Computing and Informatics, Vol. 35, 2016, 259–281

ELICITING THE END-TO-END BEHAVIOR OF SOA APPLICATIONS IN CLOUDS

Ivan Zoraja, Goran Trlin

University of Split, Faculty of Electrical Engineering Mechanical Engineering and Naval Architecture
R. Boškovića 32, 21000 Split, Croatia
e-mail: {zoraja, trlin}@fesb.hr

Vaidy Sunderam

Emory University,
Department of Mathematics and Computer Science
400 Dowman Dr, Atlanta, GA 30322
e-mail: vss@emory.com

Abstract. Availability and performance are key issues in SOA cloud applications. Those applications can be represented as a graph spanning multiple Cloud and on-premises environments, forming a very complex computing system that supports increasing numbers and types of users, business transactions, and usage scenarios. In order to rapidly find, predict, and proactively prevent root causes of issues, such as performance degradations and runtime errors, we developed a monitoring solution which is able to elicit the end-to-end behavior of those applications. We insert lightweight components into SOA frameworks and clients thereby keeping the monitoring impact minimal. Monitoring data collected from call chains is used to assist in issues related to performance, errors and alerts, as well as business and IT transactions.

Keywords: Cloud computing, distributed systems, SOA, availability, performance, real-time monitoring, APM, BTM

Mathematics Subject Classification 2010: 68U99

1 INTRODUCTION

Like any new technology, cloud computing brings many benefits and issues [1, 2] for its users and vendors. Users reduce their needs for on-premises resources (computers and IT professionals) and are not concerned with local deployments. They can access applications via the internet and pay only for usage on a per-month or per-year basis. The financial risk for users is lower since they can try the software before buying it. The vendors get a more flexible market since they can easily attract new customers, sell directly to decision makers, and gain a more predictable profit. The vendors can easily update applications, share applications among multiple users, and gain a better understanding about the usage patterns.

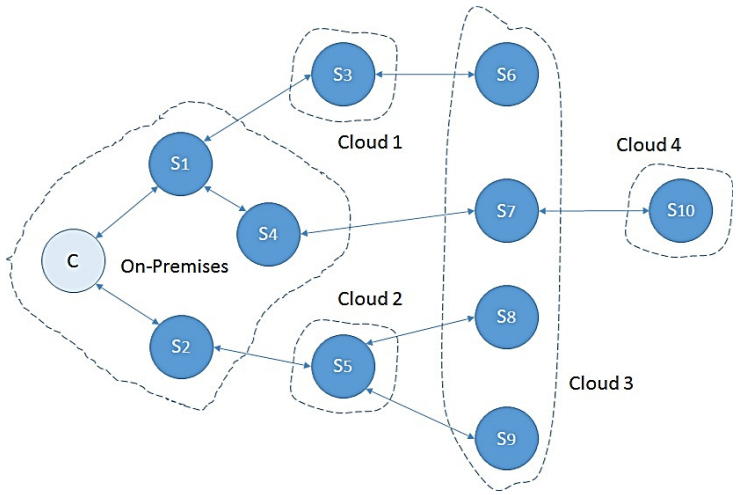


Figure 1. A call chain across multiple clouds and on-premises

SOA applications consist of *services* which expose their functionality, *clients* consuming that functionality, and *channels* used to exchange messages. With reference to Figure 1, a service can reuse other services in a *call chain* to implement own functionality, so an SOA application can be represented via a *graph* with services (S) and clients (C) as vertices and communication routes as edges. Modern SOA applications may span multiple clouds and on-premises centers forming very complex computing ecosystems [3, 5] that support more and more users running more and more transactions in perplexed usage scenarios. Research community invests significant effort on providing insights into the run-time behavior of such complex systems [4].

In addition to data security, *availability* and *performance* are the primary concerns of cloud users and vendors. Therefore, they need efficient and versatile tools to online monitor the end-to-end behavior of cloud applications for rapid finding, predicting, and proactively preventing root causes of issues. The monitoring sup-

port [6, 7] for both production and development phases is of great benefit for business critical and high-performance applications since they not only have to be available $24 \times 7 \times 365$ but also must run flawlessly and efficiently for all users at all times.

Our real-time monitoring approach called cSOOM (Cloud Service-Oriented Online Monitor) is able to assist in many critical issues in the development, testing, deployment, and production phases of SOA applications. The possibility to visualize interoperable services and application transactions, to identify and predict performance bottlenecks and locate runtime errors, to observe IT transactions and content is a powerful approach that will address many today's challenges in the realm of SOA applications deployed on-premises and in the clouds.

The next subsection provides an overview of related monitoring approaches and positions the niche cSOOM focuses on. The Section 2 discusses both the functional and non-functional requirements of cSOOM, while Section 3 describes its software architecture with an accent on techniques for inserting custom code into SOA frameworks. Section 4 provides an insight into visualization approaches while Section 5 deals with errors and availability of single services and call chains. In Section 6 we turn our attention to the performance of a single service and the one of call chains. Finally, we finish off with Section 7 which concludes the paper, discusses our findings and results, and gives suggestions for future work.

1.1 Background and Related Work

Online (*real-time*) monitoring refers to a set of techniques to allow both the observation and manipulation of running systems, especially of their dynamic behaviours. The monitoring functionality can be implemented at various software and hardware levels. Software monitoring is more flexible to change, requires less effort to be ported on other platforms but has an impact on the system being monitored. On the other hand, hardware monitors [28] can be implemented with little or no impact on the system being instrumented, but they are difficult to change and are very hard to port to new platforms. Generally, hardware approaches are used to monitor fine-grained entities such as registers and memory locations, while software monitoring instruments the coarse-grain entities such as objects and services. Hybrid implementations try to balance these issues.

Software monitoring techniques have been applied to various types of software systems but the most challenging software systems to be monitored are distributed systems [25, 24] since they exhibit complex communication and interaction patterns. Distributed processes usually reside in different address spaces and communicate by exchanging messages. On the top of those facilities are implemented message passing, distributed shared memory, distributed object, and SOA paradigms. Techniques and approaches for monitoring messages passing systems, such as PVM and MPI, are extensively investigated in [26, 27]. Real-time monitoring of distributed shared memory systems, such as TreadMarks and Orca, are described in [8, 9] while monitoring of distributed object-based systems, such as CORBA and DCOM, are discussed in [22, 23].

The widespread use of SOA applications, especially of those deployed across heterogeneous cloud platforms, have highlighted the need for powerful monitoring systems and tools that can support and automate all phases of their software development processes. In order to ensure the needs of SLAs, an agent-based approach of monitoring the non-functional requirements of heterogeneous services deployed on-premisses is described in [29]. Several vendors apply their general monitoring solutions [30, 31, 32, 33] to work in the SOA world. A comparative analysis of such approaches with SOOM tools are given in [9]. While SOOM focuses on monitoring SOA application deployed on-premisses, cSOOM is designed to monitor SOA applications in heterogenous clouds and is centered around a notion of a call tree that spans multiple clouds.

2 MONITORING REQUIREMENTS

The quality of SOA applications as well as the enterprise overall productivity and revenue is directly affected by issues in the applications. The main purpose of real-time monitoring [8, 9] is to elicit (measure, quantify) non-functional (quality) requirements of running applications. The capabilities and conditions to which monitoring approaches must adhere are driven by the perceived needs of the main SOA actors: *users*, *operators*, *business managers*, *programmers*, and *testers*. We monitor SOA applications running in heterogeneous environments and our primary objects being monitored are clients and services forming a *graph* which for a single client call, in many practical cases, results in a *tree*.

We perform *application-level* monitoring since all collected data reflects the runtime behavior and can be related to the constructs understood by the actors. In contrast, component-based monitoring reports averages of individual components (e.g. database) and low-level monitoring (e.g. network) collects data that cannot be easily related to the application constructs. Due to its flexible and application centric design and architecture where all instrumentations and measurements are implemented at the application level, cSOOM is fully capable of monitoring heterogeneous SOA application in various cloud environments: *SaaS*, *IaaS*, and *PaaS*.

With reference to Figure 2, in an SOA application we recognize service and client side objects to be monitored. Service side objects include *services*, *endpoints* and *operations*, while client side objects revolve around *proxies*. Services expose their *operations* via one or more endpoints and each endpoint provides multiple operations callable by clients. Each endpoint can be accessed via multiple communication protocols, that exchange *messages* implementing SOAP or REST architectural styles. A service *description*, usually provided in a language such as WSDL, describes service operations and provides a reference for generating client side proxies. They are client side objects intended for managing client calls and underlying message exchange patterns (MEPs). Usually, proxies encapsulate all service operations into a single cohesive class. *Stubs* or *dispatchers* handle messages on the service side.

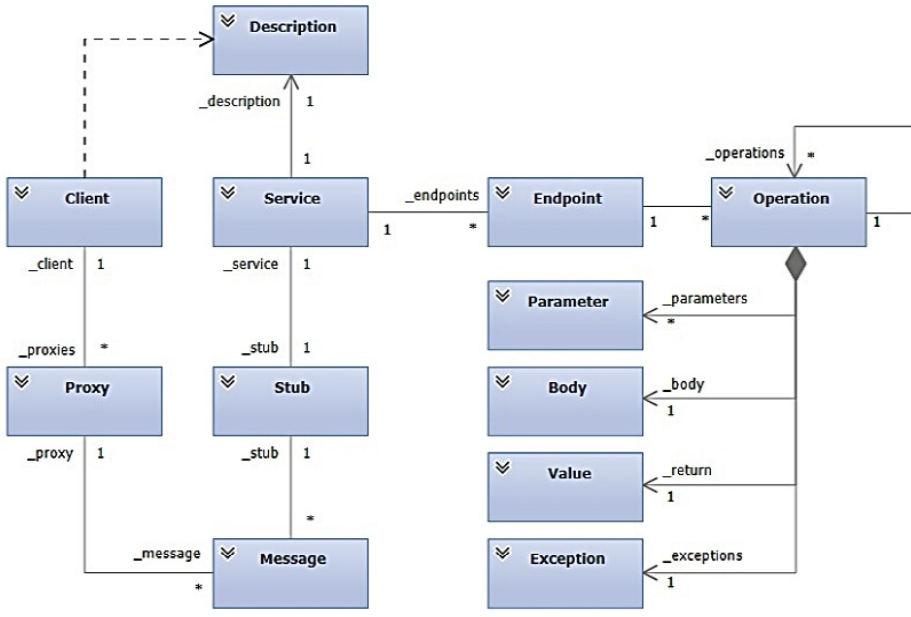


Figure 2. Objects being monitored

3 LOGICAL ARCHITECTURE

To demonstrate our concepts and ideas as well as to explore monitoring potentials for SOA applications we developed a prototype (cSOOM) as a real-time monitoring system. We also developed some rudimentary tools to visualize, analyze, and correlate the collected monitoring data. The logical architecture of cSOOM is based on the SOOM architecture [7]. While SOOM is a versatile and powerful platform for online SOA monitoring, cSOOM expands on this idea and puts cross cloud systems and their corresponding call trees as its fundamental concepts. This concept is necessary to successfully and efficiently elicit the end-to-end runtime behavior of cross cloud systems, since they are mutually dependent and exhibit complex interaction patterns.

3.1 Components and Connectors

cSOOM itself is an SOA application. With reference to Figure 3, the cSOOM **Server** acts as a managing component that controls multiple cSOOM **Agents** and stores relevant monitoring data in a database. An Agent is designed according agent-oriented architectures. It manages a group of clients and services that may reside on a single computer, network, or a Cloud. The core monitoring functionality is implemented via lightweight components (cSOOM **Intruders**) inserted into

SOA frameworks thereby keeping the monitoring impact minimal. Using cSOOM **Tools**, operators invoke monitoring actions on the objects being monitored. These commands are converted into SOAP or REST messages and distributed to the corresponding Agents.

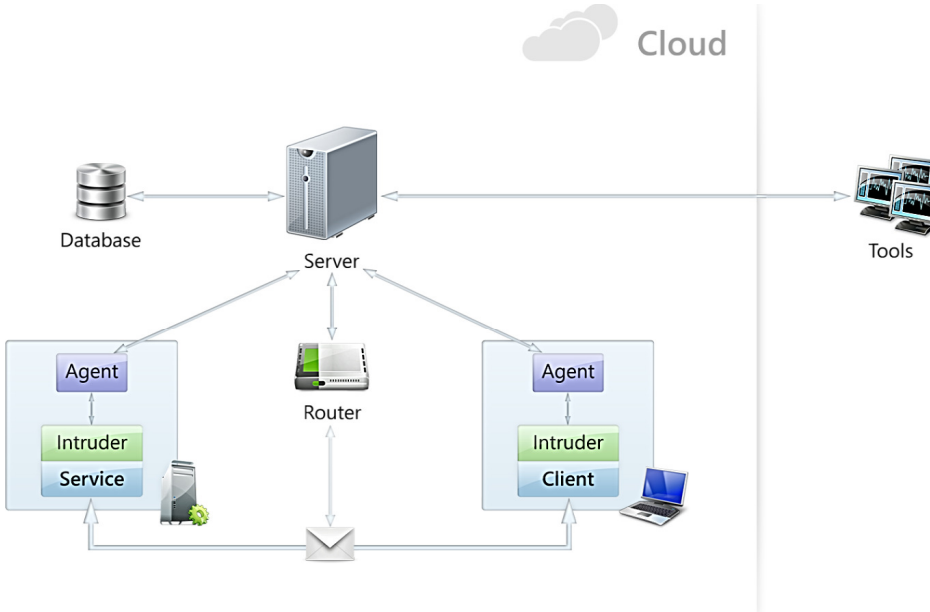


Figure 3. The logical architecture of cSOOM

The Agents delegate requested monitoring actions to the Intruders which are responsible for intercepting SOA messages, executing monitoring actions, and handling events. For SOA applications based on the REST architectural style we reduce the number cSOOM components to be deployed by integrating the Agent functionality with the Server code. The cSOOM **Router** redirects messages depending on a set of predefined rules. That way, cSOOM performs load balancing in systems under heavy load and therefore helps achieve stability and scalability [10]. The criteria for balancing can be either performance data of a single service or a call chain as well as the semantic of the service call being invoked. The communication between cSOOM components is implemented via *asynchronous messaging*.

3.2 Injecting Monitoring Code

To insert custom code into SOA services and clients we utilize various aspect-based techniques such as *extension points* for Windows Communication Foundation (WCF) services [12] and *dynamic proxies* for Java services deployed on the OSGI Framework [20]. Intruders read incoming messages and add *custom headers*

to outgoing messages. cSOOM makes use of the Lamport algorithm [11] to determine temporal order among events generated by the Intruders and reconstruct SOA call trees. *Logical clock* is maintained by injecting logical clock values in outgoing monitoring requests. That way, cSOOM gains complete control over all messages exchanged in the SOA application being monitored.

3.3 Instrumenting .NET Services

Since we used WCF as an SOA framework in many our test cases, here we provide a brief overview of extension points where our core monitoring code (Intruders) is injected in WCF proxies and stubs (dispatchers) at start-up. In WCF terminology, an extension is an object the class of which implements a required interface. In a call stack extensions are executed in the predefined order. The WCF proxy supports three extensions, as follows:

1. *Parameter inspection* – allows developers to inspect and modify parameters being sent to the service and those received from the service.
2. *Message formatting* – enables developers to perform custom serializations into the message object.
3. *Message inspection* – provides a last chance to modify the message object before it is supplied to the channel stack. After the code at the last extension is executed, the message object is supplied to the channel stack for delivering.

The WCF dispatcher supports five extension points. Three of them are aimed at matching the proxy's extension points and the additional two are used for the following purposes:

1. *Operation selection* – enables developers to select an operation in an endpoint.
2. *Operation invoking* – allows developers to implement custom invocation mechanisms.

An Intruder is packaged as a standard DLL. It is installed via the target applications configuration file, by adding specific XML markup, and loaded dynamically upon startup of the service (or client) being monitored. Intruders are executed when the corresponding extension points in the WCF execution stack are reached.

3.4 Instrumenting Java Services

The OSGi technology provides a set of specifications for creating dynamic components for Java applications which communicate via services. Since an OSGi application is composed of multiple *bundles* we implement our intruding component as a bundle. The Intruder contains functionalities that implement dynamic proxies, service registering, service interception, and the communication with the Agent.

A dynamic proxy is an instance of class `java.lang.reflect.Proxy`, and is used to implement interfaces that are defined at runtime. Service methods are invoked using mechanisms available in *Java Reflection*. When a service is registered by the OSGi framework event `ServiceEvent.REGISTERED` is raised.

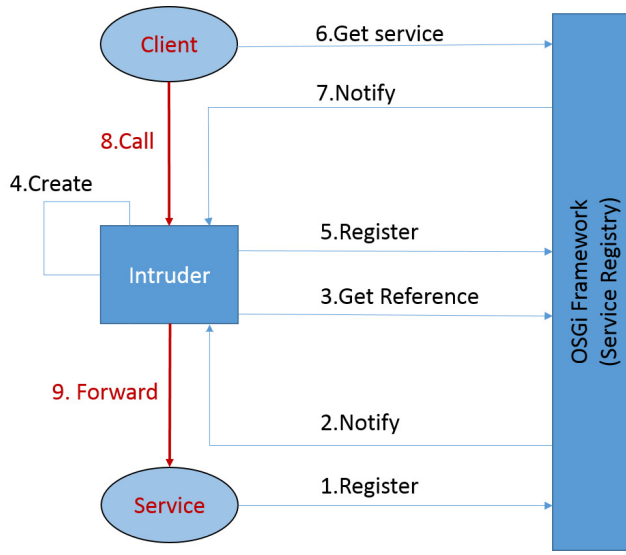


Figure 4. Intercepting OSGi Services

Figure 4 shows the interaction pattern when a service call is intercepted [34]. First, the service bundle being monitored registers itself by the OSGi framework which notifies the Intruder about that event. The Intruder obtains the reference to the service, adds a dynamic proxy to the service, and registers them by the framework. When a client requests a service reference the framework notifies the Intruder about the request and the Intruder returns a reference to the service with the dynamic proxy. Upon receipt of the requested service the client calls an operation on the service. The Intruder intercepts that call, forwards the call to the service, and returns the result back to the client. When an Intruder is invoked it creates a communication route to the Agent using information from headers inserted by cSOOM in SOAP and REST messages.

4 VISUALIZATIONS AND VIEWS

In order to visualize complex SOA applications and monitor their runtime behavior we develop some rudimentary tools that are integrated in the cSOOM Dashboard or implemented as Web or Microsoft Excel applications. Tools are classified according monitoring aspects such as errors, transactions, or performance. Figure 5 shows an SOA graph (depicted in Figure 1) being monitored in the cSOOM Dashboard

(desktop application) in the *conceptual mode*. In the *map* mode the Dashboard shows SOA components on their geographical locations.

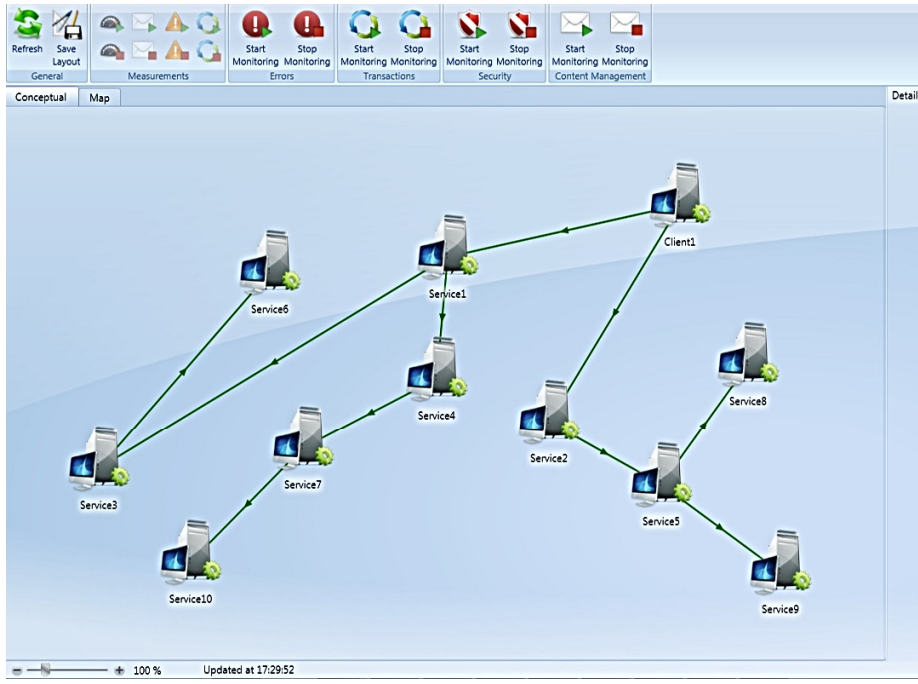


Figure 5. The cSOOM Dashboard

The graph is dynamically constructed and the Dashboard depicts the services, their consumers, and communication routes being monitored by cSOOM. Connections between services and clients being monitored are displayed using lines that change color depending on the state of the connection. If the connection is active, a line is green and when the connection becomes inactive, the line turns black. Also, there is an arrow in the middle of the line which shows the communication direction. A plug-in version of the Dashboard can be added to the MS Visual Studio 2013 to support the testing of WCF clients and services.

The cSOOM tools support four types of views on the SOA graph being monitored:

1. the *service-based view*,
2. the *metadata view*,
3. the *transaction-based view*, and
4. the *content view*.

In the service-based view, both the services and clients are represented as vertices in a graph while communication routes are depicted as edges. The details for the selected clients, services, and connection channels can be shown in separate panes. To get a better insight into behaviour of a large number of nodes the Dashboard provides controls to zoom in, zoom out, and to save the current graph.

SOAP based services expose their interfaces in a high level language called WSDL which is a platform-independent description (*metadata*) of a service including ABC (*address*, *bindings*, *contracts*) and information about *security*, *transactions*, *reliability*, and *faults*. With reference to Figure 6, the Metadata Viewer is a cSOOM tool that is able to visualize endpoints of a running service (`PatientInfoService`) using metadata exchange mechanisms. For example, metadata for WCF services is obtained via the `MetadataExchangeClient`, `MetadataImporter`, and `WsdlImporter` components. Metadata can be used to auto generate client proxies or extract other useful information about the service under monitoring.

Transactions are one of the most important aspects of any SOA application. cSOOM is capable of monitoring both *IT* and *business* transactions. Sequence and communication diagrams, in an UML style, are automatically generated in the *transaction-based view* when cSOOM detects operation invocations among the nodes under monitoring. The transaction's resulting state (committed, aborted, in-doubt) is reported to the cSOOM operator in real time. SOA messages transfer business data (content) among components of the SOA graph. Via its Intruders cSOOM extracts business information from request and response messages and stores content data into a database or files. Complex data types, which are passed and returned in call chains, can be displayed in a hierarchical fashion using the name/value semantics and used to improve business processes [13].

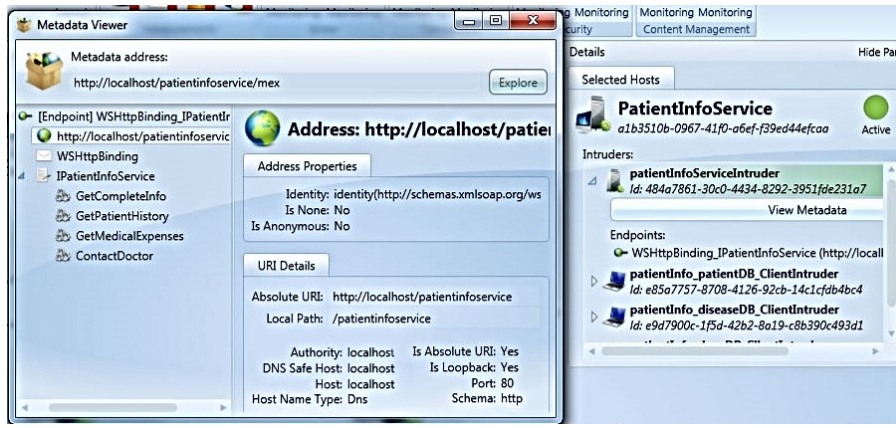


Figure 6. The Metadata Viewer

5 MONITORING AVAILABILITY

To elicit the run-time behaviour of an application and to verify the architecture (non-functional requirements), the users utilize various tools and techniques to measure, analyze [26], and visualize [14] the application performance and availability. SOA users expect 100% uptime from software they use. In practice, this goal is almost never reached due to numerous factors such as broken network connections, exhausted disk space, and high load. We define availability as a ratio between the *up time* and the total time. The *down time* is usually caused by some unexpected events. We refer to them as application errors since most of them can be detected at the application level. cSOOM can, in a proactive fashion, detect services that are not working and automatically notify the interested parties.

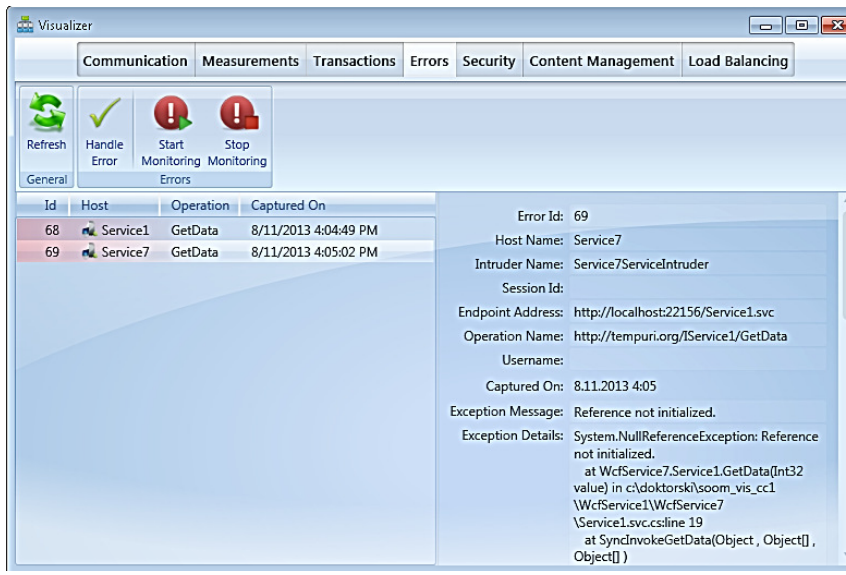


Figure 7. Monitoring errors

To demonstrate the cross cloud monitoring abilities of cSOOM in respect to the end to end behavior of SOA applications we tested the following monitoring features: **errors**, **performance**, and **transactions**. With reference to Figure 1, our test-bed consisted of ten services deployed in four clouds (*Amazon Web Services EC2 Virtual Machines*) and a client (*Android v4.0 phone*), connected via WLAN to the on-premises. Our services make use of a `sleep` function to produce work the performance range of which can be expected. Results provided here were obtained by the cSOOM tool suite.

5.1 Monitoring Errors and Alerts

Error detection and reporting techniques, also referred to as *deep diving monitoring*, provide the programmers with low level details about the application misbehavior. Runtime exceptions and security violations are the two essential types of errors in SOA applications. cSOOM successfully deals with both types of errors by detecting errors at run-time at the precise location of the issue using inspection mechanisms available in the underlying SOA frameworks and implemented by Intruders. cSOOM catches the exception, marks the faulting node problematic (the exclamation point in the Dashboard), and stores the complete information about the issue in the database at the Server. By analyzing the entire call tree invocations the user can discover how the particular error was propagated and what was the real cause of the problem.

False credentials, reply attacks, eavesdropping, etc. are common security threats in SOA applications. To obtain security related data cSOOM makes use of the security and auditing mechanisms provided by the underlying SOA frameworks (e.g. WCF and Java WS). Security threats are identified at real-time and acted upon immediately via the alerting mechanisms by notifying the observers via e-mail or SMS messages. Root cause analysis is significantly simplified, as cSOOM points out to the particular service operation which has caused the error. An example of a captured error is depicted in Figure 7. System developers are automatically provided with the complete distributed stack trace of the application being monitored. This enables SOA developers to quickly resolve the issues and prevent further problems, leading to reduced down times and other end-user experience related issues.

5.2 Monitoring Transactions

An error on a particular service node could not only reduce the availability of that operation but could also affect one or more related operations to fail. A group of SOA operations to be executed in a call chain are referred to as an SOA transaction. We differentiate between business and IT transactions. Both transaction types represent a group of distributed operations but IT transactions have the well-known property: all its operations must succeed or fail as a group. An IT transaction can end in two ways: with a *commit* or a *rollback*. When commits, the modifications made by its operations are saved. If an operation within a transaction fails, the transaction must be rolled back, undoing the effects of all operations in the transaction. The third state, *in-doubt*, is a temporary state which can happen due to the errors in the transaction managers and coordinators.

In Figure 8 we show a committed IT transaction as a call chain in the cSOOM Dashboard via a sequence diagram – note that some services are not shown since the picture would be too wide for this type of presentation. In the header we show `transaction identifiers`, the `initiator` of the transaction which is in this case the client, the `transaction protocol`, and the `time` when the transaction

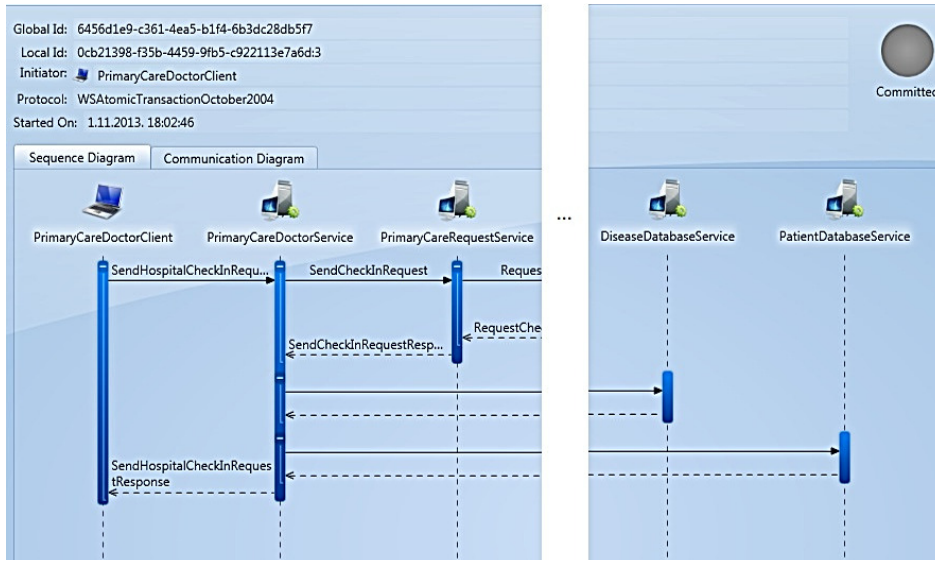


Figure 8. Monitoring distributed IT transactions

is triggered. The diagram dynamically shows the calls and returns of operations that participate in the transaction. To monitor IT transactions cSOOM makes the following assumptions about the transaction support from the underling SOA framework:

1. Services must be **transaction aware**, meaning be able to react on rollback events and revert back to the original state.
2. Transactions are executed with the help of transaction coordinators, such as DTC (Distributed Transaction Coordinator), which must handle low level communication to implement the 2PC (Two Phase Commit Protocol).

6 MONITORING PERFORMANCE

An SOA application can be depicted as a directed graph where vertices represent both services and clients while edges represent communications routes used to exchange messages. With reference to Figure 9, a call chain can be represented as a single tree of execution starting when a client calls an operation on a service which, in turn, calls operations on other services, possibly located on another machine or cloud, to complete the requested task. A client is the root node of an SOA call tree since it starts the invocation chain by calling an operation on a service. It can be implemented as a desktop, web, or a mobile application. A service can be a composite or a leaf. In order to implement the requested functionality, a compos-

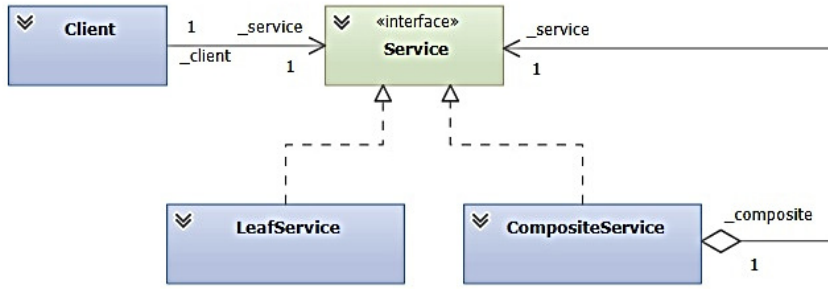


Figure 9. The call tree as the composite pattern

ite service may delegate calls to other services while leaf services do not delegate calls.

6.1 Performance Metrics

From technological point of view, the basic performance metrics of an operation are the call *duration* and *frequency* [16, 18]. Call durations impact transaction times while call frequencies determine throughput. From end users perspective, the main measure of system performance is the transaction time [19, 17]. Once instrumentation is successfully performed, calculating call durations and frequencies can be done on each node by measuring certain time spans, as shown in Figure 10. Performance data necessary for the calculation of call durations is defined as a set of points in time, as follows:

x – is the moment when a node starts receiving the request,

z – is the moment when a node starts sending a request,

w – is the moment when a node received the response,

y – is the moment when a node sent the response.

We make use of the following times in an SOA call chain:

Execution time (E_i) – is the time elapsed between the moment node i starts receiving a request and the moment it sent the response.

Processing Time (P_i) – is the time spent by node i on computation.

Delegation time ($D_{i\to j}$) – is the time spent by node i on waiting for service j to finish its computation.

Latency Time ($L_{i\to j}$) – is the time elapsed between the moment a node i starts sending the request and the moment node j received the request.

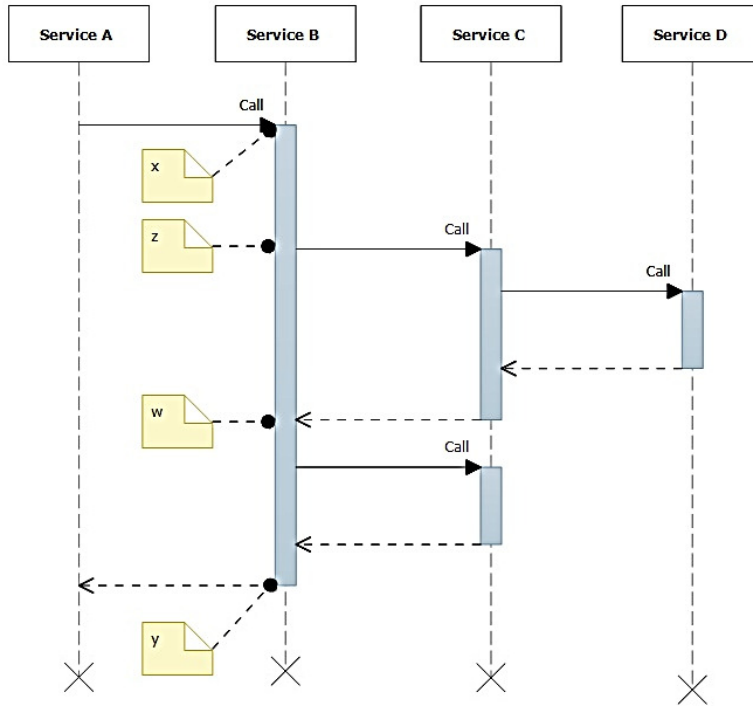


Figure 10. Crucial times in an SOA call tree

The times are calculated as follows:

$$\begin{aligned} E_i &= y_i - x_i \\ D_{i\to j} &= w_{i\to j} - z_{i\to j} \\ L_{i\to j} &= D_{i\to i} - E_i \\ P_i &= E_i - \sum_{j=0}^{M} D_{i\to j} = y_i - x_i + \sum_{j=0}^{M} (w_{i\to j} - z_{i\to j}) \end{aligned} \tag{1}$$

Finally, total execution time (T), the equivalent of transaction time from end users perspective, is defined as:

$$T = \sum_{i=1}^{N} \left(P_i + \sum_{j=1}^{M} L_{i\to j} \right) \tag{2}$$

Taking necessary measurements at each node and using the previous equations, we can reconstruct the complete SOA call tree and determine requested call durations, resulting in the overall transaction time. Gathered data can be further

analyzed using various workload models to identify bottlenecks and fine tune the performance.

The above equations are defined under some constraints. First, we assume that each service node appears only once in the SOA call tree (no indirect recursions are allowed). Although recursion is a useful programming technique, we argue that it is not well suited for service calls – at least at the current state of communication protocols and observed latency times. Second, currently, we only deal with call trees generated by synchronous calls. Finally, we monitor services executed by a single thread. All these constraints will be addressed in our future research on SOA monitoring in cross cloud platforms.

6.2 Measuring Performance

cSOOM measures the performance of a single service as well as the performance of a group of services that form a call chain. For individual operations on a service being monitored, cSOOM records various frequencies and durations, such as the number of calls per second and execution times. In addition, we provide a common analytics for calculating averages and ratios. For example, with reference to Figure 11, an operation `GetDoctorList` participates with 33% in the total call time on the service being monitored. That kind of information can be used to detect performance bottlenecks.

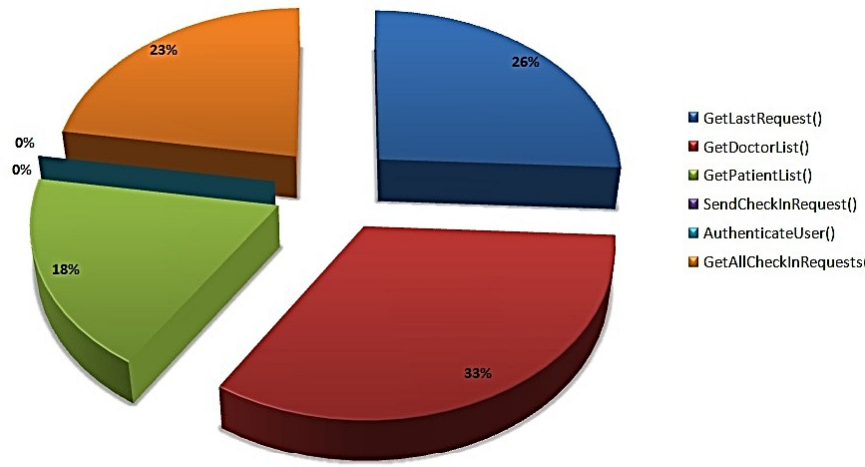


Figure 11. Performance of a single service

In a call chain, both composite and leaf service nodes are implemented to exhibit fixed and random processing times. As shown in Table 1, the mean processing times

per node, obtained in thirty measurements on the SOA graph shown in Figure 1, fall into the expected value ranges. Execution and processing times, obtained in a single measurement, are depicted in Figure 12. It can be concluded that results are as expected – both the execution and processing times are compositely calculated and therefore decrease as the depth level of the node increases. As expected, leaf nodes have the smallest processing and execution times. Also, the processing and the execution times on leaf nodes, such as node 8 and 10, are the same since the leaf nodes do not delegate calls.

Service	**Expected Range**(ms)	**Mean Time** (ms)	**Standard Deviation**
`Service1`	1 500	1 500.44	0.50
`Service2`	750	750.26	0.43
`Service3`	800–900	859.11	26.29
`Service4`	850–900	882.41	12.97
`Service5`	100–900	473.57	241.49
`Service6`	100–200	134.82	24.94
`Service7`	500–600	564.94	32.48
`Service8`	100–800	466.89	113.23
`Service9`	100–800	415.63	107.84
`Service10`	500–1 000	774.43	152.43

Table 1. Expected and measured processing times

7 CONCLUSIONS AND FUTURE WORK

The above concepts for monitoring cross cloud SOA applications and our prototype implementation, called cSOOM, have proved to be both successful and enlightening in eliciting the end-to-end behaviour of those applications. cSOOM performs monitoring actions while SOA applications being monitored are running in heterogenous environments potentially spanning multiple on-premisses and various cloud ecosystems. All collected data reflects the runtime behavior (**errors**, **performance**, and **transactions**) and the complete SOA graph under monitoring can be dynamically visualized via various views (*service*, *metadata*, *transaction*, and *content*). The cSOOM tools can not only issue commands to observe but also to manipulate (change) the state of the objects being monitored.

cSOOM hides the complexity of SOA applications implementing a lightweight and adaptable software architecture in an agile fashion where designated components deal with specific application aspects to provide end-to-end monitoring facilities. For example, the server deals with distribution and parallelism providing actions on services as if they were deployed locally. cSOOM performs *application-level monitoring* managing data that can be related to constructs both the programmers and the end users of an SOA application can understand. In contrast,

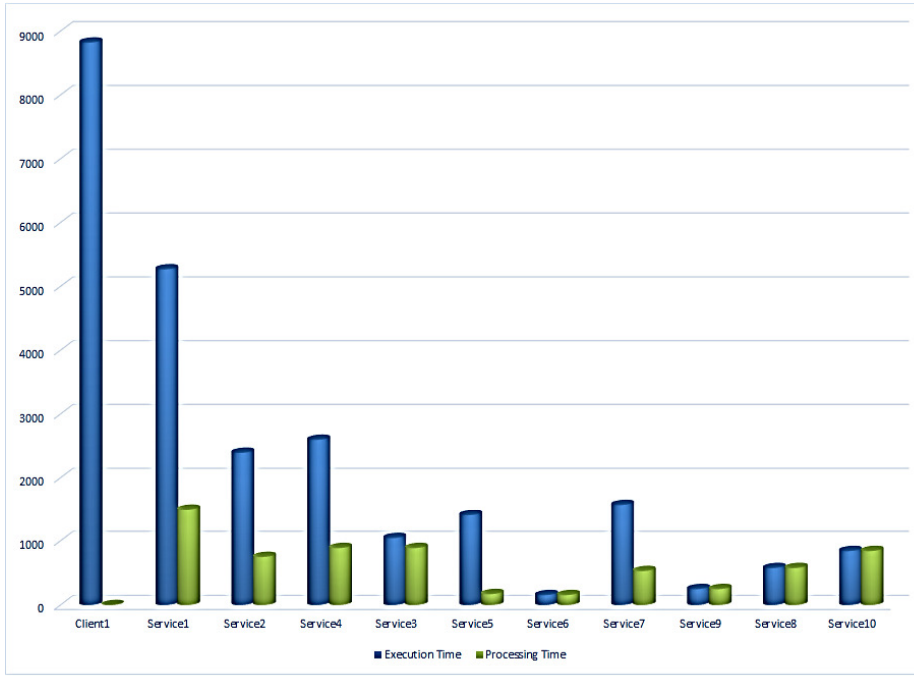


Figure 12. Performance of a call chain

component-based monitoring reports averages of individual components while low-level monitoring approaches collect data that cannot be easily related to the application constructs.

Monitoring actions (requests) are executed in isolation and therefore cannot compromise actions issued by the other tools. cSOOM tools are connected to a single monitoring system which provides a consistent set of monitoring actions where the tools are aware of each other. When, for example, one tool performs an operation on a service all interested tools are notified about that event. The SOA mechanisms provided by underlying SOA frameworks are transparently handled and cSOOM tools can be built without taking care of their particular implementation details. To achieve this, cSOOM inserts own lightweight components into SOA frameworks using various inspection mechanisms that can be applied at the deployment and run times, thereby keeping intrusiveness minimal. Our monitoring approach can be applied in production and development scenarios and assist in improving the application's efficiency and achieving a $24 \times 7 \times 365$ availability.

Since our findings and results have opened new questions and identified new research directions they strongly motivate our further investigations. Firstly, we

will investigate call chains that include asynchronous invocations and parallel executions when a service delegates calls via multiple threads directly or via thread pools. We plan to enhance the monitoring capabilities of IT transactions to support joined transactions that are initiated by coordinators implemented on different IT platforms. A long term research will be performed on discovering and routing strategies with various load balancing techniques based on service content and performance.

Finally, a significant accent in future research will be placed on the REST architectural style and mobile clients, especially on those when multiple mobile clients access a single service deployed in a cloud. Mobile clients are incapable of forming classical peer-to-peer networks making them dependant upon centralized servers. This is due to the fact that mobile clients are almost always behind NATs and firewalls, rendering them incapable of accepting TCP connections from the outside world. Therefore, a two-way communication channel with a mobile client is difficult to establish and maintain. This elevates demands for service availability to a new level, as in case of a single service failure, all clients could suddenly become non-functioning.

Mobile clients are also prone to the changes of geographical positions. Today, it is not uncommon for an end user to travel hundreds of miles in a single day and, therefore, mobile clients are required to be aware of their environment. Applications, and their monitoring agents, must be able to adapt to different network environments, ranging from excellent WiFi connections to no internet signal at all and to provide maximum QoS levels in these varying conditions. Current solutions are mostly limited to providing post-mortem crash report in non-dynamic environments. Mobile clients that invoke cloud services are gaining thousands of users on a daily basis, making companies completely dependent on their flawless functioning, so a sophisticated monitoring support might raise end user experience significantly, while reducing system downtimes and increasing end users' productivity.

REFERENCES

[1] Chappell, D.: The Benefits and Risks of Cloud Platforms – A Guide for Business Leaders. Available on: `http://www.davidchappell.com/` [Accessed, May 2013].

[2] Nguyen, M. B.—Tran, V.—Hluchý, L.: A Generic Development and Deployment Framework for Cloud Computing and Distributed Applications. Computing and Informatics, Vol. 32, 2013, No. 3, pp. 461–485.

[3] Compuware: State of the Art and Trends, Whitepaper, Application Performance Management Best Practices, 2013. Available on: `http://www.compuware.com/en_us/application-performance-management.html` [Accessed, October 2013].

[4] Fiadeiro, J. L.—Lopes, A.: An Interface Theory for Service-Oriented Design. Theoretical Computer Science, Vol. 503, 2013, pp. 1–30.

[5] Menascé, D.—Casalicchio, E.—Dubey, V.: On Optimal Service Selection in Service Oriented Architectures. Performance Evaluation, Vol. 68, 2009, No. 8, pp. 659–675.

[6] Hershey, P.—Runyon, D.: SOA Monitoring for Enterprise Computing Systems. 11th International Enterprise Distributed Object Computing Conference, 2007, pp. 443–450.

[7] Zoraja, I.—Trlin, G.—Matijević, M.: Monitoring SOA Applications with SOOM Tools: A Competitive Analysis. Business Systems Research Journal, Vol. 4, 2013, No. 1, pp. 21–35.

[8] Zoraja, I.: Online Monitoring in Software DSM Systems. Shaker Verlag, Aachen, 2000.

[9] Zoraja, I.—Zulim, I.—Štula, M.: CORAL – Online Monitoring in Distributed Applications: Issues and Solutions. WSEAS Transactions on Computers, Vol. 7, 2008, pp. 113–118.

[10] Macias, E. M.—Sanchez, D.—Suarez, A.—Sunderam, V.: Optimization of Execution Time Inspired Cross Layer Design Using Effective Load Balancing in a LAN-WLAN Environment. International Journal of Computational Science and Engineering, Vol. 4, 2009, No. 3, pp. 182–194.

[11] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21, 1978, No. 7, pp. 558–565.

[12] Lowy, J.: Programming WCF Services: Mastering WCF and the Azure AppFabric Service Bus. 3rd Edition. O'Reilly Media, 2010.

[13] Hernaus, T.—Pejić–Bach, M.—Bosilj–Vukšić, V.: Influence of Strategic Approach to BPM on Financial and Non-Financial Performance. Baltic Journal of Management, Vol. 4, 2012, pp. 376–396.

[14] Bode, A.: Parallel Program Performance Analysis and Visualization. Proceedings of Second Workshop on Environments and Tools for Parallel Scientific Computing, Townsend, Tenessee, 1994, pp. 246–253.

[15] Murarasu, A.—Weidendorfer, A.—Bode, A.: Workload Balancing on Heterogeneous Systems: A Case Study of Sparse Grid Interpolation. 4th Workshop on Unconventional High Performance Computing, 2011.

[16] Zhou, W.—Wen, J.—Gao, M.—Liu, J.: A QoS Preference-Based Algorithm for Service Composition in Service-Oriented Network. Optik – International Journal Light Electron Optics, Vol. 124, 2013, No. 20, pp. 4439–4444.

[17] De, P.—Choudhury, P.—Choudhury, S.: A Framework for Performance Analysis Of Client/Server Based SOA and P2P SOA. 2010 Second International Conference on Computer and Network Technology (ICCNT), IEEE, 2010, pp. 79–83.

[18] Grinshpan, L.: Solving Enterprise Applications Performance Puzzles: Queuing Models to the Rescue. John Wiley & Sons, 2012.

[19] van Hee, K.—Sidorova, N.—Stahl, C.—Verbeek, E.: A Price of Service in a Compositional SOA Framework. Computer Science Report 07/16, Technische Universiteit Eindhoven, 2007.

[20] The OSGi Technology. Available on: `http://www.osgi.org/Technology` [Accessed, 27 April 2014].

[21] Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons, 1991.

[22] Rackl, G.—Ludwig, T.—Lindermeier, M.—Stamatakis, A.: Efficiently Building On-Line Tools for Distributed Heterogeneous Environments. Proceedings of the International Workshop on Performance-Oriented Application Development for Distributed Architectures, Munich, 2002.

[23] Rackl, G.—Lindermeier, M.—Rudorfer, M.—Süss, B.: An Infrastructure for Monitoring and Managing Distributed Middleware Environments, Middleware 2000, April 2000, pp. 71–87.

[24] Nijburg, E. J. H.—Valverde, R.: A Business Continuity Monitoring Model for Distributed Architectures: A Case Study. International Journal of Applied Science and Technology, Vol. 1, 2011, No. 2, pp. 5–14.

[25] Ludwig, T.—Wismüller, R.—Sunderam V.—Bode A.: OMIS – On-Line Monitoring Interface Specification, Version 2.0. Technical report 9. LRR-Techniche Universität München, 1997.

[26] Wismüller, R.—Bubak, M.—Funika, W.: High-Level Application Specific Performance Analysis Using the G-PM Tool. Future Generation Computer Systems, Vol. 24, 2013, No. 2, pp. 121–132.

[27] Wismüller, R.: On-Line Monitoring Support in PVM and MPI. Proceedings of EuroPVM/MPI'98, Springer Verlag, Lecture Notes in Computer Science, Vol. 1497, 1998, pp. 312–319.

[28] Sharifian, H.—Sharifi, M.: Network RAM Based Process Migration for HPC Clusters. Journal of Information Systems and Telecommunication, Vol. 1, 2013, No. 1, pp. 47–53.

[29] Balfagih, Z.—Hassan, M. F. B.: Agent Based Monitoring Framework for SOA Applications Quality. 2010 International Symposium on Information Technology (ITSim), 2010, Vol. 3, pp. 1124–1129.

[30] AppDynamics: The Power of the Business Transaction: The New Way to Manage Application Performance. Business Whitepaper, Available on: `http://www.appdynamics.com/`, October 2011.

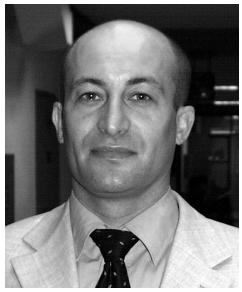
[31] BMC Software: BMC Atrium Discovery and Dependency Mapping. Available on: `http://www.bmc.com/`, June 2011.

[32] IBM: IBM Tivoli Composite Application Manager for Transactions – Guide to Agentless Transaction Tracking. Datasheet. Available on: `http://www-01.ibm.com/software/tivoli/products/composite-application-mgr-transactions/`, August 2012.

[33] Kowall, J.—Cappelli, W.: Gartner's Magic Quadrant for Application Performance Monitoring. 2013.

[34] Banović, J: Monitoring Java SOA Applications. Diploma work supervised by Ivan Zoraja. University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, 2011.

[35] Žmuda, D.—Psiuk, M.—Zieliński, K.: Dynamic Monitoring Framework for the SOA Execution Environment. Procedia Computer Science, Vol. 1, 2010, No. 1, pp. 125–133.



Ivan Zoraja is Professor of Computer Science at University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, Croatia and the Director of company Zoraja Consulting. He received his Ph.D. degree in computer science from the Technical University of Munich, Germany, in collaboration with the Emory University in Atlanta. He was supported by many scholarships during his study, including the Deutscher Akademischer Austausch Dienst (DAAD). His primary research interests are software engineering, heterogeneous distributed and parallel systems, business processes, and 3D rendering algorithms for simulating surgical operations. His scientific research has been supported by grants from the Croatian Ministry of Science, the Croatian Institute for Technology and the Bavarian Ministry of Science. As an external software consultant, architect, and trainer he worked for many European companies including Siemens and Ericsson.

Goran Trlin is a Ph.D. student of computer science at University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, Croatia. He received his Master's degree in computer engineering from the University of Split, Croatia. His primary research interests are software engineering, cloud computing, heterogeneous distributed systems, mobile and web applications, and business intelligence systems. He teaches computer science courses at graduate levels and advises graduate and post–graduate theses in the area of distributed computing, 3D simulations, and software architecture.

Vaidy Sunderam is Samuel Candler Dobbs Professor of Computer Science at Emory University. He is also Chair of the Department of Mathematics and Computer Science, and Director of the University's strategic initiative in Computational and Life Sciences. He joined the Emory faculty in 1986 after receiving his Ph.D. from the University of Kent, England where he was a Commonwealth Scholar. His research interests are in heterogeneous distributed systems and infrastructures for collaborative computing. He is the principal architect of several frameworks for metacomputing and collaboration, and his work is supported by grants from the National Science Foundation and the U.S. Department of Energy. Professor Sunderam teaches computer science courses at the beginning, advanced, and graduate levels, and advises graduate theses in the area of computer systems. He is the

recipient of several recognitions for teaching and research, including the Emory Williams Teaching award, the IEEE Gordon Bell prize for parallel processing, the IBM Supercomputing Award, and an R & D 100 research innovation award.