event just after the service call or after the service execution has been finished.

When the call finishes, the proxy will transmit the result to the calling node (B), which perceives it as a answer from the original service X.

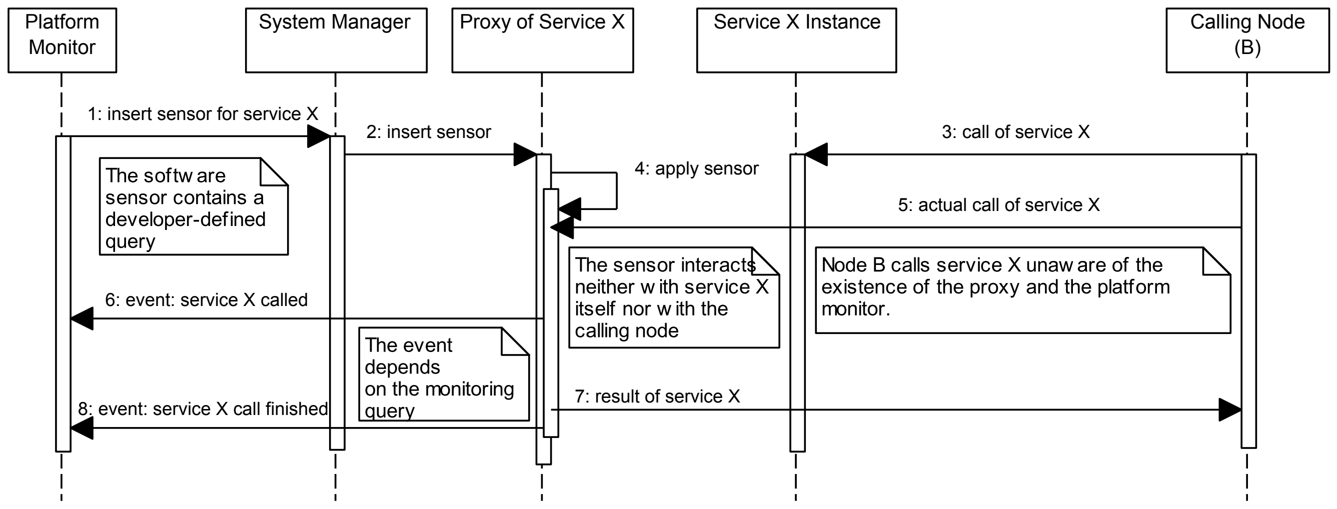

Platform Monitor
System Manager
Proxy of Service X
Service X Instance
Calling Node (B)
1: insert sensor for service X
2: insert sensor
3: call of service X
4: apply sensor
5: actual call of service X
The software sensor contains a developer-defined query
The sensor interacts neither with service X itself nor with the calling node
Node B calls service X unaware of the existence of the proxy and the platform monitor.
6: event: service X called
The event depends on the monitoring query
7: result of service X
8: event: service X call finished

Figure 7. Event-based service monitoring in the extended SCA platform

With the mechanisms of the platform monitor described above, it is possible for a developer to implement own software sensors and event consumers for services and to monitor them according to his needs. This is a completely new feature for Apache Tuscany and is to our best knowledge a highly innovative feature for any service platform.

V. Quantitative Evaluation: Availability Enhancement

In order to evaluate our approach with regard to the availability enhancements, which have been our main concern, 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. In the next section, we will describe an additional scenario, showing how a node can decide to adapt the protocols used by its services based on monitored information about the types of clients that dominate the system.

A. Evaluation 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*: 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.

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 Figure 8. 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 occurs. More details will be analyzed in Section V.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

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

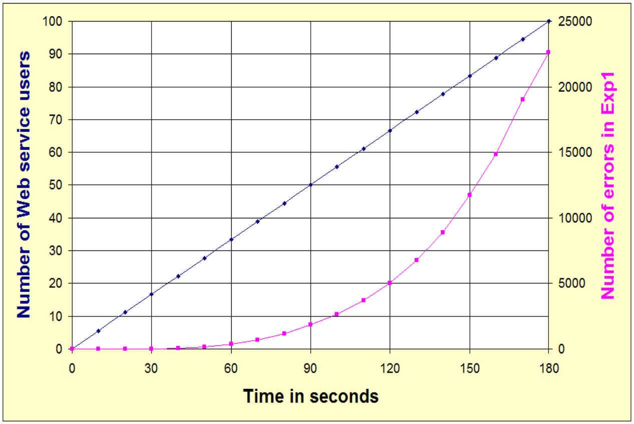


Figure 8. Experimental service invocation pattern

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 are running on the Apache Tuscany release platform and the invocations are equally distributed to them. The number of instances (3) was chosen empirically, so that it could almost always satisfy the given invocations' curve (Figure 8). For this case, as well as for the next two cases, the distribution of the invocations among the instances was simulated. 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 Figure 8, 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.

B. Comparison Results

Figure 9 and Figure 10 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 Section 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 much slower than in Exp1, 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. Figure 10 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 decreasing number of users, though this is not shown with the present experiments.

During the last minute of the evaluation, Exp4 presents a

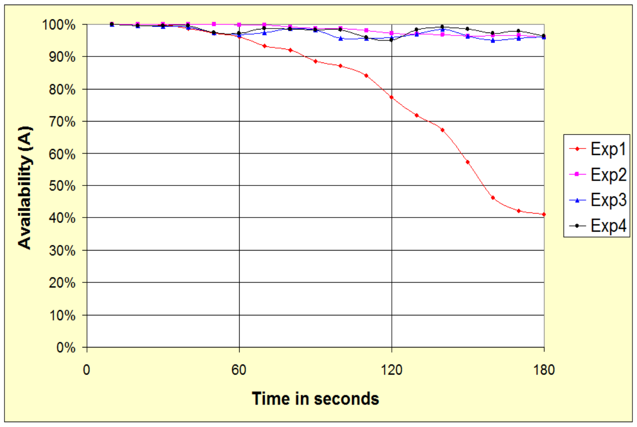


Figure 9. Measured availability

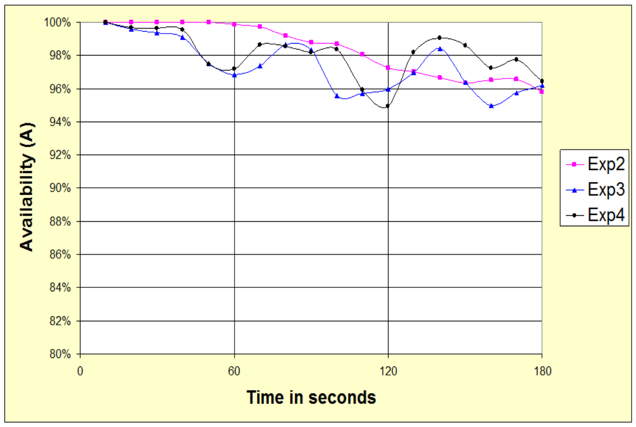


Figure 10. Adaptation effects

higher availability, because the number of service instances is increased 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 of their trade-off with the 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. Qualitative Evaluation: Dynamic Protocol Adaptation

While a quantitative evaluation has been presented in the previous scenario, the evaluation of this section is characterized as qualitative. A qualitative evaluation does not directly compare approaches in order to mathematically prove which one is the best one, but it rather provides (measurable) hints about how an approach could bring benefits and leaves space for further research. Indeed, we are going to explain on the basis of some experiments, why the dynamic protocol adaptation of services can offer benefits.

In this scenario, the adaptation is triggered based on client types rather than client numbers. More precisely, nodes use our platform extensions in order to extract information about the involvement of mobile service consumers and they adapt their communication protocols accordingly. Mobile usage of services is not the only case that can profit from dynamic protocol adaptation but it is by far the most important one. Thus, our evaluation will refer to this case. Before we discuss the benefits and the limitations of such adaptations based on experimental results, we describe the scenario and explain why it is our platform extension that enables it.

A. Dynamic Protocol Adaptation with Our Apache Tuscany Extensions

Although mobile SOA is pre-mature and the participants of SOA systems are usually stationary computers of IT departments, mobile SOA participants start to appear, usually as simple Web service clients. Mobile SOA participants have many differences to other participants, regarding both the way in which the devices consume the service and the QoS-efficiency of particular communication protocols [22]. For example, service buses, such as Apache Tuscany, normally cannot be used for the development of the client side, if the client is a constrained mobile device. Even more important, the standard communication protocol of Web services (SOAP) causes big delays in some cases of mobile Web service consumption. Although the service platform cannot be run and used on mobile clients, it could trigger service adaptation actions in cases of extensive mobile usage of particular services. Such an important adaptation action could be the dynamic adaptation of the protocol with which a service is offered by the platform. In the following, we describe this dynamic protocol adaptation that can be triggered by our extended platform. After that, we present some experimental results that demonstrate the importance of being able to perform such adaptations dynamically.

In order to explain dynamic protocol adaptation of a service with our extended service platform, a short description of how components and services are configured in SCA is necessary. SCA uses the Service Component Definition Language (SCDL) in order to define inside a configuration file (*composite file*) the components, the services, and their interactions inside the system. So, the architecture of a system, or of a system part, is implicitly defined in this file. Every service deployed to the service platform has to be contained in such a composite file, defining the attributes and settings of the service. The important part related to dynamic protocol adaptation of services is the determination of bindings, with which a service is made available. The bindings determine how a service communicates with other components, with each binding corresponding with a particular protocol. To explain the possibility of binding modifications in SCDL, we provide the following composite file snippet:

```
...
<service name="ServiceX">
  <binding.ws
    uri="http://www.a.com/serviceX"/>
  <binding.rmi host="www.b.com"
    port="8099" serviceName="serviceX"/>
</service>
...
```

Two main service settings can be seen in the above snippet:

- The name with which the service is defined inside the platform (ServiceX).
- The bindings with which the service is offered, in this case a Web service (SOAP) binding and an RMI binding. The Web service binding only needs the URI of the service, while the RMI binding needs the host address, the port number and the specific name of the service at the location it connects to.

To modify an existing service so that a new binding for it is added, the composite file must be edited and re-deployed. Let us assume that initially only the Web service binding is present for Service X and the usage of this service in the system changes in such a way that the addition of an RMI binding is desired. The first step for the modification is to fetch the current resources of the service from the DHT and to edit the composite file by adding the part written in bold font in the snippet above. The second step is to upload the resources back to the DHT, replacing the old, unedited files with the mechanisms introduced earlier. The last step is the re-deployment of the service which is recognized as a restart of it. After the re-deployment, the additional binding for service X can be used.

SCA supports several protocols and bindings, but it is not recommended to use every binding for every service from the beginning, and, of course, this is never done in the praxis. This is because the existence of many open bindings may lead to increased complexity, unnecessary traffic inside the network, or even security gaps. For this reason, a dynamic and adaptive modification of the bindings of a service is preferrable and is supported by our extended service platform. Furthermore, the whole process is easier with our implementation, because the programmatical re-deployment of a service is simplified in our extended platform, as it can be done through a simple function offered by the overlay.

Another example mechanism, not binding-based this time, for dynamically modifying the way with which a Web service can be accessed, is the activation or de-activation of compression for the SOAP communication. It is not implemented and supported by Apache Tuscany originally, but it is possible through modifications in the configuration of the used Web container. The platform monitor could sense an increase in the number of clients that would profit from compression, e.g., mobile clients, so that the adaptation action of activating compression would be then enforced.

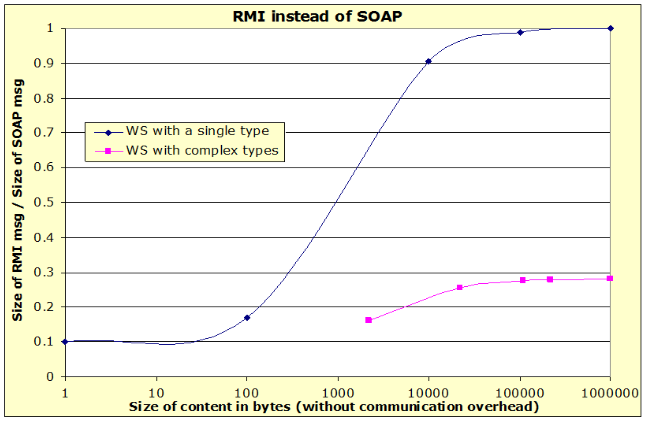
B. Experiments Showing the Potentials of Dynamic Protocol Adaptation

The experiments correspond with the scenario decribed in the previous subsection. Thus, it is assumed that one or more services are offered with a SOAP interface, which means, for our platform, with a Web service binding ("binding.ws"). The existence of more access methods (or bindings) for this service, e.g., over RMI or with data compression, may not be desired from the beginning, for various reasons. This can be, for example, because of system complexity, server costs, or security concerns, caused by the existence of many open endpoints.

Thus, the experiments are meant to answer the following question: "Assuming that the usage of a service changes in such a way that we need to reduce the communication overhead (for example, because more and more mobile clients are consuming it), can our monitored data help us decide which of the re-deployment options that our extensions enable is the most adequate?". It is reminded that Apache Tuscany offers various different types of bindings, but let us abide by our example and prove how the adequacy of RMI and compression depend on other data that could be captured by our platform monitor. In order to demonstrate this, many different experimental setups were possible. However, an interesting comparison is presented, for which no equivalent experimental results were found in literature. This is obviously because the interest in such comparisons is much bigger in the case of self-adaptive SOA platforms than in any other case.

The experimental setup is as follows:

- Two Web services were tested. One sends responses with complex types (a List of complex objects), the other sends responses with single types (a String of varying size).
- The size of the data in the response messages has been varied from 1 to 1000000 bytes (X-axis). In the case of the complex data, the minimum size was ca. 2000 bytes (= size of one Object).
- The two services were called directly with SOAP communication, as well as with the two alternative access methods, i.e., with the RMI protocol and with compression.
- The reduction of the data needed for the transmission of the responses was measured in all cases and was expressed as the size of the "reduced" response divided by the size of the original SOAP response (Y-axis). As mentioned, this overhead reduction is usually unimportant for strong workstations with great connections, but it may be critical for constrained mobile clients [23]. As nicely described in [24], this gap will continue to exist. Even the latest analyses of future technologies



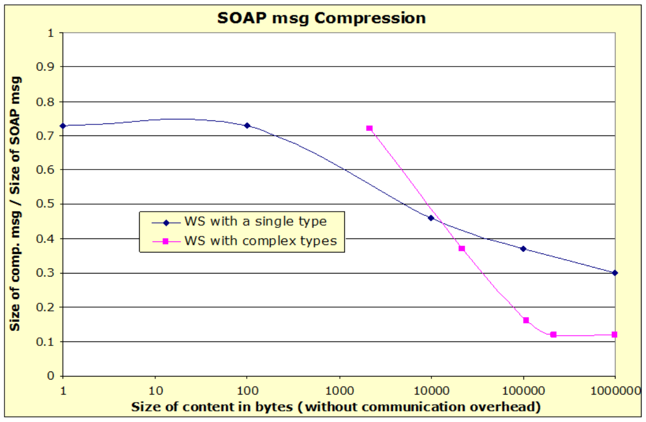


Figure 11. Overhead reduction with different protocol adaptations under varying conditions

for wireless communications strengthen this argument. In the book of Sesia et al. [25] about LTE (Long Term Evolution of 3G mobile networks), 5 categories of user equipment are defined, with smartphones being placed only under the second or third category. According to this categorization, devices of higher categories will be able to use wireless internet connection rates that are up to 6 times bigger. Of course, the wired connections of the future will be even faster than that, not to mention the fact that devices less capable than smartphones will be able to consume Web services. So, the big differences of device capabilities and connection qualities will maintain the need for adaptation and the overhead reduction shown in our experimental results will be always important, as the size of the data that is processed and wirelessly transmitted is growing parallel to all other technological developments.

No detailed analysis of the results shown in Figure 11 is necessary for our qualitative evaluation. The results show unambiguously that the two techniques perform differently under different conditions. For example, compression reduces the overhead significantly for single-typed big data, while the opposite is true for RMI. The conditions (data sizes and data types, in this example) can be perfectly captured by our platform monitor and exploited by a developer-defined adaptation logic. Concerning the exact logic, i.e., in order to answer the question “which adaptation action should be taken under which conditions?”, further experiments including all influencing aspects are needed and, of course, the application-specific requirements, as well as the developer preferences, play an important role. A corresponding decision support is an interesting area of research and is a subject of our future work.

VII. Conclusion

In this work, a concept for distributing the core parts of a service platform and enriching them with self-adaptation mechanisms in order to offer fault-tolerance and higher service availability has been presented. Based on a prototypical implementation of our concept, the mentioned enhancements were shown primarily through an evaluation scenario where service availability was measured for the original and the extended platform. The prototypical implementation was done as an extension of the state-of-the-art SCA platform, Apache Tuscany. In addition to the availability measurements, further possible enhancements through different adaptation actions were explained through a qualitative evaluation. In the following, we mention some limitations of our approach, as well as further aspects that we see 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.

However, the most important incentives for further research can be found in the qualitative evaluation that has been presented. There, it has been explained how the diversity of the users of the platform can lead to the need for different adaptation actions. As an example, mobile clients have been mentioned. On this basis, it must be researched how the different possible adaptation actions match different situations, so that new decision algorithms can be integrated in the logic of a self-adaptive SOA platform, such as the one presented in the work at hand.

VIII. Acknowledgments

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.

References

[1] Apostolos Papageorgiou, Tronje Krop, Sebastian Ahlfeld, Stefan Schulte, Julian Eckert, and Ralf Steinmetz. Enhancing Availability with Self-Organization Extensions in a SOA Platform. In *International Conference on Internet and Web Applications and Services (ICIW 2010)*, pages 161–166. IARIA, 2010.

[2] M. P. Papazoglou and W. J. Heuvel. Service-oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal*, 16(3):389–415, 2007.

[3] Rainer Berbner, Michael Spahn, Nicolas Repp, and Ralf Steinmetz. Heuristics for QoS-aware Web Service Composition. In *International Conference on Web Services (ICWS 2006)*, pages 72–82. IEEE, 2006.

[4] Dieter Schuller, Apostolos Papageorgiou, Stefan Schulte, Julian Eckert, Nicolas Repp, and Ralf Steinmetz. Process Reliability in Service-Oriented Architectures. In *Third IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2009)*, pages 640–645. IEEE, 2009.

[5] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. QoS-aware Replanning of Composite Web Services. In *International Conference on Web Services (ICWS 2005)*, pages 121–129. IEEE, 2005.

[6] Apostolos Papageorgiou, Stefan Schulte, Dieter Schuller, Michael Niemann, Nicolas Repp, and Ralf Steinmetz. Governance of a Service-Oriented Architecture for Environmental and Public Security. In *Fourth International ICSC Symposium on Information Technologies in Environmental Engineering (ITEE 2009)*, pages 39–52, 2009.

[7] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA*. Pearson Education, 2005.

[8] Jorge Salas, Francisco Perez-Sorrosal, Marta Patio-Martnez, and Jimnez-Peris. WS-Replication: a Framework for Highly Available Web Services. In *15th international conference on World Wide Web (WWW 2006)*, pages 357–366. ACM, 2006.

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

[10] Apache Software Foundation (ASF). Apache tuscany project, 2009. http://tuscany.apache.org/, last accessed on July 2010.

[11] E. Gjorven, R. Rouvoy, and F. Eliassen. Cross-layer Self-adaptation of Service-oriented Architectures. In *Third Workshop on Midleware for Service Oriented Computing (MW4SOC 2008)*, pages 37–42, 2008.

[12] G. Tosi, G. Denaro, and M. Pezze. Towards Autonomic Service-Oriented Applications. *International Journal of Autonomic Computing*, 1(1):58–80, April 2009.

[13] V. Issarny, M. Caporuscio, and N. Georgantas. A Perspective on the Future of Middleware-based Software Engineering. In *IEEE International Conference on Software Engineering (ICSE 2007), Proc. of the Workshop on the Future of Software Engineering (FOSE 2007)*, pages 244–258. IEEE, 2007.

[14] W. Bradley and D. Maher. The NEMO P2P Service Orchestration Framework. In *37th Annual Hawaii International Conference on System Sciences (HICSS 2004)*, page 90290.3. IEEE, 2004.

[15] D. Galatopoulos D. Kalofonos and E. Manolakos. A P2P SOA Enabling Group Collaboration through Service Composition. In *Fifth International Conference on Pervasive Services (ICPS 2008)*, pages 111–120. ACM, 2008.

[16] C. Rathfelder and H. Groenda. iSOAMM: An Independent SOA Maturity Model. In *8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08)*, pages 1–15. IFIP, 2008.

[17] Julian Eckert, Marc Bachhuber, André Miede, Apostolos Papageorgiou, and Ralf Steinmetz. Readiness and Maturity of Service-oriented Architectures in the German Banking Industry - A Multi-Participant Case Study. In *IEEE International Conference on Digital Ecosystems and Technologies 2010 (IEEE DEST 2010)*. IEEE, 2010.

[18] SoKNOS project. Service-oriented Architectures Supporting Networks of Public Security. http://www.soknos.de, last accessed on July 2010.

[19] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., 2001.

[20] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[21] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*. Springer Verlag, 2005.

[22] Apostolos Papageorgiou, Jeremias Blendin, André Miede, Julian Eckert, and Ralf Steinmetz. Study and Comparison of Adaptation Mechanisms for Performance Enhancements of Mobile Web Service Consumption. In *The 6th IEEE World Congress on Services (SERVICES '10)*, pages 667–670. IEEE, 2010.

[23] M. Tian, T. Voigt, T. Naumowicz, H. Ritter, and J. Schiller. Performance Considerations for Mobile Web Services. *Computer Communications*, 27:1097–1105, March 2004.

[24] C. Canali, M. Colajanni, and R. Lancellotti. Performance Evolution of Mobile Web-based Services. *IEEE Internet Computing*, 13:60–68, March 2009.

[25] Stefania Sesia, Issam Toufik, and Matthow Baker. *LTE, The UMTS Long Term Evolution: From Theory to Practice*. Wiley Publishing, 2009.