DISCUSSION ON A FRAMEWORK AND ITS SERVICE STRUCTURES FOR GENERATING JSLEE BASED VALUE-ADDED SERVICES

Thomas Eichelmann, Woldemar Fuhrmann, Ulrich Trick, Bogdan Ghita

Research Group for Telecommunication Networks, University of Applied Sciences Frankfurt/M., Frankfurt/M., Germany
Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, United Kingdom
University of Applied Sciences Darmstadt, Darmstadt, Germany
E-mail: eichelmann@e-technik.org

ABSTRACT

This paper describes the general structure for a framework to generate value-added services. These presented service structures provide the basis for service composition within the JSLEE framework. It offers the possibility that value-added services are created, orchestrated, changed and managed from predefined and pre-deployed service components at runtime, within the JSLEE framework. The paper focuses on different approaches for the structure of service components. Several important characteristics of the structures are compared and discussed.

KEYWORDS

JSLEE, value-added services, service composition in telecommunication, service structure comparison

1. INTRODUCTION

The development of value-added services is very time consuming and experts are required to develop these services. The dream is to develop value-added services as easy as web services in the IT sector. To reach this goal, service description languages and graphical development tools may help. For the IT sector these tools already exist. A common tool to develop web services is e.g. BPEL (Business Process Execution Language) [1]. So, some projects [2] build their own tools for the development of value-added services, others try to use existing tools to generate these services [3].

This document describes the principles for an architecture to generate value-added services and focuses on the description and the analyses of considered component structures. A service can consist of one or more service components. Such a service component implements the service workflow or a part of it. Fine grained service components can be composed to complex services. In this paper JSLEE (Java Service Logic Execution Environment) [4] is used as execution environment for the services. But the discussed characteristics described in section 2.3 may also be relevant for other execution environments like Servlets or JEE (Java Platform, Enterprise Edition) Beans. The paper is structured as follows. In section 2.1 the principle structure for an architecture is defined. Four possible component structures are presented in section 2.2. In section 2.2.1 a structure is introduced which consist of a single component. From this first structure another one is derived, which supports multiple service components for parallel execution. In section 2.2.3 the orchestration of service components and in section 2.2.4 the

choreography of service components are described. In section 2.3 selected characteristics of the described component structures are discussed.

2. FRAMEWORK ARCHITECTURE, UNDERLYING SERVICE STRUCTURE AND SELECTED CHARACTERISTICS

In this chapter the possible architecture for a service creation framework is described and the general elements of the architecture are presented. JSLEE is assumed as execution environment for the services. Four different service structures are presented and selected characteristics are discussed.

2.1. The framework architecture

The architecture that is presented here consists of four main elements, service creation environment, service deployment, service execution environment (SEE) and service transport layer. It is designed for service creation, deployment and execution. This is a generic architecture based on an architecture that is used in the BMBF project TeamCom [3]. In Figure 1 an overview of the generic architecture is illustrated.

The service creation environment consists of the elements, developer GUI and code generator. With the developer GUI the service developer can describe the workflow of the service. This service description has to be translated by a code generator into the language of the service execution environment. In this case Java is used as this language. The service description is translated into the Java language and a deployable unit (DU) is build from the service description. The service can be deployed on one or more application servers. It is executed in the service execution environment. The service components are managed in the component model. In section 2.2 four approaches of the component structure are described.

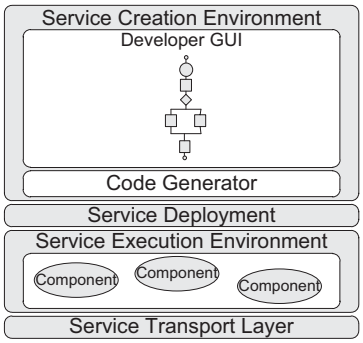

Figure 1. Framework architecture

The intention of the Graphical User Interface (GUI) is to offer the development of value-added services in a simple and fast way. Therefore the framework requires a description language that is simple but powerful enough to describe telecommunication services. Different approaches are known for a usage of a graphical user interface for the description of telecommunication services. Some approaches try to implement their own GUI e.g. [2]; others e.g. [3] use existing graphical development tools. The BPEL process in Figure 2 is an example, how value-added services can be designed with an existing tool. In this case the Oracle JDeveloper is used to build an echo service which sends incoming Instant Messages (IM) back to the sender (bpelecho_client). The process consists of three activities, a receive activity (receiveIM) which waits for incoming messages, an assign activity (Assign_Message_and_MessageReceiver) which copies the text from the received message into the new message and configures the sender and the receiver of this new message, and an invoke activity (sendbackIM) which returns the message back.

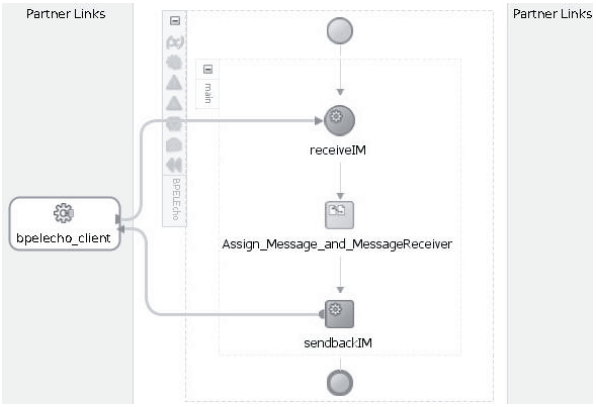

Figure 2. BPEL Process Echo Service

If the output of the GUI is not an executable value-added service, a code generator is required to convert the output of the GUI into the service code. A service description may consist of some definition files, description files and other resources which serve as input for the code generator (see Figure 3). In the case JSLEE is used as execution environment, a deployable unit is generated as output of the code generation process. The service deployment is used to copy the service to the application server. It uses the deployable unit that was generated by the code generator. The deployable unit consists of the service code, the required resources and the deployment and service descriptors. The descriptors define the service components and other parameters of the service (see Figure 3).

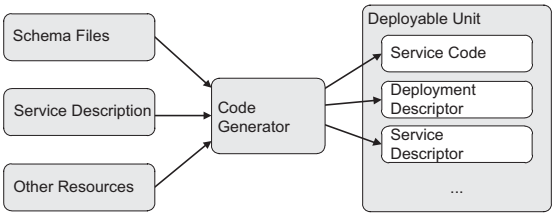

Figure 3. The Code Generator

As service execution environment the JSLEE framework is used. JSLEE is designed for ensuring low latency and providing high throughput to accomplish the requirements for communication services. The components of the service are executed in the component container. In JSLEE, these service components are called Service Building Blocks (SBBs). Four different structures of these service components are described in section 2.2.

The Service Transport Layer in the JSLEE framework abstracts different protocols in order to enable upper service layers to be independent of a specific communication protocol. Therefore it supports several communication networks. In [4] so-called Resource Adaptors (RAs) are defined abstracting the underlying infrastructure. These Resource Adaptors provide a common Java API that hides the communication protocol underneath. The communication with the SBBs is accomplished by the use of events.

2.2. The component structure

After once being generated, a service can consist of one or multiple service components. In the latter case, these components have to communicate together by exchanging events. The service components have to cooperate to ensure the fulfilment of the service workflow. In this section four different service component structures are presented, the single component structure in section 2.2.1, the parallel component structure in section 2.2.2, a structure that orchestrates

service components in section 2.2.3 and a structure that uses choreography for the components in section 2.2.4.

2.2.1. Single component structure

In this concept the code generator creates only one component which represents the whole service. Within this component a state machine controls the service workflow. It decides about the events that are allowed to be received in the individual states. Figure 4 shows a scenario which represents a service that consists of only one SBB and two resource adaptors. The SBB in this figure sends and receives events from the resource adaptors. The RAs translates incoming protocol messages from the network to events and vice versa. From all three activities in the echo service in Figure 2 only one SBB will be generated.

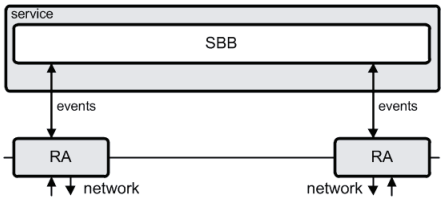

Figure 4. Single component structure example

This concept was introduced in [5], implemented and proved in [3, 6]. In principle it is possible to generate value-added services that consist of only one component, but with this approach it is not possible to realise parallel execution of service workflow elements. This characteristic leads to the new approach, the parallel component concept.

2.2.2. The parallel component approach

As described in the previous section, the problem with the execution of parallel workflow elements leads to a parallel component approach. JSLEE only supports sequential program execution in one SBB. It is not allowed to use multithreading within an SBB. A possibility to use parallel program execution in JSLEE is to use more than one SBB. They can be executed in parallel from each other. The concept to use this characteristic to obtain parallel executed service parts is described in [7]. If a parallel activity is required in the workflow, a SBB is generated for each parallel workflow part. One SBB is generated for each parallel element sequence within the workflow. The elements which are represented by one SBB are executed sequentially. The generated SBBs (see Figure 5) are communicating with the help of events.

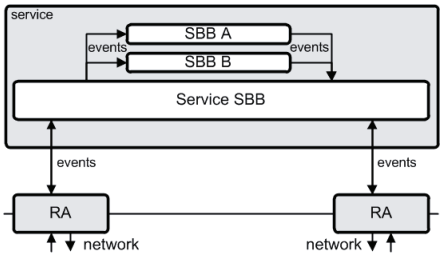

Figure 5. Parallel component structure example

If the workflow of the Service SBB reaches the position in the workflow where parallel execution is required, events are fired to the SBBs which represent the other parallel parts of the workflow. After the Service SBB has fired its events, it waits for the answer events from the called SBBs. The Service SBB uses request events to signal the other SBBs to start processing. After the Service SBB has fired its events to all parallel SBBs, it waits for the returning response events. With the request events the parallel SBBs receives the required parameter values from the main SBB. These values are used to initialise and to activate the SBBs. After

processing of the SBB the changed parameter values are assigned to the response event and delivered back to the main SBB. If the Service SBB has received the response events from all parallel SBBs, the Service SBB can copy the parameter values from the events and continues processing.

2.2.3. Orchestration of service components

In the next two sections the idea of parallel service components is further extended. This section describes a component structure that allows the orchestration of service components. This concept was introduced in [8, 10], a prototype is already in development, and an enhanced framework will be published soon. In orchestration, a central process takes control over the involved services and coordinates the execution of different operations on the services involved in the operation (according to [9]). A special control SBB is required to control the service workflow and to coordinate the SBBs of the service. The control SBB assigns the work to the other SBBs, sets the required parameters, and decides, on which events an SBB has to listen and what events he has to fire. For each element of the workflow a new SBB is required. These SBBs implement the work task of the respective workflow element. For the echo service in Figure 2, three SBBs are required, i.e., one for each activity. The control SBB communicates with these SBBs via events. The control SBB is triggered by a service start event and initiates the required SBBs. The control SBB decides which SBB should be called next. In Figure 6 the control SBB is activated by the service start event. The control SBB determines which SBB is the next and fires an event with the required parameter to this SBB. The called SBB (in this case SBB A) performs its work, e.g. communicating with a resource adaptor. Subsequently the result is returned to the control SBB. With this information, the control SBB calls the next SBB according to the state machine. This procedure is repeated until the workflow is completed.

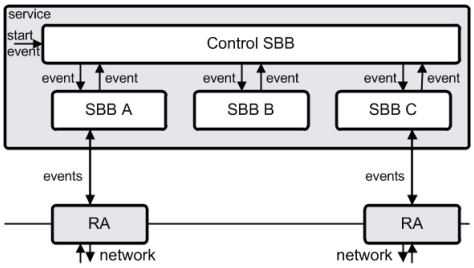

Figure 6. Orchestration component structure

This concept requires most of the intelligence in the control SBB. In the next concept, choreography is used for the service component structure.

2.2.4. Choreography of service components

Choreography does not require a central component. Each service involved in the choreography knows when to execute its operations and also knows its interaction partners (according to [9]). In this choreography based concept (introduced in [8]), the generated SBBs control themselves. No special control SBB is needed. These self-controlled SBBs communicate directly with each other. They get activated when they receive an event from its predecessor. The event includes all required parameters. So the SBB can start working after receiving the required event. Figure 7 illustrates an example service with the self-controlled SBB architecture. This service consists of three SBBs, SBB A, SBB B, and SBB C. Two resource adaptors are used. The service is activated when SBB A receives an event from a resource adaptor. The SBB A communicates with the resource adaptor and fires a new event to SBB B. After SBB B finished its work (e.g. copy and set parameters), this SBB fires an event to the SBB C. When SBB C has finished its part of the workflow, it fires an event to the resource adaptor.

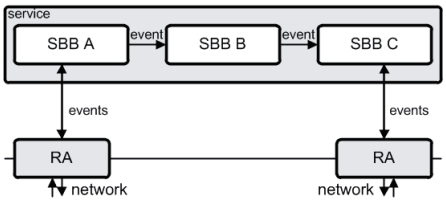

Figure 7. Choreography component structure

2.3. Selected characteristics for the component structure

All of the described concepts have their advantages and disadvantages. In this section several common characteristics of the described concepts are discussed. In Table 1 the different concepts of the component structure are compared based on selected characteristics.

Table 1. Selected characteristics of the component structure

Selected characteristics	Single Component Concept	Parallel Component Concept	Orchestration Component Concept	Choreography Component Concept
Parallel execution	No	Yes	Yes	Yes
Loose coupling	No	No	Yes	Yes
Code Generator required, precompiled	Yes	Yes	Depends on implementation	Depends on implementation
Easy expandable	No	No	Yes	Yes
3rd-Party development	No	No	Yes	Yes
Distributable service parts	No	No	Yes	Yes
Live reconfigurable	No	No	Yes	Yes

2.3.1. Parallel execution

As already mentioned in the section 2.2.1 the execution of parallel workflow elements is required in many value-added services. For a workflow which was generated with the single SBB concept only one component is created, which implements the whole workflow. In this case parallel execution of parallel parts of the workflow is not possible. The parallel component concept was specially developed to support parallel workflow execution. Also the component orchestration and the component choreography concept support parallel workflows.

2.3.2. Loose Coupling

Loose coupling means that components which are part of a service are mostly independent from each other. The first concept only consists of one component, so loose coupling is not supported. If there are no parallel sections within the workflow the parallel component approach behaves similar as the first concept and consists only of one component. Every parallel section which is added to the workflow, also adds a new component to the service. But between these service components a relatively high amount of dependencies exists. In both of the other two concepts, the orchestration and the choreography, a service consists of multiple components. Here loose coupling can be realised.

2.3.3. Code Generator required

For the single component concept and for the parallel component concept a code generator is required. The service components have to be compiled and the deployable unit has to be built with all components, the deployment descriptors, and other required resources. The orchestration concept and the choreography concept can also be compiled and put into a deployable unit. But it is also possible that the workflow will be deployed directly to the AS.

There the workflow will be analysed, the required SBBs are already deployed so the service is composed from the already deployed components. The composed service can then be executed.

2.3.4. Easily expandable

The support of new protocols depends on the underlying execution environment and on the implementation of the component structure. An execution environment like JSLEE supports the new protocols by adding a new resource adaptor for the respective protocol. In the single component approach and the parallel component approach the support of new protocols is possible but these approaches require a code generator. For each protocol which has to be supported, the code generator needs to be changed. The new protocol support has, in a worst case scenario, to be implemented directly within the code generator source code. So the extension of the code generator, for a support of new protocols, may not be easy. The other two approaches do not require a code generator. In these cases the service consists of different components. Special service components implement the communication with the resource adaptors. To support a new resource adaptor and a new protocol, only these special service components have to be implemented.

2.3.5. 3rd party development

The single component concept and the parallel component concept do not really support 3rd party development. Changes have to be made in the Code Generator. The orchestration and the choreography approach both use defined service components for the communication with resource adaptors, these service components also use a defined set of interfaces which has to be implemented. These service components can be developed by 3rd party developer.

2.3.6. Distributable service parts

In the single component concept, the service consists only of one component. So it is not possible to distribute the service to other computers. If there are parallel program parts in a workflow for a service that is generated from parallel components, the distribution of service components is possible but should already be known at compilation time. The orchestration and choreography concepts consist of multiple service components which can be composed at runtime. The communication between the service components takes place via events. If some components of the service are running on another computer, this event has to be sent over the network between the components. Instead of sending an event to a local service component, the event is send to a resource adaptor. The resource adaptor translates the event into protocol data units (PDUs) and sends these PDUs to the according JSLEE application server. There, another resource adaptor receives the PDUs and translates them into an event. This event is sent to the service components.

2.3.7. Live reconfigurable

Live reconfigurable in this case means the possibility to modify the service parameters or individual service components of the services and to add or remove service components during runtime. This feature is not possible with the single component and the parallel component approach. There the service has to be compiled with all required elements and they are not changeable once the service has been deployed. The orchestration and choreography concepts allow the reconfiguration of the running service. Parameters can be modified and service components can be added or removed while the components are deployed.

3. CONCLUSION

With all of the presented approaches it is possible to develop value-added services. Also with the single component concept, services can be created, but without support of parallel execution. For the single and the parallel component concepts prototypical implementations already exists and for the orchestration and choreography concepts a prototype is in development [10]. With

the new prototype other criteria such as performance issues can be evaluated. The choreography concept and the orchestration concept offer the most advantages. With a combination of the last two approaches, complex value-added services can be generated from fine grained service components at runtime. Also it is possible to dynamically reconfigure and expand the service by a rearrangement or the reconfiguration of the service components. With the presented approaches the concepts of service composition can be made available within JSLEE [10]. This allows an enhanced service creation and management framework on top of JSLEE. A prototype for this advanced framework is in development. This prototype will offer all the advantages from the choreography and the orchestration concepts and in addition, an advanced service management system, and a marketplace which offers service component sets, resource adaptors, and also value-added services. To support the reconfiguration and reorganisation of services at runtime, a web-based live GUI will also be part of the prototype.

ACKNOWLEDGEMENTS

The research project providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1724B09. The authors of this publication are in charge of its content.

REFERENCES

[1] D. Jordan *at all*. (2007) "Web Services Business Process Execution Language Version 2.0," OASIS.

[2] SPICE Project Web Site (2011, February): http://www.ist-spice.org.

[3] TeamCom Project Web Site (2011, February): http://www.ecs.fh-osnabrueck.de/teamcom.html.

[4] Ferry, D. (2008) "JAIN SLEE (JSLEE) 1.1 Specification, Final Release," OpenCloud, Sun.

[5] Eichelmann, T. *at all*., (2008) "Creation of value added services in NGN with BPEL," In *Proc. SEIN 2008*, pp186–193.

[6] Lehmann, A. *at all*., (2009) "TeamCom: A Service Creation Platform for Next Generation Networks," In *Proc. ICIW 2009*, on ICIW CD.

[7] Eichelmann, T. *at all*., (2009) "Support of parallel BPEL activities for the TeamCom Service Creation Platform for Next Generation Networks," In *Proc. SEIN 2009*, 2009, pp69–80.

[8] Eichelmann, T. *at all*., (2010) "Enhanced Concept of the TeamCom SCE for Automated Generated Services Based on JSLEE," In *Proc. INC 2010*, pp75-84.

[9] Juric, M.B., Mathew B. & Sarang, P. (2006) "Business Process Execution Language for Web Services, Second Edition," Packt Publishing.

[10] Steinheimer, M. (2011) "Entwicklung und Bewertung von Architekturansätzen für die Kombination von feingranularen Servicekomponenten zu Mehrwertdiensten," Master Thesis, University of Applied Sciences Darmstadt.