FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 43 (2018) No. 2
ISSN 0867-6356
DE GRUYTER OPEN DOI: 10.1515/fcds-2018-0005 e-ISSN 2300-3405

On Monitoring and Self-Adaptation to Dynamic Nature of SOA in ReServE Environment

Anna Kobusińska *

Abstract. Reliability is one of the bigest challenges faced by service-oriented systems. Therefore, to solve this problem, we have proposed ReServE — Reliable Service Environment. ReServE increases fault-tolerance of SOA systems and ensures consistent processing despite failures. However, the proposed environment imposes also the performance overhead. Thus, in this paper, we extended ReServE and added a monitoring feature provided by the M^3 service. As a consequence, the extended environment can adjust appropriately the load of its modules to the changing interaction and behaviour patterns of service oriented systems. We have experimentally shown that the proposed solution, while providing the required level of reliability, decreases significantly the performance overhead.

Keywords: SOA, fault-tolerance, recovery protocol, monitoring

1. Introduction

In recent years, the rapid proliferation of development and deployment of distributed systems based on service-oriented architecture (SOA) has been observed [1, 17]. In this approach, new applications are assembled from existing web services, implemented by software modules operating according to the established criteria, and representing specific functionality [13, 20, 26]. Such services are described, discovered and accessed over the Internet by using standard protocols. They have several features, which make them attractive components of distributed systems. Among the others they are: reusable, loosely-coupled, autonomous, and platform-independent. They are also self-contained, and do not depend on the context or state of other services.

Services are published by service providers and accessed by clients, who can benefit either from a single service or from a composite one, implemented by composing several services from independent providers in order to meet the objectives impossible

*Institute of Computing Science, Poznań University of Technology, Poznań, Poland, anna.kobusinska@cs.put.poznan.pl

to achieve through a single service. In consequence, the service-oriented approach allows to create new, complex applications, as well as to easily integrate existing systems. For that reason, in the SOA, an unprecedented so far flexibility in design of distributed applications is achieved.

On the other hand, SOA-based systems inherit all the challenges related to the construction of distributed systems. Among the others, they face the problem of unreliability of computing infrastructure. This problem manifests itself in failures of SOA components, leading consequently to limitations in availability of services, and thus affecting dependability of the whole system. Failures of SOA components may also cause problems with reliability of business processes (understood as a series of interactions between services and their clients). The problem of reliable interactions results from the possible inconsistency between the service state and its view at the client's side, which can appear after server or client failure and subsequent resumption of work. In the result, the state of a business process may be incorrect, and hence, further processing may be infeasible [9].

Above mentioned situations are highly undesirable from the viewpoint of service-oriented system clients, who expect that provided services are available, and business processing is reliable and uninterrupted. Since for many service-oriented applications the reliability aspect is particularly important, the provision of a satisfactory level of fault-tolerance of SOA systems and applications is a crucial practical and research problem. To solve this problem, we have proposed ReServE — Reliable Service Environment, which aims in increasing SOA fault-tolerance [4, 12, 14]. The proposed solution preserves business process state during the failure-free processing, in order to transparently recover a consistent processing state (i.e., one that could be accomplished in the failure-free processing), in case of failure of one or more system components. To achieve this, a well-known idea of logging interactions of business process participants, and replaying them during the recovery procedure is applied [9]. The proposed protocol of message logging and recovery used in ReServE is specially tailored to requirements of service-oriented systems. The presented solution focuses on seeking automated mechanisms that neither require user intervention in the case of failures, nor the knowledge of service semantics. It also respects the independence of service providers, allowing them to implement their own recovery policies and employ local methods to achieve fault-tolerance. But, although ReServE has many advantages, its performance could be improved.

In the proposed approach, in order to ensure reliability in a transparent way, requests issued by clients are intercepted by Client Intermediary Modules (CIM), and redirected to the Recovery Management Units (RMU). In turn, RMU modules direct requests to Service Intermediary Modules (SIM), from where they reach the service. There are many distributed RMU modules, but each CIM and SIM are registered in one distinguished, default RMU that is responsible for coordinating the recovery procedure in case of failures. RMU modules play the key role in ReServE environment, not only because of their impact on correctness of the processing, but also for performance reasons. One RMU module can be a default module for many clients and services, thus it is the most heavily loaded element of ReServE. Moreover, some RMU modules may be more overloaded, while the others are not fully used.

This may, in turn, lead to a sudden drop in service performance, in particular when a certain critical number of services and clients supported by ReServE is exceeded. Therefore, in oder to increase the environment performance, it is desirable to introduce in ReServE the mechanisms, which will prevent such situations.

Adaptation to a changing load is a non-trivial issue, which has been considered in many systems of various characteristics [3, 15, 16]. The existing approaches have to be tailored to SOA features and ReServE requirements. Among the others, in service-oriented systems, the mechanisms that impede overloading have to take into account the continuous nature of load changes. The load does not have to be linearly dependent on the number of clients or services. It is obvious that some services at certain times may be more popular than others, and as such they will be invoked more often. Similarly, clients can also generate a different number of requests. As a consequence, the load of *RMU* modules can also vary over time.

In order to properly estimate the overloading of the proposed environment, it is important to define the load and select the criteria that determine it. The criteria commonly considered for this purpose in distributed systems, such as a number of performed requests and a processor or a memory load, may be insufficient in the case of SOA systems. In general, in service-oriented systems such criteria may depend also on the service provider and client. Moreover, the criterion which is valid for one provider may be irrelevant (or less important) for another one (for example the energy consumption).

To solve above mentioned problems, in this paper we propose the solution that is able to monitor on-line the ReServE environment, in order to find, predict, and proactively prevent overloading of its components. In the proposed solution, M^3 service, being a part of DyMST tool [5], is responsible for monitoring, collecting and analyzing various statistics related to *RMU* characteristics. On the basis of results gathered by M^3, it is possible to estimate the load of each *RMU*, and react appropriately when such load increases. The monitoring feature, implemented by M^3, does not interfere with the primary ReServE functionality, related to increasing fault-tolerance. Thus, ReServE overhead associated with introduction of monitoring is, in the proposed solution, kept minimal. At the same time, due to monitoring and subsequent self-adaptation to the changed load, the performance of ReServE can be significantly improved.

The rest of the paper is organised as follows. Section 2 presents the existing services and platforms proposed for SOA monitoring. Next, system model is described in Section 3. Section 4 describes the architecture and general idea of ReServE environment. The discussion on the possible approaches to monitoring ReServE is presented in Section 5. Finally, the proposed solution is given in Section 6, and evaluated in Section 7. The paper is concluded in Section 8, which proposes also the future work directions.

2. Related Work

This section presents the contributions related to the monitoring of service-oriented systems. The market for monitoring systems and tools that observe the run-time behavior is evolving rapidly [8, 19, 21]. However, most of existing solutions have not been designed and implemented with SOA applications in focus. Instead, they are developed for general-purpose monitoring, such as monitoring of virtual machines or network stacks, and then enhanced and integrated to be used in the SOA world. Most current tools are focused on a particular monitoring aspect, for instance on application performance (APM) [11], network performance (NPM) [18], and business transactions (BTM) [27]. Below, solutions developed for service-oriented systems that might be used with ReServE environment are presented.

Paper [23] presents the solution that enables on-demand adjustment of the monitoring process to the changing SOA characteristics. The proposed solution discusses how to gather and represent knowledge about the monitored system in order to ameliorate the monitoring process. Next, the authors propose the Dynamic Adaptive Monitoring Framework (called DAMON), which can be installed in SOA-based environments in order to provide it with adaptive monitoring features. The evaluation results presented in the paper show that DAMON framework can significantly reduce monitoring overhead and increase scalability of the monitored environment. Although DAMON has many advantages, it is hard to integrate it with RESTful web services.

Another solution, presented in [7], is a monitoring framework, which ensures sustainability of service-oriented systems at the infrastructure level. The framework integrates several ontologies to monitor the performance of service oriented systems. The paper introduces the Service Monitoring Ontology that captures all the information about the service domain, and enables to merge other existing ontologies related to SOA systems performance. The solution is strongly ontology-focused.

The monitoring and self-adaptation approach of service-oriented collaboration networks is presented in [22]. The proposed solution regulates local interactions to maintain desired system functionality. While monitoring, it uses the notion of socially inspired trust. To solve the problem of monitoring, analyzing, and evaluating specific behavior of the SOA-based systems, the authors of [22] propose to integrate two separate independent frameworks — one providing a real Web service testbed extensible for dynamic adaptation actions, and the other enabling the self-adaptation. Unfortunately, the proposed solution is focused on the capabilities of human actors, defined as Human-Provided Services (HPSs) [25] and traditional Software-Based Services (SBSs). In the result, the business process users are engaged in the monitoring process, which is avoided in ReServE. In the proposed environment we assume that users are not engaged in the processing related with the work of the environment.

Another solution for monitoring SOA applications is SOOM (Service-oriented Online Monitor). SOOM is focused on multiple monitoring aspects [28], and enables monitoring of interoperable services, observation of IT transactions, identification of performance bottlenecks, locating and pro-actively predicting run-time errors. For this purpose, the considered solution inserts its lightweight software components into target SOA frameworks using various instrumentation techniques. The functionality

of this solution is based on performing a collection of operations on monitored objects, which can be invoked unconditionally and conditionally. Moreover, by applying various synchronization techniques, SOOM resolves conflicts when operations are executed concurrently. The advantage of SOOM is the support of RESP and SOAP architectural styles. Unfortunately, SOOM is focused on monitoring the SLAs and SLEs features at the service level, which makes it less appropriate from the viewpoint of its integration with ReServE

Finally, M^3 (*Metrics, Monitoring, Management*) distributed management platform, which manages loosely-coupled environments is proposed in [5]. M^3 follows the SOA paradigm — all its components are in fact loosely-coupled RESTful web services [10] with well-defined interfaces. M^3 main assumption is to keep the most up to date knowledge on managed resources (components) and dependencies between them. For this purpose, M^3 consists of distributed autonomic managers, and distributed registry that stores the information about their location, the components managed by them, as well as the dependencies between these components. Managers are self-configurable — they can automatically extend their own functionality by contacting plug-in repository and requesting appropriate plug-ins that are downloaded and dynamically loaded (sensors and effectors). Mangers are responsible for automatic detection of the components present in their surrounding. Being focused on keeping knowledge about dependencies between managed components, M^3 enables usage of mechanisms of failure and anomaly diagnostics, hence adding self-healing property to the system. M^3 is equipped with some mechanisms that ease human administrators in their work, and try to replace them in part of their responsibilities.

In this paper we decided to focus on the M^3 solution and to integrate it with ReServE service.

3. System Model

Throughout this paper a distributed service-oriented system is considered. The system consists of a number of autonomous, loosely-coupled RESTful web services [10, 24, 26], exposed as resources, and identified by a uniform resource identifiers upon which a fixed set of HTTP operations is applied. Thus, a client who wants to use a service communicates with it via a standardized interface, e.g., GET, PUT, POST and DELETE methods [24], and exchanges representations of resources. It is assumed that both clients and services are piece-wise deterministic, i.e., they generate the same results in the result of a multiple repetition of the same requests, assuming the same initial state. Services can concurrently process only clients' requests that do not require access to the same or interacting resources. Otherwise, the existence of a mechanism serializing access to resources, which uniquely determines the order of operations, is assumed.

The communication model used in the paper is based on a request-response approach, and does not guarantee the correct delivery of messages (they may be lost or duplicated). The considered communication channels do not provide FIFO property. Additionally, the crash-recovery model of failures is assumed, i.e., system components

may fail and recover after crashing a finite number of times [2]. Failures may happen at arbitrary moments, and we require any such failure to be eventually detected, for example by a Failure Detection Service [6].

We assume that each service provider may have its own reliability policy, and may use different local mechanisms that provide fault tolerance. Therefore, in the paper, by a recovery point we denote an abstraction describing a consistent state of the service, which can be correctly reconstructed after a failure, but we do not make any assumptions how and when such recovery points are taken (to take a recovery point logs, checkpoints, replicas and other mechanisms may be used). It is assumed that each service takes recovery points independently (and has at least one recovery point, representing it's initial state). Similarly, the client may also provide its own fault tolerance techniques to save its state.

4. ReServE — Reliable Service Environment

In this Section, the concept and the general architecture of ReServE environment is presented. The detailed description of ReServE has already been put forward in [4, 12, 14], and it is summarized here in order to make a paper self-contained.

4.1. General idea

The idea of the proposed solution results from the fact that in SOA processing is based on exchanging messages. The performance of the business process, understood as a sequence of interactions between clients and services, results frequently in resource state changes and leads to client-service inter-dependencies. Upon a failure of one of interacting business process participants, such dependencies may force remaining participants that did not fail to rollback. Otherwise, the state of the business processes can reflect situations impossible in any correct failure-free execution. On the other hand, due to the SOA assumption on autonomy of services, the failure of one process cannot influence the processing of the others. Since service providers do not provide information on the internal implementation of services, it is not known which interactions introduce inter-process dependencies and result in state changes. Therefore, in general, the recovery of a failed business process component should be isolated to avoid the cascading rollback of other processes.

Based on above observations, ReServE environment intercepts the communication between processes during the failure-free processing, and logs all interactions (requests of service invocations and the appropriate replies) in the persistent storage offered by the environment. The intercepted messages reflect the complete history of communication. Hence, after the failure of any client or service, replaying their messages in the same order as before the failure allows to recover the consistent processing state. But, due to the fact that business process participants may have their private mechanisms providing reliability, their state after the failure may be partially reconstructed with the use of local mechanisms. Therefore, the recovery process should

exclude the messages recovered with the use of local reliability mechanisms. The task of ReServE environment is to find such messages, and replay the remaining ones to the service in the same order as before the failure. After re-execution of recovered requests, the proposed environment intercepts replies from the service, because they have already been sent to clients and other services during the failure-free execution. ReServE ensures idempotency of obtained requests. The client's request, to which the reply has already been obtained by ReServE and saved in the persistent storage, is sent to the client immediately, without the need of sending the request to the service once again. Thus, the same message (i.e., the message with the same identification number) may be send by a client multiple times, with no danger of multiple service invocations.

4.2. ReServE architecture and primary modules description

The architecture of ReServE is shown in Figure 1. ReServE has a modular construction, and consists of Recovery Management Units (RMU), Client Intermediary Modules (CIM) and Service Intermediary Modules (SIM). Below, a description of the components of ReServE environment is shortly presented and tasks which they perform are briefly characterized.

Recovery Management Units (RMUs) are the backbone of the proposed environment. The main task of these modules is to ensure durability of messages exchanged among clients and services during the failure-free business process execution. For this purpose RMU stores requests and replies sent among business process participants in Stable Storage, able to survive all failures. The saved history of interactions is then used during recovery of consistent processing state. It is assumed that ReServE consists of multiple RMU modules, which are distributed and located at different system nodes. Since there are many RMUs, the history of interactions is dispersed among them.

The RMU includes the following components: (1) Management Module — manages the execution of specific operations associated with saving the state of communication and rollback-recovery actions; (2) Stable Storage — stores the data necessary to recover the state of processing; (3) Garbage Collection Module — monitors the status of non-volatile memory and removes unnecessary data, in order to ensure high performance of ReServE service; (4) Recovery Cache Module (RCM) — volatile memory that stores information on services and RMU modules in which these services are registered.

To make ReServE environment transparent to business process participants, and to fully control the flow of messages in the system, proxy servers, called *Service Intermediary Modules* (SIM) and *Client Intermediary Module* (CIM), were introduced. CIM and SIM serve as proxies for clients and servers, respectively. They also hide the internal architecture of ReServE environment, and the details of recovery process. For this purpose, modules intercept requests and replies issued by clients and services, and forward them to appropriate RMU modules. Additionally, proxy servers implement some of tasks associated with processing, and thus they reduce the amount

of work performed by services and clients. This allows to reduce the requirements for clients and services that use the ReServE environment.

In addition to above mentioned functionality, CIM provides the appropriate interface for accessing RMU functions designed directly for clients. Among them are: the ability to store the state of client's processing, the acquisition of response to recent request issued within any conversation (i.e. processing performed by a client during realization of a business process), the acquisition of a list of active conversations, and the list of RMU modules with which the client had communicated. Furthermore, CIM module contains the client cache module, which is developed in order to increase processing efficiency. When the client invokes the services registered in RMU module other than client's default module, then addresses of RMU in which the requested service is registered, is saved in the cache of client's CIM.

In turn, the main task of SIM module is to monitor the service status, and to react in case of the possible failures. It is the service proxy that is responsible for initiating and managing the service recovery. For detection of service failure, FADE service [6] is used. Service proxy is designed to operate in two modes: normal and recovery. Normal mode is performed during failure-free processing. In normal mode, the task of service proxy server is to manage clients requests sent by RMU to the service, and to capture generated responses and returning them to the RMU module. In turn, recovery mode is activated when the service is restarted after a failure. In its first phase, called restore phase, SIM requires the service to roll back to the latest possible recovery point (i.e. service internal state preserved specially for the purpose of the recovery), taken by the service before the failure. Next, SIM asks RMU to replay all requests obtained by the service after this recovery point was taken. In the second, retry phase, service proxy server is responsible for receiving from RMU, and sending to service all requests, which re-execution will allow to recover a consistent processing state. Is should be mentioned that requests have to be replayed in the appropriate order. SIM performs also the detailed analysis of requests directed to the service. It filters out the outdated requests, i.e. the requests sent before the failure but received by service after its rollback-recovery.

4.3. Outline of ReServE processing

When processing starts, each of its participants registers itself in the selected RMU. Each service is registered in one RMU (referred to as a service default or master RMU), but the single RMU can be used by many services. In turn, the client can be registered simultaneously in many RMUs, but always one of them (called a client's master RMU) stores information on other RMU's used by the client.

Let us assume, that in Figure 1 client A and service Y have the same default RMU module: RMU_N, while client B default RMU is RMU_1 and service X default RMU is RMU_k. When client A invokes service Y, the request issued by a client is intercepted by the client's CIM (1), and forwarded to the client's default RMU_N (2). If the required service is registered in this RMU, like in the considered example, the request is saved in the RMU's Stable Storage, and forwarded to the service through

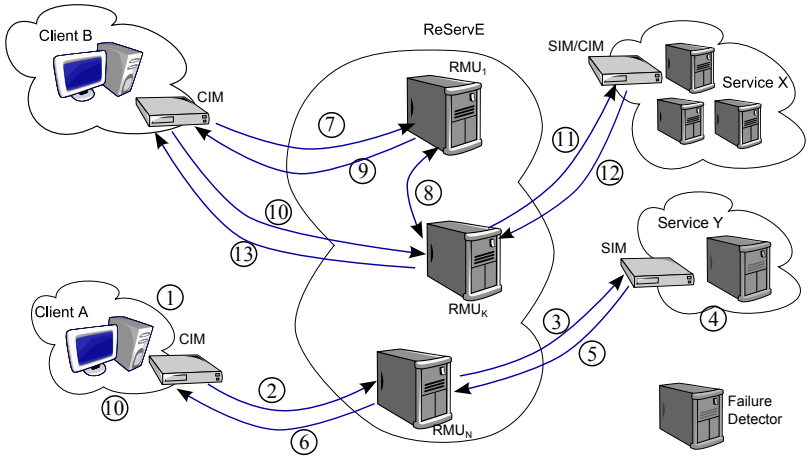


Figure 1: *ReServE* architecture

its SIM (3). The service processes request (4), and sends the response back to RMU_N (5). The response is saved in the stable storage and forwarded to the client through its CIM (6). If the RMU module obtains the client's request, to which the response has already been saved, then such a saved response is sent to the client, and there is no necessity to send this request once again to the service (in this way the mechanism of communication history is also used to ensure idempotency of all requests). However, when client B invokes service X (client and invoked service have separate default RMU modules), then a client obtains the URI of requested service default RMU from its default RMU (7,8), and sends back this information to the CIM (9), which reissues the request to a proper RMU (10, 11).

In case of client's application failure, we assume that there are three ways to resume the processing of a business process, which can be intermixed. The first one depends on the state explicitly saved by the client in it's default RMU, along with the identifier of the latest request, for which a response has been received and processed before saving the state. The second one is based on repeating all requests issued by the client after saving its state in the RMU. The third one uses the special RMU's resource that represents the latest service response sent from RMU to the client. In the case of web-driven applications, the latest response from the service is enough for client's recovery, as it enables the client to determine the progress of the business process and its continuation (according to the HATEOAS principle of Resource Oriented Architecture [10]). Since, in general the client communicates with many RMU modules, such a latest response should be chosen on the basis of information received from each RMU, the client contacted before its failure. Therefore, CIM obtains from its default RMU a list of all such RMUs, contacts with them, and asks them for the identifier of the last response. The response with the highest request identifier represents the last response, which was received by the client.

After the service failure, its SIM starts the rollback-recovery of the service state by getting from the service the information on its available recovery points taken before the failure, along with the information on the identifier of the last message contained in each recovery point. Simultaneously, SIM asks the service default RMU to send to it an identifier of the oldest request, which has not received a response yet. Based on obtained information, SIM indicates the recovery point to which the service has to be rolled back (the one that contains the responses which have not been saved in the RMU). When the service state is rolled back to the chosen recovery point, SIM requires RMU to resubmit all requests performed by the service before the failure, and not saved in the chosen recovery point. All such requests are designated, sent again by the RMU, and re-executed by the service, in the same order as before the failure.

5. Discussion on possible approaches to monitoring and adaptation of ReServE

The proposed ReServE environment, does not prevent the overloading of its RMU modules, which may lead to reduced ReServE performance. One of the reasons of such a situation is the changing interest in services registered in the RMU, resulting, for example, from the greater popularity of a given service at a certain period of time. With the increased service popularity, also the number of its invocations increases, which has the impact on the increased RMU processing and reduction of its performance.

Also, in the existing ReServE environment, there is still a high probability that a large number of clients and services will choose the same RMU as their default module. This is due to the currently used method of assigning default RMU modules to clients and services, which is done statically, by using the configuration file. As a consequence, some RMU modules have to process disproportionately large number of requests obtained from clients and replies from services, in comparison with other modules. Consequently, also the stable storage of such RMU modules is overfilled. As a result, the performance of the overloaded RMUs (and thereby also the ReServE environment) may be unsatisfactory from the viewpoint of some of business process participants.

In order to solve this problem and prevent overloading of RMU modules, the ReServE environment should monitor the changing number of supported clients and services, and varying amount of interchanged messages, and adapt to the observed changes. While introducing the above mentioned functionality to ReServE environment, it is important to guarantee that monitoring and adaptation to the changing nature of the processing will not affect the correctness of the recovery protocol applied in ReServE. On the other hand, it has to be underlined that the adaptation to the changing load does not need to result in the load balancing, but it's aim is to prevent the overloading of modules.

One of the possible solutions to solve this problem is the dynamic changeover of default RMU modules. In the consequence, users whose default RMU module

is excessively overloaded are allocated and registered in another module, which will become their default RMU. Another solution consists in giving the possibility to choose a given RMU module as a default one, only if the number of users registered in such RMU does not exceed a settled limit value. In both approaches, the question of who, when, and under which conditions settles the default RMU has to be answered. It also should be indicated how the new default module is designated.

Let us consider first that the dynamic changeover of default RMU modules is allowed. In the most straightforward solution, the RMU module itself decides when its load is too high and it cannot serve as a default module for all registered clients and services (the most complete knowledge about the load of the RMU module at a given moment has this module itself). In such case, RMU informs some of registered SIM and CIM modules that they have to change their default RMU. The considered solution enables fast reaction in the case of RMU load changes.

Unfortunately, despite its undoubted advantages, such a solution is difficult to be deployed in ReServE, because the continuous changing over default RMU would require a significant modifications in the recovery protocol. Those modifications are related to the necessity of providing the access to the history of interactions among business process participants, which is used during the recovery of the consistent processing state. In the current solution, the messages that form such a history are saved by the default RMU modules. It means that along with changing over the default RMU module, the appropriate messages from its stable storage will have to be moved to the stable storage of a newly chosen default RMU. Such a solution is impractical, due to the size of the history of interactions. Alternatively, the history of interactions could be left unaltered on current RMU, but then, the request and replies which are required by recovery procedure have to be obtained from all RMU modules, which played a role of default RMUs for a given client or service. Both above mentioned approaches introduce an unacceptable performance overhead.

Another difficulty encountered in the considered solution is related to the determination of the moment in which the changeover of default RMU modules occurs. It is assumed that the default RMU initiates the switching procedure. However, this solution, although seems to be uncomplicated, may encounter some difficulties during the realization. Let us consider that RMU resigns from being a default RMU but it has not obtained all service replies. Such a reply is thus forwarded to the newly chosen default module. However, in the meantime, new default RMU might have also changed over. As a result, the reply will circulate among RMU modules, and may not reach the client. The solution of this problem may rely on the introduction of a minimum time between consecutive changing over, or switching only after all required replies have been received by the considered default RMU module. The first method does not provide the flexibility, while the second one introduces the delay in the business process execution.

An alternative to the solution in which each RMU module itself determines when default modules should be changed, is the approach in which all RMUs determine together which of them are overloaded, and designate the moment of switching. On the basis of the obtained knowledge they also select the modules which will play the role of default modules. For this purpose, RMU modules may use the distributed consensus

protocol. Unfortunately, such a solution introduces an unsatisfactory communication overhead. Moreover, it is necessary to mark the initiator of consensus protocol in such an approach. Also, the problems discussed earlier, related to the exchange of history of communication saved by default *RMU* modules, or to the determination of the moment of change over of default *RMU* are encountered here.

Regardless of the approach discussed above, the dynamic switching of default *RMU* modules requires notifying clients and services on the new default module. It should be pointed out that the procedure of swapping default *RMU*s has to take into account the needs of clients and services taking part in the business processes, not only *RMU*s themselves. Since the needs of business process participants can change dynamically, clients and services should also take part in the process of determining which *RMU* can swap with the current default module.

After considering pros and cons of the solution based on the dynamic swapping of default *RMU* modules, let us consider the alternative approach, in which the *RMU* load is examined during the registration of business process participants in ReServE service. In this approach, *RMU* may refuse to register a new client or service if it is too overloaded. Although such a mechanism is not flexible, its important advantage is simplicity and efficiency.

It was noted that a large part of *RMU* load is introduced by clients who register (log in) for one session, and when it ends, they log out. Since the subsequent sessions are usually not related to the previous one, clients may register in another *RMU* to perform the new session. Therefore, in the next Section we will fo cus on the solution in which the *RMU* load is verified along with registration in ReSErvE service.

6. The proposed solution — integration of ReServE and M^3

In this Section, the protocol that increases the performance of ReServE environment is put forward. The proposed solution monitors the load of *RMU* modules and allows the appropriate adaptation to the load changes. Based on the analysis of the existing solutions that allow monitoring of SOA systems, we have decided to choose M^3 platform [5] and integrate it with ReServE environment. M^3 has several advantages over the remaining solutions, presented in Section 2: it focuses on monitoring of non-functional parameters of the system and performance metrics. Moreover, it supports the RESTful web services, and is self-configurable, which eases the task of its management.

6.1. M^3 architecture and functionality

Figure 2 presents M^3 architecture. M^3 platform consists of three components: Manager, Registry and Repository.

Managers are independent services that are responsible for management of one or more resources. The management actions include among the others: automatic

detection of resources in Manager surrounding, self-configuration with Registry and Repository services, or the support for consumers metrics (i.e. memory, graphs, db).

Registry service can be described as a central knowledge repository for the whole M^3platform. Registry is responsible for keeping information about active Managers present in the system and about their monitoring and management capabilities.

Finally, Extensions Repository is a simple service, often combined with the Registry, which stores the extensions, e.g. plug-ins that extend Manager functionality.

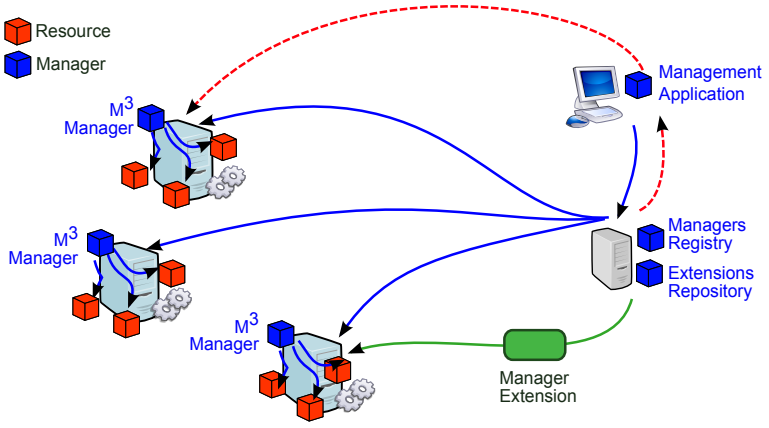


Figure 2: M^3 architecture

Each Manager detects resources in its surrounding, and registers itself in the Registry, along with the information about detected resources. In general, this information contains locations of managed resources, as well as monitoring and management capabilities, i.e. sensors and effectors.

Extensions are automatically uploaded and deployed in Managers if particular resources managed by the extension are detected. The range of functionalities provided by extensions is very wide, including: the possibility to define managed resource, discovery of managed resources, defining effectors and sensors for resources, and defining metrics for sensors that return performance values. The above mentioned sensors are extensions that provide functionality for collecting various monitoring data. They also provide definitions of metrics. In turn, effectors provide endpoints for controlling the environment in which a Manager is placed. In collaboration with sensors they can create control loops in order to continuously make impact on the environment. An example of sensor can be an extension that collects CPU usage from the operating system and provides average CPU usage metric for the last hour. Successively, one can use the effector to control the number of threads assigned to a web application, depending on the current and the predicted number of requests.

Registry and Extensions Repository send corresponding extensions to the Manager, that are meant to be used with the detected resources. Manager installs extensions and configures itself in order to support new functionality. It updates its new monitoring and management capabilities in the Registry and periodically checks for new versions of installed extensions.

Combining knowledge from Managers, Registry possesses a complete knowledge on the managed system. In order to keep the knowledge database up to date, the Registry sustains a simple pulse mechanism with all active Managers in the system. When a Manager registers or disconnects, Registry updates its knowledge repository accordingly. Similarly, Managers are obliged to monitor its surrounding for occurrences of new resources and inform the Registry about this fact. The Registry also provides a search feature, which can be used by external management application to search the knowledge database for managed resources, sensors and effectors.

The architecture does not depict it explicitly, but all platform components have RESTful Web services API.

6.2. Monitoring *RMU* functionality

As mentioned in the Section 5, different service providers can theoretically opt for different load measures, depending on their specific needs. It is difficult to predict all possible criteria according to which one can evaluate the *RMU* burden. Therefore, in this Section, we focus on the following performance metrics:

- system related metrics (average load, CPU usage, memory usage, throughput, storage device usage, network interface traffic),
- utilization of service/node,
- intensity of requests per second to a service,
- average request processing time for a service.

Additionally, we enable using metrics, which can be composed as an aggregation and normalization of above mentioned metrics. The function used to determine the value of this metric can be an average, a weighted average or e.g., a logistic function. Such a calculated metric can be used as an input for the extension used by the Manager from M^3 platform.

In the proposed solution, each *RMU* facilitates the configuration file with the URI that indicates where the desired metrics are located. This enables clients and services to choose the appropriate plug-in, which will be used by M^3 service to monitor *RMU*. Such a solution allows also the service provider to propose its own plug-in for calculation of any complicated load measure.

To add the monitoring functionality, M^3 service has to be launched at the same node as the *RMU* module. The service should be started before clients and services begin to register in *RMU*. The selection of *RMU* is made during the first registration of the *CIM* module in the system, and is based on the analysis of values of load metrics. In the proposed solution, such values are exposed by M^3 under the network address `<M3 URL>/resources/load`. The registration process and the selection of the appropriate *RMU* module is described in the Algorithm 1. Its outline is presented below.

```
Upon startup of client C_i
    :: Load values from the configuration file
electionOn ← RMU from configuration file
MasterRMU ← RMU from configuration file
RMUSet ← RMU from configuration file
RMUM3ManagersSet ← RMU from configuration file
    :: Set the RMU indicated in configuration file as a default one
CurrentRMU ← MasterRMU
    :: If client is not interested in elecion of the least overloaded RMU, leave the current
    RMU as a default
if !electionOn then
    return
end if
    :: Complete the RMU file with the main RMU
RMUSet ← RMUSet ∪ MasterRMU
RMUM3Managers[MasterRMU] ← MasterRMUM3Manager
    :: Get the metrics values from M^3 Managers
for RMU in RMUSet do
    LoadMap[RMU] ← getLoad( RMUM3Managers[RMU] )
end for
    :: Set the initial values
CurrentLoad ← inf
for RMU in RMUSet do
    if CurrentLoad > LoadMap[RMU] then
        CurrentLoad ← LoadMap[RMU]
        CurrentRMU ← RMU
    end if
end for

Upon recovery of client C_i
    :: Get the values from the configuration file
MasterRMU ← RMU from configuration file
RMUSet ← RMU from configuration file
    :: Add current RMU to the set of RMU
RMUSet ← RMUSet ∪ MasterRMU
    :: Register in RMU which has a default RMU role before the failure
for RMU in MasterRMU do
    if isMasterRMU( RMU ) then
        CurrentRMU ← RMU
        return
    end if
end for
```

Algorithm 1: Selection of the RMU module by CIM based on the monitoring knowledge from M^3

Each client intermediary module has a configuration file with the list of RMU modules available in ReServE service. One of the RMUs (the one which is located

in the nearest geographical distance from CIM module) is designated as the default RMU module. In the file there is also additional information: the M^3 communication ports of the load monitoring Managers for each RMU module, and the boolean value that indicates wheather the election of the least overloaded RMU has been started (lines 1–5).

If the client is not interested in choice of the least overloaded RMU it uses the default module, which is set in the configuration file (line 7). Otherwise, CIM checks if the RMU used by a client as a default one during his previous usage of the ReServE service is on the list and tries to register in this RMU module. If such module is not available, or it is overloaded and does not allow the registration, CIM starts the process of election of the new default RMU module. For this purpose, CIM contacts with M^3 Managers and retrieves the value of the chosen load metric (lines 13–18). Out of the obtained values the minimal one is chosen (line 20). Next, the RMU associated with the chosen value is elected as a default RMU (lines 21–26). After the registration confirmation, the client starts its processing, and acts correspondingly to the description presented in section 4.3, and in the paper [12](lines 27–38). In case of a negative response, the client sends a registration request to RMU, which metric value directly precedes the previously selected one. Finally, when all RMU module deny to register CIM, it starts the selection procedure from the beginning. For further processing purposes, the information on the chosen default RMU module is added to the configuration file of CIM.

The described approach does not influence the work of ReServE related with providing the reliability, and does not influence its burden. It also does not have an impact on the quality of work of other clients or services registered in the chosen RMU module. Moreover, all decisions made by RMU are based on its local knowledge, avoiding in this way the problems related to coordination between RMU modules described in the previous section.

Analogous adaptation mechanisms are implemented during the registration of new services in ReServE environment.

7. Simulation Experiments and Performance Evaluation

In this section, the performance of the proposed integrated approach, in which ReServE service and M^3 service are integrated is evaluated. The purpose of the performed simulation experiments was to estimate the overhead introduced by the proposed integrated solution during failure-free processing, and to examine how the integrated environement affects the recovery time. The obtained results are compared to the results obtained by ReServE environment without the monitoring feature.

In experiments, workstations with the following characteristics: SuSE Linux 11.3 kernel 2.6.34.8-0.2-desktop x86_64, with Intel Pentium 4 3.20GHz x 2 processor, and 8 GB RAM were used. Workstations served as clients, services and RMU modules.

In the performed experiments, a specially developed application, called Patient Registry, was used as a service. The main porpose of the service was to record all basic personal data of patients of the emergency room in the health center. Patient

Registry fulfills the requirements of ReServE environment. We assumed that each service and its SIM module run on the same workstation — the service is implemented with the RestLet 1.1. environment, while its proxy server uses proxy server called MProxy 0.4 [5].

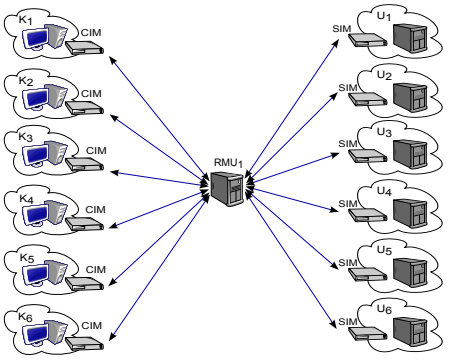
The task of workstations acting as clients was to generate requests issued to services and to measure the time of their processing from the moment of sending the request until the receipt of reply. Due to limitated number of available workstations, each physical node simulated the work of multiple logical clients (up to 40 clients per 1 node). Such a configuration was achieved by running many logical threads of client applications within a single physical machine. To implement client application, the Apache JMeter 2.3.4 software was used. In turn, client proxy, similarily to service proxy, was build with the MProxy 0.4 [5].

Other workstations were used to run RMU modules. Each of them had an access to the local PostgreSQL database, which was used as a stable storage.

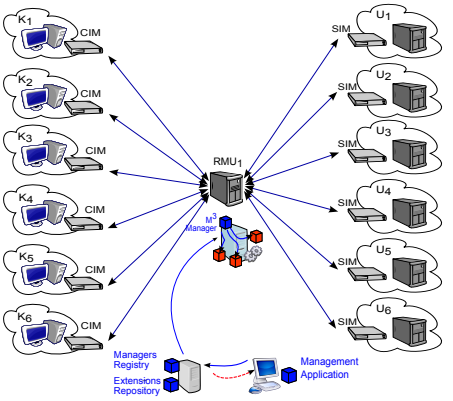
Simulation experiments were performed for ReServE architecture with 1, 2 or 3 RMU modules. Additionally, each of the considered configurations, was integrated with M^3 service. Table 1 presents the considered simulation environment architectures. In the performed experiments 6 workstations were used as clients, and 6 workstations played a service role. Services joined ReServE service sequentially. For 10 to 100 threads representing logical clients were running per client workstation. As a result, the simulation experiment involved up to 600 threads representing clients, and each thread repeated the simulation experiments 500 times. Moreover, it was assumed that the time associated with the service processing is constant, and is set to 100 ms (in general, the service's request processing time depends neither on the number of clients, nor on the size of data).

In the first configuration all clients and services were registered in one, common RMU module (Table 1 (a)). The second approach assumed the existence of two RMU modules, where in each RMU 3 services and 3 clients workstations were registered (Table 1 (c)). In the next configuration there were already all 3 RMU modules, with 2 clients workstations and 2 service workstations registered in each of them (Table 1 (e)). Next, for each of above mentioned configurations, each RMU module is extended by the Manager of M^3 service (Table 1 (b,d,f)). In considered configurations, logical clients running on one machine contacted one service running on the corresponding node (i.e. K1 invokes U1, K2 invokes U2, etc.) by sending 32kB requests to it.

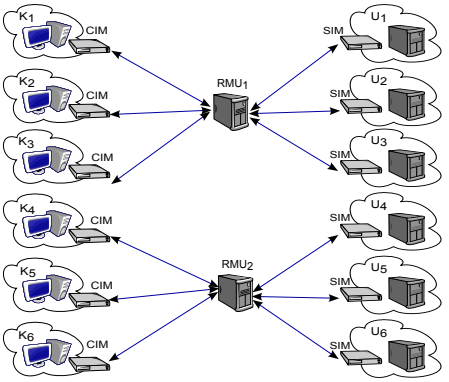
During the monitoring, M^3 Managers use *system load* metric, which is calculated as the average number of runnable processes over the given period of time (we include also the processes in the uninterruptible sleep state, that is, those processes which are waiting for disk I/O). The value of this metric is obtained with the use of the *uptime* command. The higher the metric value is, the higher is the burden of the monitored component.



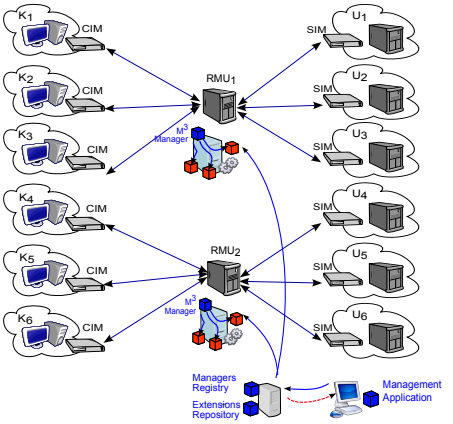
(a) Test environment: 1 RMU



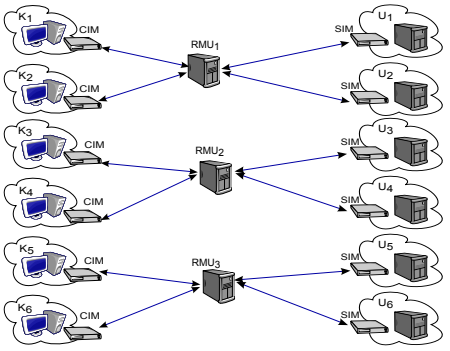
(b) Test environment: 1 RMU + M^3



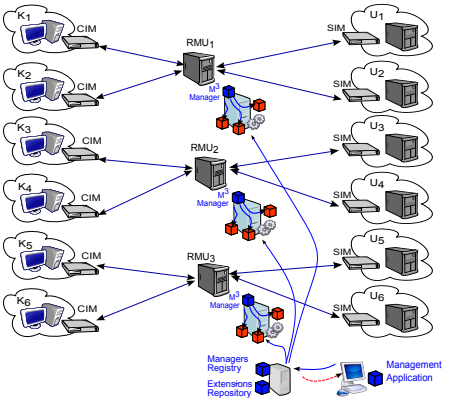
(c) Test environment: 2 RMU



(d) Test environment: 2 RMU + M^3



(e) Test environment: 3 RMU



(f) Test environment: 3 RMU + M^3

Table 1: ReServE test architecture

Client	RMU	RMU+M^3	2RMU	2RMU+M^3	3RMU	3RMU+M^3
24	596,94	597,73	373,17	371,72	366,24	362,50
48	1347,01	1349,13	474,65	470,13	387,26	385,32
72	2081,15	2082,00	698,59	697,41	532,605	532,38
96	2853,30	2855,28	1017,42	1016,82	712,46	710,01
120	3677,42	3678,24	1342,81	1342,11	936,75	932,94
144	4390,53	4392,13	1682,83	1681,22	1199,33	1189,98
168	5160,39	5162,41	1978,22	1977,00	1443,47	1441,23
192	5974,60	5975,69	2276,89	2274,27	1719,16	1713,98
216	6701,23	6702,33	2543,93	2541,30	1950,04	1942,48
240	7495,94	7496,92	2856,130	2853,21	2160,20	2169,91

Table 2: The processing time [ms] of 32kB request during failure-free processing

The first simulation experiment investigates how the integration of ReServE with M^3 service impacts the overhead introduced by the environment during the failure-free processing. For this purpose, the obtained results are compared with the overhead introduced by the pure ReServE environemnt, which uses the corresponding number of RMU modules. In the performed experiments, for each of the above mentioned configurations, the response times recorded by clients were compared. Table 2 presents the obtained results —- the average processing time of requests (in milliseconds) depends on the number of RMU modules and different number of clients. The graphic representation of data from Table 2 is presented in Figure 3.

The obtained results meet expectations — the time of processing requests increases linearly with the increasing number of clients, which is related to the increased number of performed requests, and time needed to save them in the Stable Storage. But, surprisingly, response times recorded by clients in the presence of six independent services were higher than in the case of one service, even though the load of each service in such a situation is effectively six times smaller. To verify this phenomenon, proactive tests were carried out, which tested the total processing time of individual fragments of the RMU module's code. For this purpose, the VisualVM application in version 1.3.2 was used. The performed tests revealed that the significant processing time (around 20%) was used to support RMU I/O buffers. On this basis, it was suspected that increasing the number of simultaneously serviced services, which were the subjected of a high load, caused a significant increase in the response data stream. In the consequence, the requests issued to services cannot be handled by RMU in real time, so they must be cached. As a result, the total processing time is increased. However, even if the load of RMU module is increased, the more RMU modules, the lower is the response time in the ReServE environment. This observation is natural, because in the configurations with 2 and 3 RMU modules, the effective number of clients using one RMU module, and thus also the number of requests performed by the module, is correspondingly smaller. It can be noted that a gain from the introduction of the third RMU module is smaller, comparing with the case of introducing a second RMU module. It results from the fact that in the first case half of requests is handled by the newly connected RMU module, while in the second

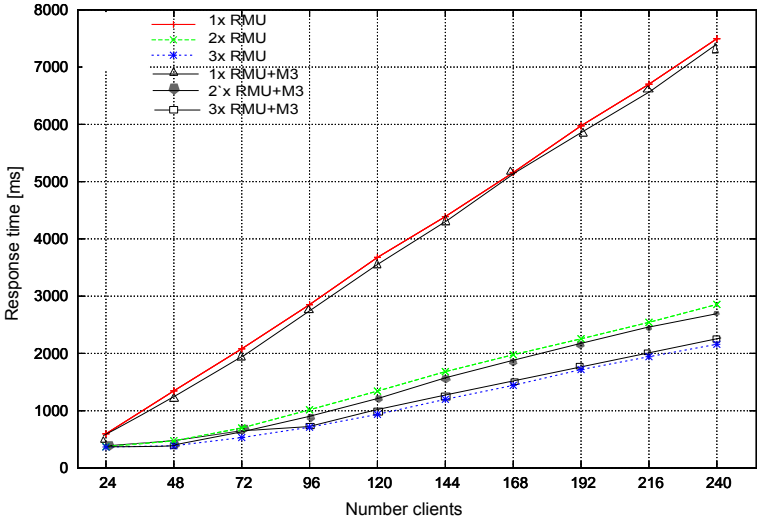


Figure 3: Processing overhead introduced by ReServE during failure-free computing

case, only 1/3 of all requests. Moreover, the benefits of offloading the stable storage are smaller if it works under a small load from the very beginning, like in the case when 2 RMU already exist.

The interesting results are obtained for the solution where ReServE is integrated with M^3 service. It was observed that the decrease in the processing time related with increasing number of services served by ReServE did not occur, contrary to the solution without M^3 service. Due to that fact, the general performance of ReServE integrated with M^3 service has not decreased in the case of failure-free processing, despite the fact that M^3 introduces the additional processing related with monitoring, The differences in the overhead introduced by ReServE environment, and by the environment with the same RMU numbers integrated with M^3 service can be ignored, because the decrease in processing time due M^3 introduction is balanced by the increase of processing time due to the lack of RMU overloading. Such a result is related to the established way of the integration, and the necessity of the communication with M^3 only at the beginning of ReServE work.

The purpose of the next simulation experiments was to examine the recovery time of the processing depending on the configuration of the environment. The obtanied results are presented in the table 3 and illustrated in Figure 4. The results show the total recovery time (in seconds) for each of the adopted architectures. As previously, we consider ReServE environment with 1, 2 and 3 RMU modules respectively, and ReServE with 2 and 3 RMU modules integrated with the M^3 service.

In the simulation experiments, the number of clients, which simultaneously use the ReServE environment does not affect the simulation results. However, the impact

Req.	1RMU	1RMU+M3	2RMU	2RMU+M3	3RMU	3RMU+M3
2000	186,59	186,00	46,22	38,82	31,30	27,56
4000	371,68	372,83	79,44	70,40	60,40	47,16
6000	556,49	557,92	128,45	113,59	90,33	71,75
8000	741,85	742,53	165,27	149,76	117,18	89,94
10000	928,18	930,28	204,80	170,83	149,31	110,38

Table 3: Recovery time [s] in different ReServE environment configurations

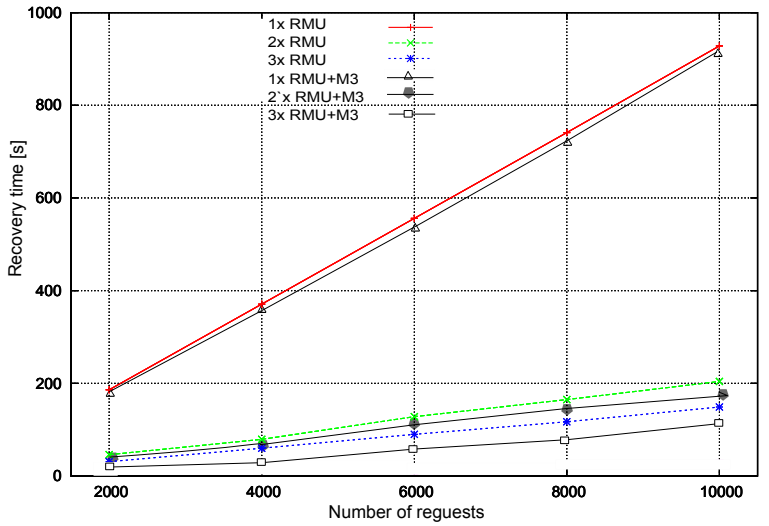


Figure 4: Business process recovery time in in different ReServE environment configurations

on the obtained results has the number of requests that have to be replayed. The recovered requests are read from the stable storage, and send by *RMU* module to *SIM* of the recovered service. The obtained results show that existence of multiple, distributed *RMU* modules improve the work of ReServE environment, which is related with the fact that recovery procedue is performed concurrently, and each *RMU* module has to replay less requests to recover a consistent processing state. The results of the carried out tests show also that monitoring of *RMU* load results in further reduction of the overhead introduced by the recovery procedure. It is related with fact that *RMU* modules have to recover less requests, so the process of obtaining them from the stable storage and relaying to appropriate *SIM* modules is more efficient.

8. Conclusions and Future Work

In this paper, we have discussed the ways to improve the efficiency of ReServE environment, which aims at increasing reliability of service-oriented systems. Since reliability issues are particularly important and imperative in SOA, we decided to provide the solution that focuses on enhancement of ReServE functionality. For this purpose we integrated ReServE with M^3 service, being a part of Dynamic Management SOA Toolkit (DyMST). M^3 monitors and controls crucial components of SOA-based systems. In the case of ReServE the monitoring pertains to *RMU* modules.

Due to its flexible architecture and design, M^3 is capable of monitoring *RMU* modules without significant modifications in the recovery protocol applied in ReServE environment. The integration of ReServE and M^3 enables to monitor and measure the quantity of work that *RMU* modules carry out and accordingly to obtained results it proposes new ReServE participants to register in the *RMU* module, which burden is the lowest.

In the paper we have discussed the possible alternative ways of ReServE monitoring, and presented the chosen solution. The proposed solution has been experimentally evaluated. The obtained results confirm the successful deployment of integration of ReServE and M^3.

During performed simulation experiments the metric, which determines the system load was considered. As compelling problem is the answer to the question what needs to be monitored to obtain the most valuable knowledge on the *RMU* burden. Thus, in the future work we will propose other monitoring metrics, and analyse their impact on the ReServE functionality in the simulation tests. A valuable element of the future work would be also the assessment of costs related to the implementation of alternative monitoring solutions and possible ways of integration of ReServE and M^3 discussed in this paper. In particular, the experimental evaluation of the impact of dynamic changeover of default *RMU* modules on the quality of processing can be interesting.

References

[1] Armbrust M., Fox A., Griffith R., Joseph A. D., Katz R., Konwinski A., Lee G., Patterson D., Rabkin A., Stoica I., et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] Avizienis A., Laprie J.-C., Randell B., and Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004.

[3] Biyani K. N. and Kulkarni S. S. Assurance of dynamic adaptation in distributed systems. *J. Parallel Distrib. Comput.*, 68(8):1097–1112, 2008.

[4] Brzeziński J., Danilecki A., Hołenko M., Kobusińska A., Kobusiński J., and Zierhoffer P. D-reserve: Distributed reliable service environment. In *Proceedings of the 16th East European Conference on Advances in Databases and Information Systems*, ADBIS'12, pages 71–84, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Brzeziński J., Dwornikowski D., Kobusińska A., Kobusiński J., Sajkowski M., Sobaniec C., Szychowiak M., Wawrzyniak D., and Wojciechowski P. T. Dependability infrastructure for SOA applications. In Ambroszkiewicz S., Brzeziński J., Cellary W., Grzech A., and Zieliński K., editors, *Advanced SOA Tools and Applications*, Studies in Computational Intelligence, vol. 499, pages 203–260. Springer Berlin Heidelberg, 2014.

[6] Brzeziński J., Dwornikowski D., and Kobusiński J. FADE: RESTful service for failure detection in SOA environment. In Malyshkin V., editor, *Parallel Computing Technologies*, volume 6873 of *Lecture Notes in Computer Science*, pages 238–243, Kazan, Russia, Sept. 2011. Springer Berlin.

[7] Chen C., Zaidman A., and Gross H. A framework-based runtime monitoring approach for service-oriented software systems. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications, QASBA 2011, Lugano, Switzerland, September 14, 2011*, pages 17–20, 2011.

[8] Drusinsky D. Run-time monitoring using bounded constraint instance discovery within big data streams. *ISSE*, 12(2):141–151, 2016.

[9] Elmootazbellah N., Elnozahy, Lorenzo A., Wang Y.-M., and Johnson D. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[10] Fielding R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] Funika W., Godowski P., Pegiel P., and Król D. Semantic-oriented performance monitoring of distributed applications. *Computing and Informatics*, 31(2):427–446, 2012.

[12] Hołenko M., Kobusińska A., Wawrzyniak D., and Zierhoffer P. The impact of service semantics on the consistent recovery in SOA. In *Proc. of the 12th IEEE International Symposium on Parallel and Distributed Processing with Applications*, ISPA'14, pages 109–116, Milano, Italy, Aug. 2014. IEEE Computer Society.

[13] Huhns M. N. and Singh M. P. Service-oriented computing: Key concepts and principles. volume 9, pages 75–81. IEEE, 2005.

[14] Kobusińska A. and Hsu C.-H. Towards increasing reliability of clouds environments with restful web services. *Future Generation Computer Systems*, page in press, 2017.

[15] Lahami M., Krichen M., and Jmaïel M. Runtime testing approach of structural adaptations for dynamic and distributed systems. *IJCAT*, 51(4):259–272, 2015.

[16] Lim A. S. Automatic analytical tools for reliability and dynamic adaptation of complex distributed systems. In *1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '95), November 6-10, 1995, Fort Lauderdale, Florida, USA*, pages 1–8, 1995.

[17] Mell P., Grance T., et al. The NIST definition of cloud computing, computer security division, information technology laboratory, national institute of standards and technology gaithersburg. 2011.

[18] Moradi F., Flinta C., Johnsson A., and Meirosu C. Conmon: An automated container based network performance monitoring system. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon, Portugal, May 8-12, 2017*, pages 54–62, 2017.

[19] Najem M., Benoit P., Ahmad M. E., Sassatelli G., and Torres L. A design-time method for building cost-effective run-time power monitoring. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(7):1153–1166, 2017.

[20] Newcomer E. and Lomow G. *Understanding SOA with Web services.* Addison-Wesley, 2005.

[21] Neykova R., Bocchi L., and Yoshida N. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.

[22] Psaier H., Juszczyk L., Skopik F., Schall D., and Dustdar S. Runtime behavior monitoring and self-adaptation in service-oriented systems. In *Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2010, Budapest, Hungary, 27 September - 1 October 2010*, pages 164–173, 2010.

[23] Psiuk M. and Zielinski K. Goal-driven adaptive monitoring of SOA systems. *Journal of Systems and Software*, 110:101–121, 2015.

[24] Richardson L. and Ruby S. *RESTful Web Services.* O'Reilly Media, 2007.

[25] Schall D., Truong H. L., and Dustdar S. Unifying human and software services in web-scale collaborations. *IEEE Internet Computing*, 12(3):62–68, 2008.

[26] Thomas E. *SOA Principles of Service Design.* Prentice Hall PTR, 2007.

[27] Wagner S., Fehling C., Karastoyanova D., and Schumm D. State propagation-based monitoring of business transactions. In *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taipei, Taiwan, December 17-19, 2012*, pages 1–8, 2012.

[28] Zoraja I., Trlin G., and Sunderam V. S. Eliciting the end-to-end behavior of SOA applications in clouds. *Computing and Informatics*, 35(2):259–281, 2016.

Received 27.02.2018, Accepted 25.04.2018