SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **43**:1–31
Published online 8 December 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1148

Distributed OSGi built over message-oriented middleware

Marek Psiuk*,†, Daniel Żmuda and Krzysztof Zieliński

AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Electronics
Department of Computer Science,
al. A. Mickiewicza, 30 30-059, Krakow

SUMMARY

This paper describes key aspects of remote service invocation in federations of OSGi containers. It refers to the OSGi Remote Service Admin specification and describes its efficient implementation over message-oriented middleware. Scalability problems of several different approaches to implementation are identified, and a solution in a form of innovative Remote Service Admin model extension is proposed. The extension, named On-demand Remote Service Admin, is analyzed and validated in the context of a motivating scenario. Validation includes performance and scalability evaluation, which confirms that all assumed requirements have been satisfied by the constructed prototype. Finally, the presented research is compared with related works. Copyright © 2011 John Wiley & Sons, Ltd.

Received 26 January 2011; Revised 24 October 2011; Accepted 25 October 2011

KEY WORDS: OSGi; Remote Service Admin; dynamic service groups; message-oriented middleware; network of brokers

1. INTRODUCTION

Modern enterprise applications are often distributed. Distributed processing has become a mainstream because of the global scale required for enterprise applications. This is why most established computational paradigms and platforms such as Remote Method Invocation (RMI) [1], Common Object Request Broker Architecture [2], or Jini [3] assume remote operation invocation, and location transparency is present.

Contemporary distributed applications such as telemedical systems or mobile pervasive systems require dynamic software environments with flexible, asynchronous group communication. This requirement favors the OSGi Service Platform [4, 5] and its extensions for distributed service computing. OSGi was initially aimed at a specific range of systems: mobile computing or automotive electronics. However, recent extensions of the platform, such as advanced distribution mechanisms, enable leveraging OSGi for enterprise and Web applications.

The basic goal of OSGi‡ is to create a dynamic component-oriented Java platform for applications developed in accordance with service-oriented design principles [6]. The OSGi framework provides an execution environment for applications, which are called *bundles*. Bundles expose their functionalities as services according to publish, bind, and find model [7]. Each bundle can be deployed and activated at runtime. Bundles can dynamically choose services that they are using. Furthermore, the

*Correspondence to: Marek Psiuk, AGH University of Science and Technology Faculty of Electrical Engineering, Automatics, Computer Science and Electronics Department of Computer Science, al. A. Mickiewicza 30 30-059, Krakow.
†E-mail: marek.psiuk@agh.edu.pl
‡http://www.osgi.org/

OSGi framework enforces strict modularization of bundles, which entails that there is no need to shutdown the entire JVM when a particular bundle is modified.

Aspects of distributed computing are addressed by the *Remote Services* specification published in the OSGi Service Compendium [8]. According to this specification, all remotely accessible services are handled by an internal *distribution provider* mechanism that creates service endpoints and proxies. Remote Services provide a foundation for *Remote Service Admin* (RSA) defined in the OSGi Enterprise Specification [9]. RSA defines an API for the distribution provider and service discovery, which can be used for managing distribution policy. Besides API, specification sets up general requirements, leaving implementation aspects open. Service orientation of the OSGi framework makes message-oriented middleware (MOM), such as the Java Messaging Service (JMS), the most natural choice for the implementation of distribution mechanisms. This is mainly due to the loose coupling of interactions between services, granularity of invocations, and heterogeneity of execution platforms.

The goal of this paper is to present how the RSA OSGi distribution model can be implemented with the use of advanced MOM features such as broker clustering and distributed destinations provided by a network of brokers. The need for a design of such scalable RSA implementation is motivated by a scenario of Teleconsultation System (TCS). We propose different approaches to RSA over MOM and show that all of them have some scalability problems. The main contribution of this work is an RSA extension that solves the identified problems. This paper presents in detail how OSGi distribution APIs might be mapped onto the MOM abstraction to provide a global view of available services. We also introduce a new service discovery protocol. The presented study is compared with projects such as Remoting OSGi (R-OSGi) [10] as well as with existing RSA implementations that constitute a widely accepted practical approach towards providing OSGi with the feature of remote services.

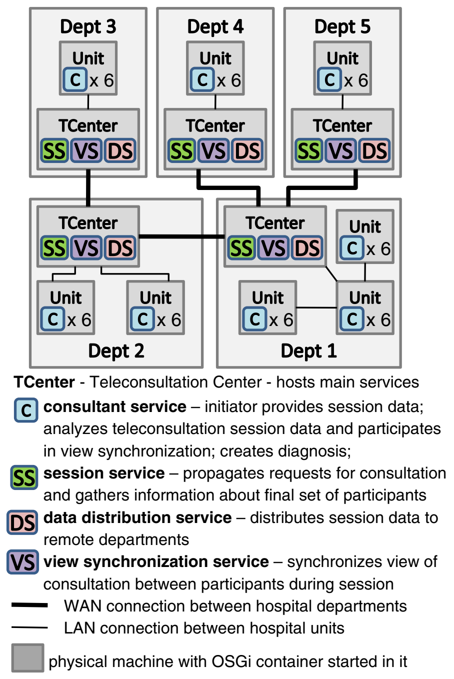
The structure of the paper is as follows. Section 2 outlines the requirements for the RSA implementation, which are illustrated by a motivating scenario. Section 3 analyzes various aspects of RSA implementation over the MOM platform. These considerations focus on mapping RSA functionality onto advanced MOM mechanisms and scalability aspects. The end result is an RSA extension referred to as On-demand RSA (ORS), which resolves scalability issues for invocations of dynamic service groups, and is described in Section 4, named Scalable On-demand Distribution Provider. This section also covers implementation aspects, including the Federation State Synchronization Protocol (FSSP). Section 5 describes a case study of the proposed ORS model, which refers to the requirements formulated in the context of the motivating scenario. Related work is presented in Section 6. Finally, the paper ends with conclusions in Section 7.

2. MOTIVATING SCENARIO—TELECONSULTATION SYSTEM

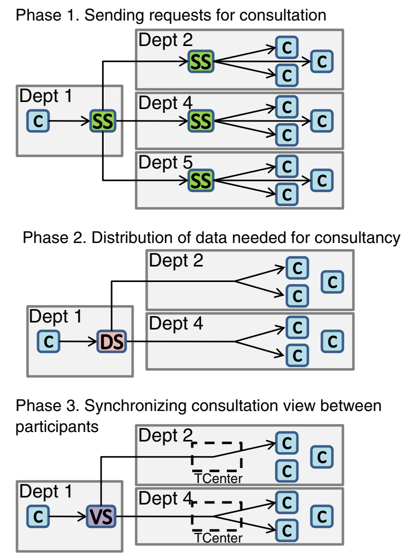
This section outlines the requirements for the presented solution with the aid of a motivating scenario—a hospital TCS. In the case of TCS, we consider a hospital with geographically dispersed departments, where there is often a need to quickly consult patient data with specialists located in remote departments. A TCS with similar assumptions can be found in [11, 12]. When a patient is in the hospital, his or her case is documented, among others, with large volumes of high-resolution data produced by computed radiography, magnetic resonance, computed tomography, or ultrasound. However, the latest innovations in telemedicine enabled automated acquisition of such information as blood pressure, pulse, and temperature remotely, on a regular daily basis, when the patient is at home. We are referring to systems providing such functionality as systems of remote patient data acquisition. Current research papers such as [13–19] prove that OSGi is an appropriate choice for implementation of such systems. It motivates OSGi-based TCS implementation as it could be integrated with remote patient data acquisition. Thanks to this, consulted patient data could be easily enriched with information acquired remotely. We are assuming that compliance with OSGi is sufficient for such integration. Another motivation for OSGi-based TCS is leveraging service-oriented architectures (SOA). As presented in [7], OSGi is capable of designing an application with a micro-SOA approach where the business logic is decomposed into lightweight services in a single address space (a single process of OSGi container). Micro-SOA, combined with mechanism of remote service invocation, allows TCS to benefit from the SOA paradigm by increased scalability, availability, flexibility, and extensibility as discussed in [20–22].

The OSGi-based systems that considered TCS is presented in Figure 1. It is deployed in a hospital with five geographically distributed departments. Departments are connected through the Internet, either directly or by virtual private networks. Each department encompasses one to three units (all units share the same geographical location, e.g., different floors of the same building). The central element in each department is a Teleconsultation Center (TCenter)—a server node with OSGi container that hosts the main TCS bundles. Besides this, each unit has a single node with an OSGi container. The TCenter and unit nodes are connected via a local area network (LAN). TCS is designed in accordance with the SOA; therefore, each major feature of the system is exposed as a service. Services provided by TCenter bundles are as follows: *session service* (SS), *data distribution service* (DS), and *view synchronization service* (VS). Different bundles are responsible for each of those services. OSGi container in each hospital unit hosts another set of bundles that provides *consultant services* (Cs). All mentioned services are briefly described in Figure 1(a).

In TCS, specialists in remote locations can be consulted by means of *teleconsultation sessions* (TSs). Each TS is divided into three phases, which are presented in Figure 1(b). Specialists in all hospital units operate their personal workstations, which are represented in TCS as Cs. In the first phase, a specialist initiates a TS by dispatching a request to the local SS. The request is then forwarded to all Cs, which match the criteria specified by the specialist initiating the session. Following this step, other specialists reply; thus, the SS assembles the final set of session participants. In the second TS phase, data are uploaded by specialists to the local DS (or downloaded by DS from specialized medical equipment). The DS distributes data to remote departments where session participants download it to their workstations. Finally, in the third phase, the actual TS is enacted. During the session, each participant can interact with the teleconsultation view. As depicted in Figure 1(b), the VS is used to synchronize view changes between participants. Because the VS performs resource-consuming


Dept 3
Dept 4
Dept 5
Unit
C x 6
TCenter
SS VS DS
Dept 2
Dept 1
TCenter - Teleconsultation Center - hosts main services
C consultant service – initiator provides session data; analyzes teleconsultation session data and participates in view synchronization; creates diagnosis;
SS session service – propagates requests for consultation and gathers information about final set of participants
DS data distribution service – distributes session data to remote departments
VS view synchronization service – synchronizes view of consultation between participants during session
WAN connection between hospital departments
LAN connection between hospital units
physical machine with OSGi container started in it

(a) High-level service deployment inside hospital infrastructure


Phase 1. Sending requests for consultation
Dept 1
Dept 2
Dept 4
Dept 5
C
SS
Phase 2. Distribution of data needed for consultancy
DS
Phase 3. Synchronizing consultation view between participants
VS
TCenter

(b) Phases of sample *teleconsultation session*

Figure 1. Motivating scenario—Teleconsultation System.

calculations, TCenter provides multiple instances of that service for load balancing purposes. At the conclusion of the session, specialists agree upon a diagnosis, which is uploaded to the DS.

The presented deployment of TCS and the anatomy of TS set the precise requirements for suitable RSA implementation. The requirements are as follows:

(R1) Containers should be connected in a network, and a service exported in one of the containers should be accessible in all other containers, not only those directly connected to it. This is important in TCS, as not all departments are directly connected to each other.

(R2) Each service should be able to invoke operations on a selected group of services located in remote containers (cf. Figure 1(b)) with the following different strategies:

  (R2a) simultaneous, for example, distributing teleconsultation data (Phase 2);
  (R2b) load balancing, for example, distributing workload between VSs (Phase 3);
  (R2c) failover, that is, choosing alternative Cs in case of failure (Phase 3).

(R3) Both one-way communication—or the in-only pattern [23] (important for efficiency when the data distribution is required in Phase 2)—and two-way communication—or the request–response pattern (querying Cs in Phase 1)—should be provided.

(R4) As TCS may be significantly enlarged, the solution should ensure scalability up to several dozen departments (a few hundred containers) and a few thousand services. The duration of a TS and resource consumption should be, at worst, linearly dependent on the number of departments, units, and services.

The aforementioned set of requirements is not satisfied by any of the currently available implementations of RSA. Developing a suitable implementation requires selecting appropriate middleware and properly mapping its communication mechanisms onto RSA operations.

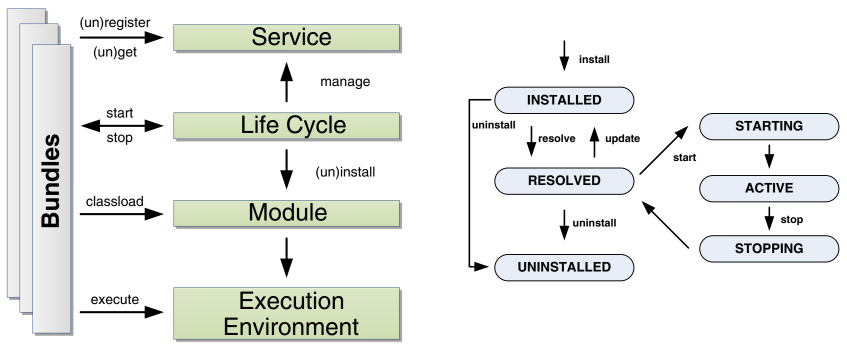
3. REMOTE SERVICE ADMIN OVER MESSAGE-ORIENTED MIDDLEWARE

This section examines whether MOM is appropriate for the RSA implementation aimed at the fulfillment of TCS requirements. To make the paper self-contained, we start with a brief description of the OSGi framework. Next, the RSA model and MOM features are presented. Afterwards, different approaches to RSA over MOM realization in the context of scalability are discussed. Finally, it is concluded that all of the presented approaches exhibit scalability issues, which are addressed in Section 4.

3.1. The OSGi framework

The OSGi framework stands as the core of the OSGi Service Platform Specification [5]. The framework provides an execution environment for applications and is divided into several layers, namely module, life cycle, and service. Interactions between particular layers are depicted in Figure 2(a). Their functionality is as follows:

- Module layer - defines a modularization model for OSGi applications, called *bundles*. A bundle is comprised of Java classes and other resources as a standard Java Archive. An important resource of each bundle is a manifest file. It contains descriptive information about a bundle, which, in particular, can contain entries about exported and imported Java packages.
- Life cycle layer - defines a runtime model for bundles along with their possible states: installed (the bundle has been successfully installed in the framework), uninstalled (the bundle has been uninstalled and cannot move into another state), resolved (all Java classes required by the bundle are available), starting, stopping, and active (the bundle has been successfully started and is running). The bundle state graph is depicted in Figure 2(b). A framework provides a life cycle management API for managing bundle state as well as extensive dependency mechanisms used to evaluate whether the particular bundle can be resolved or not.
- Service layer - defines dynamic service interaction mechanisms that are compliant with the publish, find, and bind model. When referring to OSGi services in the following sections, we



(a) Interactions between layers (b) Bundle life cycle

Figure 2. The OSGi framework.

mean instances of services. A service instance is a Java object that was registered in the framework under a set of interfaces, which is referred to in our paper as service type. Each bundle can register and search for services as well as receive notifications about service state changes.

Taking into account that all bundles are executed in the OSGi framework execution environment, a modification of a bundle does not require the whole environment to be restarted. Applications written in OSGi technology have to be aware that a whole execution environment is dynamic and that some of the services that are in use at some point may be unregistered.

According to the OSGi specification, a process in an operating system (OS) that hosts an OSGi framework is referred to as a framework instance. In this paper, we will refer to it as a container or container instance because this term is more intuitive. The specification states that each container should be uniquely identified. When we are referring to this unique identification, we are simply using the term 'container name'. The term OSGi federation is used when referring to the group of containers (uniquely named) interconnected through some communication protocol on the level of RSA mechanism.

3.2. Remote Service Admin model

Remote Services (OSGi compendium specification) introduce the notion of an *endpoint* and a *distribution provider* mechanism. The endpoint provides access to a service in another container requiring some specific communication protocol. The distribution provider creates endpoints for a service that is accessible to remote clients (export process) and registers proxies that access services external to a given container (import process). In OSGi, registered services have to be accompanied by a set of properties of which one, the service type, is obligatory. Remote Services define some properties that are used to declare the intention of exporting a service, as well as additional details, as required. Therefore, when registered services are provided with Remote Services properties, the distribution provider exports them by creating endpoints. Importing a service results in the creation of a *proxy* that accesses a remote endpoint and in the registration of the proxy as an *imported* service.[§] A container can simultaneously contain multiple distribution providers, for example, for different communication protocols. Each provider defines its own *configuration type*, which is an additional set of properties used to provide details of the export process specific to a particular provider (e.g., URL for RMI or queue for JMS). The mapping between services and endpoints, accompanied by communication characteristics, is referred to as the *topology* of a specific container federation.

Remote Services are extended by the RSA (OSGi Enterprise Specification), which provides management API for specification of which services (and under what conditions) should be exported/imported by the distribution provider. RSA introduces three independent elements: *RSA*, *Discovery*, and *Topology Manager*. The RSA provides an API for importing/exporting services but

[§]The imported service is registered with a set of (defined by specification) properties that indicate that the service was imported.

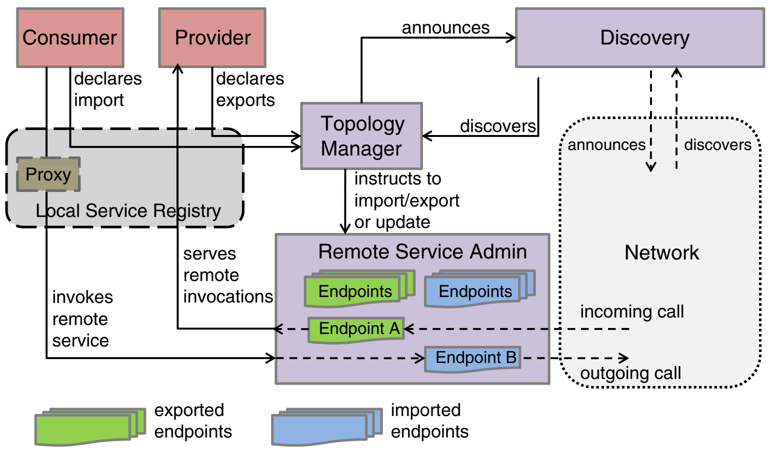


Figure 3. Remote Service Admin model.

does not initiate import/export on its own (passive distribution provider). The Discovery announces in the federation which services are exported in the local container and discovers services exported in other containers. The Topology Manager is a management agent that implements some distribution policy by controlling which services should be exported/imported. The Topology Manager cooperates with the Discovery and the RSA as depicted in Figure 3.

Another important feature of the RSA is Service Hooks, which provide request-based service importing with intents (cf. Service Hooks chapter in OSGi Core Specification [5]). As a result of the use of Service Hooks, a service can be imported only when the need for service consumption is detected. Intent is defined in the following way: 'a name for abstract distribution capability, which can be implemented by a service or can constrain the possible communication mechanisms of distribution provider … the purpose of the intents is to have a vocabulary that is shared between distribution aware bundles and distribution provider' [8]. Through the use of intents, an exported service can be annotated to express different distribution features, such as service group invocation, with different strategies: simultaneous (R2a), load balancing (R2b), and failover (R2c). Intents can also be used on the consumer side. During the presented request-based import operation, the consumer can specify its requirements[¶] by referring to particular intents. The Topology Manager can subsequently try to choose the appropriate endpoint among those that have been discovered. The implementation of the discovery process is stated as being out of the scope of its specification.

3.3. *Advanced features of message-oriented middleware*

Message-oriented middleware provides distributed communication with the use of messages. The two main messaging models of MOM are *point-to-point* and *publish/subscribe*. Both models allow multiple senders and multiple receivers, but in the point-to-point model, one message can be received only by a single receiver, whereas in the publish/subscribe model, a single message is received by all interested receivers. The MOM paradigm involves loose coupling of senders and receivers, which is facilitated by the concept of a *broker*. Senders and receivers interact only with a broker that mediates communication between its clients. The simplified MOM variant assumes a single broker that does not provide proper scalability in enterprise environments. Therefore, fully featured MOM envisions a network of brokers that distributes the workload and ensures sufficient scalability for enterprise operation.

The JMS is an API defined by the Java platform for interaction with MOM. This study focuses on OSGi—a Java-centric solution; therefore, JMS is used as the MOM interface. JMS uses *queues* to implement point-to-point messaging and *topics* to implement the *publish/subscribe* model. In

[¶]Requirements can be specified through a service filter used for every service consumption instance (Chapter 5.5 of the Core Specification).

JMS, the sending client is called the producer, whereas the receiving client is called the consumer. JMS brokers currently available such as Oracle WebLogic Server 11g (Oracle Corporation, Redwood Shores, CA, United States), IBM WebSphere MQ V7.0 (IBM Corporation, New York, NY, United States), Apache ActiveMQ 5.3 (The Apache Software Foundation, Forest Hill, MD, United States), and JBoss HornetQ 2.1.2 (Red Hat Inc., Raleigh, NC, United States) not only provide simplified MOM but also enable brokers to connect to each other in order to provide more advanced MOM features. In our work, the focus is on a specific set of features that are particularly useful for fulfilling the requirements stated in Section 2. Two key features include *distributed queues* and *distributed topics*. Any number of consumers and producers can connect to a distributed queue or topic at any broker. Producers can simultaneously produce messages to distributed destinations, and the following semantics are maintained at all times:

- Distributed queues - A single message is consumed only by a single consumer; in the case of multiple consumers, messages are evenly load-balanced over the whole consumer set. Actual broker topology and locations of consumers are irrelevant.
- Distributed topics - A single message is consumed by all consumers connected to the topic at a given moment. If the consumer is of a *durable* type, all messages produced in the topic are delivered to the consumer even if the consumer was temporarily unavailable at the time the messages were produced; messages produced for a given topic are sent only once between two adjacent brokers even if multiple consumers are waiting on the receiving side.

Most JMS brokers enable restricting the number of brokers through which a message can be passed for a given distributed destination. We will refer to this feature as message time to live.

Another important advanced MOM feature is high availability, which is achieved by broker redundancy. High availability is facilitated by replicating all operations performed on a single broker to one or many backup brokers. Broker clients connect to a single master broker, and when it fails, they switch to one of the available backup brokers by using the failover mechanism. The backup broker takes over from the point where the master has failed and continues to operate, making the failure transparent to broker clients. Thus, distribution destinations described earlier are immune to failures that occur in the broker network.

Advanced features of MOM make it an appropriate choice for RSA implementation aimed at TCS. Remote invocation of service groups (R2) in different strategies can be implemented with the use of distributed JMS destinations. The simultaneous strategy (R2a) can be implemented with the use of topics, the load balancing strategy (R2b) can be implemented with the use of queues, and the failover strategy (R2c) can be achieved through the high availability of the broker network. Connecting brokers in a network addresses requirement (R1)—brokers can communicate without being directly connected. By default, MOM only provides one-way communication; however, there are well-known solutions (discussed later) for achieving a request–response pattern that addresses requirement (R3). The scalability requirement (R4) is discussed in the following section.

3.4. Approaches to Remote Service Admin over message-oriented middleware and their scalability problems

In our approach, the OSGi container is provided with a JMS broker that enables distributed JMS destinations in intercontainer communication. Brokers are interconnected in some topologies and, as such, provide a federation of OSGi containers (a sample federation is presented in the motivating scenario). In order to propose a complete RSA implementation, the following operations have to be mapped to MOM mechanisms: (i) exporting service, (ii) importing service, (iii) invoking service, and (iv) service discovery. This section discusses several different approaches to performing this mapping. Regarding each operation, all presented approaches have the following assumptions in common:

- Exporting - The exported service is bound to a queue[‖] or a topic by the creating consumer. Consumed messages are translated into service invocation.

[‖]Distributed queues and topics introduced in the previous section are simply referred to as topics and queues (the distributed prefix is implicit).

- Importing - The imported service proxy is bound to a topic or queue by creating (or reusing) a producer. Importing also involves binding the proxy as a consumer of the response queue, which is used for receiving invocation responses. Each proxy has a dedicated response queue.
- Invoking - The proxy invocation is translated to a message that is dispatched to the bound destination. The message is consumed by the exported service consumer and translated into a service invocation. In the case of two-way invocations, service responses are translated into messages that are inserted by an ad hoc producer into the response queue of the imported service proxy. When a response message is consumed, it is unmarshaled to a value returned by a method invoked on the proxy.
- Discovering - One well-known topic for the entire federation is used by all containers to periodically announce exported services.

Given such assumptions, the mapping of operations (i)–(iii) in the context of a single service becomes straightforward: (i) the exported service is bound to a new dedicated queue; (ii) the imported proxy is bound to the queue of the exported service; and (iii) the invocation is effected by transferring the message from the proxy to the service through its unique queue and by transferring the reply through the proxy's reply queue. However, mapping of operations (i)–(iii) for a service group invocation is not so straightforward because a variety of strategies (simultaneous, load balancing, and failover) can be requested by the consumer. In the case of service groups, the mapping can follow different approaches, which are presented in the following paragraphs. The description of the handling service response (two-way invocation) is common for all approaches and will be presented later.

Before the approaches to achieving RSA over MOM are presented, several definitions have to be introduced:

- *service group* - a group of services requested by the consumer for invocation. As such, a group can only encompass services of the same service type, the service group is always a subset of (or is equal to) the *service type group*.
- *service type group* - all services exported in the federation with a given service type.
- *service type count* - the number of different types of services exported in a single container.
- *service type superset* - the set of all possible subsets of the service type group.

Queue Approach. There is a dedicated queue for each service. RSA operations are mapped as follows: (i) the exported service is bound to a new dedicated queue; (ii) the imported proxy creates producers to all queues of services from the service group; and (iii) for each strategy, an invocation message is produced by suitable producers, as follows:

- simultaneous: by all proxy producers;
- load balancing: by the producer that had not been used for the longest time; and
- failover: by the producer chosen during the first invocation and used as long as it remains available. When it fails, another producer is chosen randomly.

Topic Approach. There is a dedicated topic for each service type group: it is referred to as topic T. RSA operations are mapped as follows: (i) the exported service is bound to the T topic; (ii) the imported proxy creates a producer for the T topic; and (iii) because the service group might be a subset of the service type group, the proxy always enriches its messages with a list of IDs that specifies services to which each message is directed. If the consumer consumes a message not directed to the exported service, the message is ignored. The invocation message is always produced for the T topic. However, depending on the strategy, the message may be enriched with the IDs of the following services:

- simultaneous: all services belonging to the group;
- load balancing: a service from the group which has not been used for the longest time;
- failover: a random service chosen during the first invocation. The service is used for later invocations as long as it remains available. When it fails, the ID of another service is used.

Subgroup Approach. There is a dedicated queue and a dedicated topic for each subset in the service type superset.[**] RSA operations are mapped as follows: (i) the exported service is bound to all queues and topics of the service type superset; and (ii and iii) the imported proxy creates a producer for the following destination depending on the invocation strategy:

- load balancing: the producer for a queue of a service type group subset is equal to the service group;
- failover: identical to the Queue Approach, but a random producer is chosen from among queues dedicated to single services belonging to the group;
- simultaneous: the producer to a topic of a service type group subset is equal to the service group.

Handling service responses in two-way invocations of a single service has already been described. When invoking a service group with load balancing and failover strategies, the procedure remains the same. The only difference occurs in the simultaneous invocation strategy, where more than one return value is produced; therefore, some additional aggregating algorithm is needed. An important fact is that handling responses with the simultaneous strategy is independent of the approach used for group invocation. Pursuant to JGroups [24] (prominent solutions for group communication in Java), we propose the following aggregating algorithms:

- GET_FIRST - the invocation waits for the first response and returns it;
- GET_FIRST_WAIT_ALL - the invocation waits for all responses and returns the first one;
- GET_N - the invocation waits for N responses and returns them;
- GET_N_WAIT_ALL - the invocation waits for all responses and returns the first N responses;
- GET_ALL - the invocation waits for all responses and returns them.

In the case of GET_FIRST and GET_FIRST_WAIT_ALL aggregations, there is not any problem, because invocation results are aggregated to a single value, which can be returned by the proxy (it complies with the service contract). In the case of GET_N* and GET_ALL aggregations, there is a need to return more than one value by the proxy. The current version of the RSA specification does not provide mechanisms that make it possible. The solution for this problem in the form of an RSA extension is presented in the next section.

Thanks to the use of advanced MOM features,the approaches presented fulfill the first three requirements (R1–R3) of TCS. To assess the fulfillment of requirement (R4), scalability, we have chosen two metrics: *Messages Involved*, the number of JMS messages involved in N invocations of a given service group; and *Memory Used*, the memory required by a container for exporting all remote services, that is, the memory used to create the required JMS destinations. The results of scalability analysis for each approach are presented in Table I. The first section of the table presents the *Messages Involved* metric. In the Queue Approach, for the purpose of simultaneous invocation, each message has to be produced to all queues of the service group. Therefore, a message with the same data can be transmitted more than once between two given neighboring brokers, which results in unnecessary bandwidth consumption.[††] In the Topic Approach, load balancing and failover invocations result in transmitting messages not only to the service group, but also to the entire service type group. Therefore, given an increased size of the service type group as reflected by the *Messages Involved* metric, scalability problems may occur. The Subgroup Approach ensures a dedicated JMS destination for each selected service group and invocation strategy, therefore eliminating problems associated with the former approaches.

The final column presents the *Memory Used* metric. The memory required in Queue and Topic Approaches is linearly dependent on the service type count. In the case of the Subgroup Approach, for each service type, there has to be at least one JMS destination for each subset of the service type group. This results in linear dependency on the service type count and exponential dependency on

[**]Topics for single services can be omitted. Example: three services A, B, C of service type X are exported in a federation; therefore, seven queues $Q_A, Q_B, Q_C, Q_{AB}, Q_{AC}, Q_{BC}, Q_{ABC}$ and four topics $T_{AB}, T_{AC}, T_{BC}, T_{ABC}$ are needed.

[††]This is in contrast to other approaches where distributed topic semantics ensure that messages are sent only once between two given neighboring brokers even if multiple consumers are waiting on the receiving side.

Table I. Analysis of scalability in various approaches to Remote Service Admin over Message Over Middleware.

	Messages Involved in N invocations of a given *service group*						*Memory Used*
	simultaneous		failover		load balancing		
	produced	consumed	produced	consumed	produced	consumed	
Queues	$N * X$	$N * X$	N	N	N	N	$M * \text{avg}(X) * Z$
Topics	N	$N * Y$	N	$N * Y$	N	$N * Y$	$M * Z$
Subgroups	N	$N * X$	N	N	N	N	$2M * \sum_{Z} 2^{Y}$

X - service group size; Y - service type group size; Z - service type count; M - memory for queue/topic.

the sizes of service type groups. Exponential dependency may be acceptable when the number of service groups is close to the size of the service type superset. However, in the case of TCS, service groups frequently change as different consultation sessions are started and stopped. The number of service groups required at a particular moment is only a small fraction of the service type superset size. This significantly limits system scalability. The linear metric dependency of other approaches seems to provide acceptable scalability, but in the case of highly overloaded containers (high service type count), the amount of consumed memory can be very high.

Scalability analysis exposes an issue, which can be formulated as follows: The current RSA model makes it difficult to develop a scalable MOM-based distribution provider capable of fulfilling the requirements stated in Section 2. The model has to be extended to facilitate *Messages Involved* scalability, similar to what is observed in the Subgroup Approach, whereas the scalability of the Memory Used metric has to be better than in the Topic Approach. The scalability of *service discovery* is also very important. The naive approach proposed at the beginning of this section results in messages periodically being sent to all containers in the federation, which is not efficient. There is a need for more robust service discovery, capable of supporting the presented RSA extension. The last problem is the lack of a mechanism that enables invocation of service groups in simultaneous strategy in conjunction with the GET_N* and GET_ALL aggregation algorithms. These problems will be addressed in the next section.

4. SCALABLE ON-DEMAND DISTRIBUTION PROVIDER

The problems stated at the end of the previous section can be tackled by extending the RSA model to enable importing and exporting services on demand without degradation of scalability. The extension is introduced as a custom model referred to as On-demand RSA (ORS). ORS is mapped to MOM in a way that leverages benefits of the Subgroup Approach, ameliorating its memory consumption issues. A crucial element of ORS is the FSSP, which provides scalable service discovery.

4.1. On-demand Remote Service Admin model

The ORS model is presented in Figure 4. Major ORS contributions are as follows:

- enrichment of Topology Manager and Discovery with new elements;
- extension of Topology Manager and Discovery API;
- introduction of Asynchronous Service Group (ASG) API, which enables usage of GET_N* and GET_ALL aggregation algorithms; and
- FSSP—a protocol for announcing, updating, and synchronizing current ORS configuration state, including nodes available in the federation, services available for export, services required for import, and imported and exported endpoints.

The Topology Manager contains two new internal elements: the *Export Manager* and the *Import Manager*, whereas Discovery involves three new elements: the *Consumption Requester*, the *Consumption Listener*, and the *Provider Announcer*. OSGi specification restricts the Topology Manager

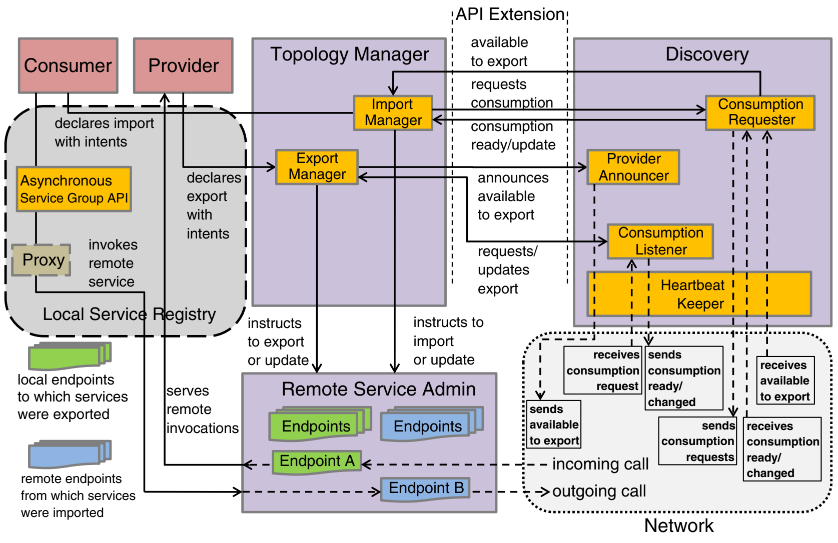

API Extension
Consumer
Provider
Topology Manager
Discovery
available to export
requests consumption
consumption ready/update
Import Manager
Consumption Requester
declares import with intents
Export Manager
Provider Announcer
Asynchronous Service Group API
declares export with intents
announces available to export
Consumption Listener
Proxy
invokes remote service
requests/ updates export
Heartbeat Keeper
Local Service Registry
instructs to export or update
instructs to import or update
receives consumption request
sends consumption ready/ changed
receives available to export
local endpoints to which services were exported
serves remote invocations
Remote Service Admin
Endpoints
Endpoints
sends available to export
sends consumption requests
receives consumption ready/ changed
Endpoint A
incoming call
remote endpoints from which services were imported
Endpoint B
outgoing call
Network

Figure 4. On-demand Remote Service Admin model.

API in discovering endpoints and the Discovery API in announcing endpoints (cf. Figure 3). The new model extends this API with the following semantics: announcing services available for export, requesting consumption, informing that consumption is prepared or has changed, and requesting or changing service export. The aim of ASG API is not only to support the GET_N* and GET_ALL aggregations but also to provide a general asynchronous alternative to regular synchronous remote service invocations. ASG API is presented at the end of this subsection. Typically, FSSP is used for updating the federation state in one container (e.g., announcing a new service available for export) and receiving this update in another container. Details of FSSP are discussed in the next section. ORS also extends the concept of intents by dividing them into the following groups:

I1 service capabilities specified by the service provider, for example, intents specifying group invocation strategies[‡‡] or communication patterns supported by a given service;

I2 demand added on by the importing container, for example, intent added when creating a synchronous or asynchronous service proxy for a particular communication pattern (either one-way or two-way);

I3 demand added on by the exporting container, for example, intent added when exporting an endpoint with additional encryption required for confidentiality;

I4 related service groups, for example, intent for requesting a group invocation strategy—combination of I2 and I3 as both exporting and importing containers are involved.

Figures 5 and 6 present UML sequence diagrams for export and import processes, respectively. Export Manager, Provider Announcer, and Consumption Listener are involved mainly in the export process, whereas Import Manager and Consumption Requester are involved in the import process. In order to better explain the export and importing processes, the following concepts are defined:

- *remote service reference* - a reference to a service. The reference contains the name of the exporting container and service ID (as defined in the specification [5]); therefore, it is unique in the federation.
- *service group reference* - the set of remote service references for each service belonging to a service group.
- *consumption request* - the message sent from the importing to the exporting container when a bundle requires a particular service (or service group) to function properly. It contains a

[‡‡]The service provider is responsible for providing information about supported group invocation strategies. It is important as load balancing and failover strategies require service to be stateless, which can be ensured only by the provider.

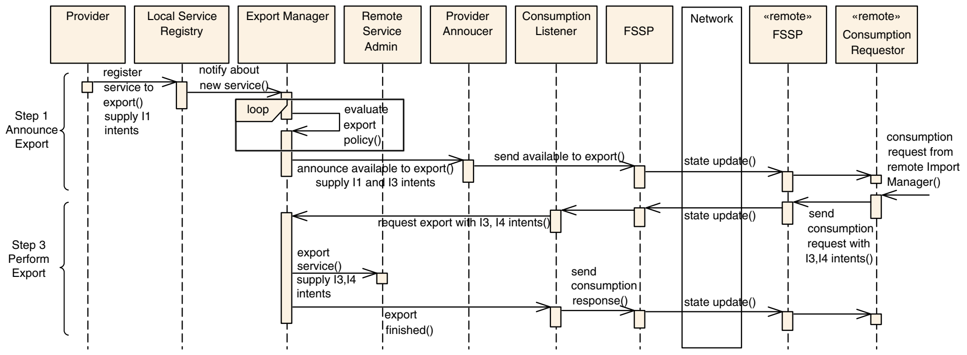

Provider
Local Service Registry
Export Manager
Remote Service Admin
Provider Annoucer
Consumption Listener
FSSP
Network
«remote» FSSP
«remote» Consumption Requestor
Step 1 Announce Export
register service to export() supply I1 intents
notify about new service()
loop
evaluate export policy()
announce available to export() supply I1 and I3 intents
send available to export()
state update()
consumption request from remote Import Manager()
Step 3 Perform Export
request export with I3, I4 intents()
send consumption request with I3,I4 intents()
export service() supply I3,I4 intents
send consumption response()
export finished()

Figure 5. Sequence diagram of the export process. FSSP, Federation State Synchronization Protocol.

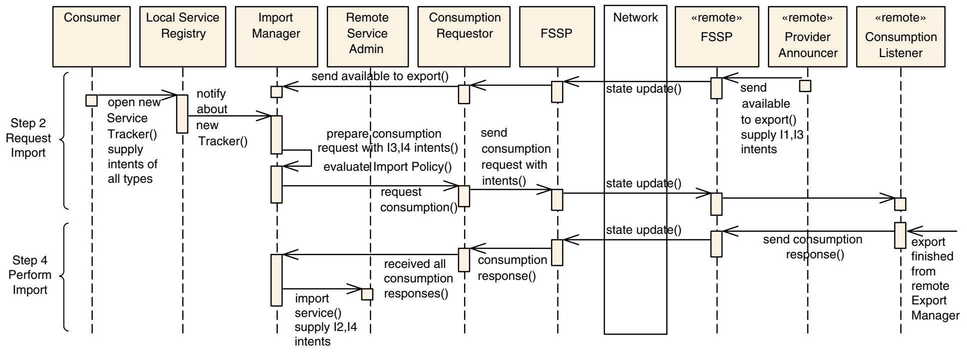

Consumer
Local Service Registry
Import Manager
Remote Service Admin
Consumption Requestor
FSSP
Network
«remote» FSSP
«remote» Provider Announcer
«remote» Consumption Listener
Step 2 Request Import
open new Service Tracker() supply intents of all types
notify about new Tracker()
send available to export()
state update()
send available to export() supply I1,I3 intents
prepare consumption request with I3,I4 intents()
evaluate Import Policy()
request consumption()
send consumption request with intents()
Step 4 Perform Import
send consumption response()
received all consumption responses()
consumption response()
import service() supply I2,I4 intents
export finished from remote Export Manager

Figure 6. Sequence diagram of the import process. FSSP, Federation State Synchronization Protocol.

remote service reference or a service group reference, a unique ID and a list of I3 and I4 intents requested for service export.

- *consumption response* - a message announced by the exporting container, containing the set of currently supported JMS destinations.
- *container group* - the set of containers associated with a service group such that each container in the set exports one or more services from the service group.

The export and import processes can be divided into four steps, which are described as follows.

Step 1 Announce Export. The export process is initiated by the Provider, which registers a service with intents (only I1 intents) and properties specifying the details of the process. Subsequently, the Export Manager is notified by the service registry about the registration of a new service. The Export Manager evaluates its export policy to decide whether a service can be exported or not. This policy can be evaluated multiple times and at different moments (e.g., its evaluation can be postponed). When the Export Manager decides that a particular service can be exported, it announces a new service available for export, providing I1 intents and intents supported by the exporting container—I3. The Export Manager announces the new service to the Provider Announcer. The Announcer forwards this announcement to FSSP, which triggers a federation state update. In all other containers, announcements are received through FSSP by the Consumption Requester. At this point, no further actions are performed (the service is not actually exported) until some remote container sends a consumption request.

Step 2 Request Import. The import process is initiated by a consumer that uses service tracker mechanisms (presented in Chapter 701 of Compendium OSGi Specification) to request a service. The consumer supplies an OSGi filter to the service tracker and then opens the tracker. The filter contains requirements for service import, including service type, properties, and intents of all types (I1–I4). The Import Manager uses Service Hooks, provided by RSA, to detect a new tracker

```
/**
   Calculates ConsumptionRequest on the basis of the Consumer's OSGi filter and
   a list of all services announced as available for export.
   The list also contains all services which are currently exported.
   The returned ConsumptionRequest contains requested I3 and I4 intents.
*/
ConsumptionRequest prepareToImport(Filter f, List availableToExport) {
    ConsumptionRequest cr = new ConsumptionRequest();
    cr.id = new java.util.UUID();
    List<RemoteService> matchingServices = new ArrayList();
    for (RemoteService available : availableToExport) {
        if (match(f.serviceType, available.serviceType)) &&
            match(f.properties, available.properties) &&
            match(f.intentsI1, available.intentsI1) &&
            match(f.intentsI3, available.intentsI3)) {
            matchingServices.add(available);
        }
    }
    if (matchingServices.isEmpty()) return null;
    cr.intentsI3 = f.intentsI3;
    cr.containerGroup = new HashSet<OSGiContainer>();
    if (f.intentsI4 == null) {
        /* using regular OSGi service ranking */
        cr.singleService = chooseHighestRank(matchingServices);
        cr.containerGroup.add(cr.singleService.exportingContainer);
    } else {
        cr.serviceGroup = matchingServices;
        cr.intentsI4 = f.intentsI4;
        for (RemoteService r : cr.serviceGroup) {
            if (!containerGroup.contains(r.exportingContainer)) {
                cr.containerGroup.add(r.exportingContainer);
            }
        }
    }
    return cr;
}
```

Listing 1: Algorithm of Consumption Request Calculation

and extract its filter. A consumption request is prepared according to the algorithm presented in Listing 1.

The Import Manager evaluates the policy to decide whether a consumption request can be sent. If this evaluation is positive, the Consumption Requester uses FSSP to send a consumption request to all containers from the container group. The Import Manager remembers I2 intents provided in the filter and associates them with a given consumption request. These intents are needed later for importing the service in step 4. If services available for export change, the consumption request has to be recalculated and, if necessary, updated (removed) through FSSP.

Step 3 Perform Export. In each container of the container group, the consumption request is received by the Consumption Listener, which stores it for further use. The Consumption Listener then performs two independent tasks: (task 1) preparing the required JMS destination and (task 2) preparing resources required by I3 intents. In task 1, the JMS destination required to support the consumption request is calculated in the following manner:

- If (*container group size* $==$ 1), then a queue with the name *EXPORT_ContainerName* is needed.
- If (*container group size* $>$ 1) and if the invocation strategy specified in I4 intents is
  - *simultaneous*, then a topic with a name resulting from concatenation of all lexicographically sorted container names from the container group is needed (e.g., for group (A, B, C), the name is *EXPORT_A_B_C*); otherwise, if it is
  - *load balancing* or *failover*, then a queue with the name *EXPORT_ContainerName* is needed.

If the calculated JMS destination is not currently exported, the Consumption Listener requests the Export Manager to export it. Thus, there is always only one JMS destination of a given name and type. The Export Manager calls the RSA to export the new JMS destination, that is, create it. The RSA attaches a single JMS consumer to the exported JMS destination. Once the JMS destination is exported, task 1 is finished, and the Consumption Listener announces its consumption response (containing the set of currently exported JMS destinations) in the federation through FSSP. In task 2, the Consumption Listener checks if the resources for some of the requested I3 intents are not currently initialized and, if so, requests the Export Manager to initialize them (e.g., prepare an encryption processor for the purposes of supporting the *confidentiality* I3 intent). When a consumption request is withdrawn, the Consumption Listener also performs two mentioned tasks: (task 1) unexporting JMS destinations that are no longer needed and (task 2) uninitializing resources of I3 intents that are no longer needed.

Step 4 Perform Import. In the importing container, the Consumption Requester waits for consumption responses from all containers from the container group. Once they are received, the Import Manager is notified. The Import Manager retrieves I2 intents stored in step 2 and performs imports by calling the appropriate method of the RSA element, supplying I2 and I4 intents. The import operation is mapped to MOM in the following way:

- If (*container group size* $==$ 1), then a producer for the exporting container queue is created.
- If (*container group size* $>$ 1) and if the invocation strategy is
    - *simultaneous*, then a producer for the topic whose name results from concatenation of all names of containers from the container group is created; otherwise, if it is
    - *load balancing* or *failover*, then producers for all queues of the container group are created.
- If the consumption request is associated with an I2 intent, which requests two-way invocation, the proxy is remembered as one of the possible targets of response messages. For the purpose of receiving response messages, each container has a single queue named *IMPORT_ContainerName*.
- If the consumption request is associated with an I2 intent that requests asynchronous API, then an asynchronous proxy is provided. Otherwise, a regular synchronous proxy is returned to the consumer.

It may happen that all containers from the container group already support the JMS destination needed by the consumption request. In this case, the Consumption Requester does not wait for consumption responses and instantly notifies the Import Manager that the import can be performed.

In order to explain how service invocation is mapped to MOM, it is important to understand that in the exporting container, each *EXPORT_** JMS destination is simultaneously used for different consumption requests, that is, different exported services. Similarly, in the importing container, the *IMPORT_** queue is simultaneously used for different consumption requests, that is, different imported services. To support it, two components, called EMplexer (exporting multiplexer) and IMplexer (importing multiplexer), are introduced. EMplexer receives messages from all *EXPORT_** destinations and passes them to exported services. IMplexer receives messages from the *IMPORT_** queue and passes them to the invoking proxies. Both components have configurable thread pools [25] that increase the number of threads on demand, that is, when the number of concurrent invocations (transmitted JMS messages) increases. Owing to this feature, resources are consumed only when necessary.

The invocation of a single service is performed by marshaling a proxy invocation to a request message that is produced to the queue of a container from the container group (in the case of a single service, there is only one container in the container group; cf. Listing 1). This message is always enriched with the remote service reference and a list of I3 intents specified in the consumption request. If the invocation is two-way, the request message may be additionally enriched with the name of the response queue of the importing container. The request message is consumed by EMplexer in the exporting container, and the reference is extracted, along with the list of I3 intents. EMplexer checks if resources for the requested I3 intents are initialized, and if they are not, the Export Manager is requested to initialize them. Upon unmarshaling, on the basis of remote service reference and I3 intents, the appropriate exported service can be invoked. In the case of two-way invocation, the returned value is marshaled into a response message, which is then produced to the response queue (name extracted from the request message). The response message is consumed by IMplexer in the importing container; its return value is unmarshaled and returned to the appropriate proxy on the basis of the consumption request ID. Depending on whether the proxy is synchronous or asynchronous, the message is delivered according to the chosen invocation model.

Group invocations are analogous to invocations of individual services, with the following differences for each invocation strategy.

In the **simultaneous strategy**, the request message is enriched with a service group reference and is produced to the topic with a name resulting from concatenation of all container names from the container group (sorted lexicographically). In each container from the container group, the message is consumed by EMplexer. A subset of the service group is calculated such that each service in the subset is exported in the current container. Upon unmarshaling, all services from the subset are simultaneously invoked. In the case of two-way invocation, responses are combined by EMplexer according to the selected aggregating algorithm (cf. Section 3.3), yielding a response message that is produced to the response queue. IMplexer in the importing container consumes response messages sent by all EMplexers from the container group and aggregates them according to the selected aggregating algorithm.

In the **load balancing strategy**, a service that has not been used for the longest time is chosen from the service group. The request message is enriched with the *service reference* of the selected service and produced to the queue of the container that exported the service. In the exporting container, the appropriate service is invoked on the basis of the extracted service reference. The remaining steps are identical to the individual service invocation scenario. For the purpose of simplicity, we have only considered the *last recently used* strategy of load balancing. Other strategies such as random choice, weighted random choice, and adaptive choice (response time based) are realizable in ORS as each service from the group can be independently reached. However, we are not discussing them as they do not influence the scalability aspect in the context presented in Section 3.4 and evaluated in Section 5.

In the **failover strategy**, a random service is chosen from the service group during the first invocation. As long as it remains available, the request message is produced to the queue of the container that exported the selected service. Similar to the load balancing strategy, the selected service is indicated by a service reference that is added to the request message. In the case of service failure, another service is chosen from the group. The remaining steps are identical to the individual service invocation scenario.

Asynchronous Service Group API

There are two main goals of ASG API: (i) providing means for both synchronous and asynchronous invocations of service groups with GET_N* and GET_ALL aggregating algorithms and (ii) providing means for asynchronous invocation of remote services. The ASG exposes two services, *ServiceGroupExecutor* and *AsyncServiceExecutor*, which can be used by service consumers in conjunction with two specific proxies: aggregating (AG) proxy (usable with *ServiceGroupExecutor*) and asynchronous (AS) proxy (usable with *AsyncServiceExecutor*). Introduced services and proxies are described in the following paragraph.

AG proxy is created when a consumer imports service group with at least one of the following intents: I2 (two-way invocation) and I4 (simultaneous strategy with GET_N, GET_N_WAIT_ALL, or GET_ALL aggregation algorithms; simultaneous strategy has to also be enabled by means of I1 intent during service export). **AS proxy** is created when a consumer imports service or service group with at least the following intents: I2 (two-way invocation and asynchronous proxy). **ServiceGroupExecutor** service can be used by consumers to invoke service group imported to the AG proxy. When its methods are called by, for example, thread A, underneath, the service realizes the invocation of the AG proxy method that was most recently called by thread A. Results of invocation can be asynchronously passed to provided ORSCallback object, or the consumer can poll results from returned ORSFuture object. Because there could be multiple values to return, ORSCallback can be notified multiple times. However, when all values are returned, ORSCallback is informed about it. ORSFuture provides methods for returning multiple values in a list or in a map (in case of a map, each value is accompanied with the name of the container that provided it). **AsyncServiceExecutor** service can be used by consumers to asynchronously invoke service or service group

```
public interface NamesService {
  public String giveRandomName(String nationality);
}

public class ASGUsagePresenter {
  public void invokeASProxy(AsyncServiceExecutor asExecutor,
                          NamesService asProxyNameService) throws Exception {
    // invocation with Future
    Future<String> returnHandle =
      asExecutor.invokeAsync(asProxyNameService.giveRandomName("polish"));
    String randomPolishName = returnHandle.get();
    // invocation with Callback
    asExecutor.invokeAsyncCallback(
      asProxyNameService.giveRandomName("polish"), new ORSCallback<String>() {
        public void notifyResult() {
            /* won't be invoked because the method is not void */ }
        public void notifyResult(String result) {
            String randomPolishName = result; }
        public void notifyException(Exception ex) {
            /* code which handles exception */ }
        }
    );
  }

  public void invokeAGProxy(ServiceGroupExecutor agExecutor,
                          NamesService agProxyNameService) throws Exception {
    // synchronous invocation with the use of ORSFuture
    List<String> randomIrishNamesList = agExecutor.invokeGroup(
      agProxyNameService.giveRandomName("irish")).getResultsList();
    Map<RemoteService, String> randomIrishNamesMap = agExecutor.invokeGroup(
      agProxyNameService.giveRandomName("irish")).getResultsMap();
    // invocation with Callback
    agExecutor.invokeGroupCallback(
      agProxyNameService.giveRandomName("irish"),new ORSGroupCallback<String>(){
        public void notifyResult(int number,RemoteService srv,String result){
          System.out.println("Received" + number + "result␣with␣value" +
                            result + "from␣service" + srv);
        }
        public void notifyResultsComplete() { /* all results received */}
        public void notifyException(Exception ex) {
          /* code which handles exception */ }
        }
    );
  }
}
```

Listing 2: Example of ASG API Usage

imported to the AS proxy. When one of service methods is called by, for example, thread A, underneath, the service realizes invocation of AS proxy method that was most recently called by thread A. Results of invocation can be asynchronously passed to the provided callback, or the consumer can poll results from the returned object. Because there is only one value to return, ORSCallback and ORSFuture objects do not have to be used.

Calling methods of AG and AS proxies do not bring any immediate result (if the method is supposed to return some value, a *null* is returned). It only provides indication that a given method should be invoked during the next call of an appropriate ASG service (*AsyncServiceExecutor* or *ServiceGroupExecutor*) executed by the same thread. The most intuitive usage of ASG API can be achieved by passing invocation of AG proxy or AS proxy to *ServiceGroupExecutor* and *AsyncServiceExecutor* services respectively. An example of such usage is presented in Listing 2.

Asynchronous Service Group API allows a consumer to invoke services asynchronously or to invoke a service group returning multiple responses without modification of a service interface. What is important is that semantics are not hidden to consumer. He or she is fully aware of aggregation operations and asynchronous invocations performed by proxies created during service import. ASG API does not need any specialized handling on the provider's side—it is transparent to providers. This RSA extension provides high expressiveness of service group invocations and ensures that the principles of SOA related to service contract are not broken.

4.2. Federation State Synchronization Protocol

Federation State Synchronization Protocol is used to synchronize information about the state of each container in the federation. Of course, it is assumed that there are no cycles in the underlying network of brokers (broker vendors provide features for cycle discovery and elimination). *Container state* is defined as a set consisting of the following elements:

- *state ID* - unique identification of a given container state instance;
- *name* - container name;
- *neighbors* - list of container names that are directly connected to the container;
- *available intents* - list of I3 intents that are supported in the export process;
- *available services* - list of services available for export;
- *exported destinations* - list of currently exported JMS destinations;
- *requested consumptions* - list of consumption requests created in this container.

Besides container state, the following additional definitions are relevant in the scope of FSSP:

- *achievable containers* - For each neighbor of a given container, a set of achievable containers can be defined involving containers that can be directly or indirectly accessed (a message to their JMS destinations can be produced) through the given neighbor.
- *significant containers* - For each neighbor Y of container X, the set of significant containers contains (i) all neighbors of X besides Y and (ii) the union of achievable containers of all neighbors of X besides Y.
- *container heartbeat* - a message sent by container X to its neighbor Y that contains the current state IDs of all *containers significant for Y*.
- *synchronization interval* - a global unit of time applicable to the entire federation. In each container, at every synchronization interval (or sooner), container heartbeats should be sent to neighbors.
- *state delta* - a difference between the new and previous container state.
- *FSSP message* - there are three types of FSSP messages: container state, state delta, and container heartbeat.

The FSSP component in each container creates the following JMS destinations:

- *neighbors topic* - a topic with a common, well-known name and message time to live (cf. Section 3.3) restricted to 1. Messages produced to this topic are consumed only by directly connected containers. The producer and the consumer are both attached to this topic.
- *state queue* - a queue with the name *STATE_ContainerName*. The consumer is attached to this queue.

The neighbors topic is used to discover neighboring containers. The state queue is used for transmitting FSSP messages between neighbors. Container X sends an FSSP message to its neighbor Y by producing the message to the state queue of Y. The message is consumed by the FSSP component in container Y. Each container periodically produces a hello message to the neighbors topic. If the hello message produced by container X is consumed from the neighbors topic in container Y, then Y knows that X is its neighbor. When hello messages from container X stop appearing in the neighbors topic, after some predefined time, it is assumed that X has failed or has disconnected from the federation. Another type of message that may be sent to the neighbors topic is a query message, generated by a container that connects to the federation or by a container that receives either a container heartbeat or a state delta with an unknown state ID. If container X receives a *query message* from container Y, then it sends its container state to the state queue of container Y. All messages (hello, query, and FSSP) contain the name of the sending container.

A basic FSSP assumption is that, in the case of a container state change, only the delta should be sent. After a container state change, a new state ID is generated, and the delta is propagated to all neighbors. Without delay, each neighbor forwards the state delta to all of its neighbors except the one from which the delta was received. All containers process the received delta and update its container state. Eventually, the state delta reaches all containers in the federation. Subsequently, each container in the federation maintains the state by sending a container heartbeat (every synchronization interval) to its neighbors.

Federation state is said to be established when container heartbeats and hello messages are the only messages transmitted by FSSP. Federation state remains established as long as the states of federation containers do not change, no new containers connect to the federation, and no containers disconnect from the federation. The time when federation state is established is referred to as an *epoch*. The time between epochs is referred to as a *state update*. The operation of FSSP in a simple container topology is presented in Figure 7. Arrows represent FSSP messages sent between containers. The figure depicts three different epochs and three different state updates. Each of the presented state updates is triggered by a different cause: a change in container state, a container disconnecting from the federation (failure), or a container connecting to the federation.

The first epoch presented in Figure 7 represents the first established state. Prior to this epoch, the following steps have to be performed by each container: sending query messages, discovering neighbors, sending initial container state, forwarding initial *states* of other containers, and learning about achievable containers and significant containers. Achievable containers are discovered by way of the following algorithm: containers whose states are received from neighbor X are achievable through X. Once epoch 1 begins, container heartbeats with proper state IDs are periodically sent to neighbors by each container, as presented in Figure 7.

The **first state update** is caused by a container state change in container D, which can occur in the following situations: (i) a new service (service change, service removal) available for export is announced—change of *available services*; (ii) a new consumption request (or its withdrawal) is sent—change of *requested consumptions*; and (iii) a consumption response is sent for a newly exported (unexported) JMS destination—change of *exported destinations*. In each case, the algorithm remains the same—the container whose state has changed instantly sends state delta to all neighbors, and neighbors instantly forward this state delta to the rest of the federation (this is depicted in Figure 7, where the new state delta of container D is propagated). Whereas in cases (i) and (iii), new information has to be sent to the whole federation, in case (ii), a new consumption request (or its withdrawal) should only reach the container group. Therefore, in case (ii), a new container state is instantly sent only to those neighbors whose sets of achievable containers encompass at least one container from the given container group. All other neighbors receive a container heartbeat (which is sent at its regular synchronization interval) with a new state ID and an annotation stating that the state change (from the last ID) is caused by an irrelevant consumption request. Therefore, if the first state update in container D is caused by a new consumption request directed to container A, then container D sends the state delta to C, C sends this state delta to D, and C sends

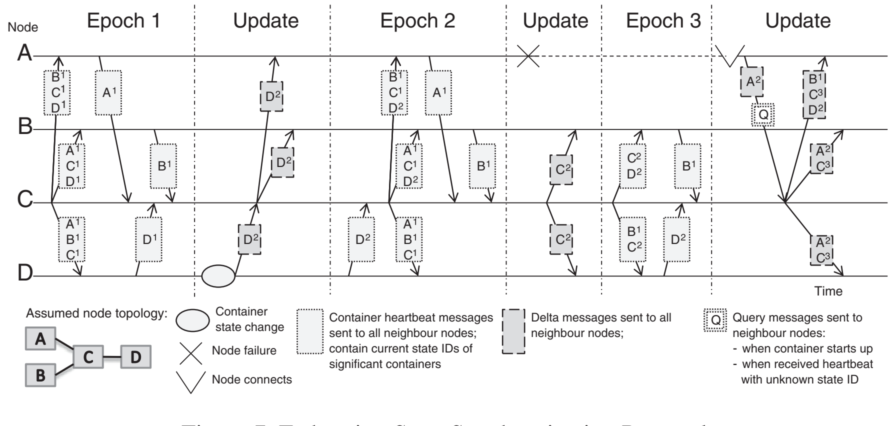

Node
Epoch 1
Update
Epoch 2
Update
Epoch 3
Update
A
B
C
D
Time
Assumed node topology:
Container state change
Node failure
Node connects
Container heartbeat messages sent to all neighbour nodes; contain current state IDs of significant containers
Delta messages sent to all neighbour nodes;
Query messages sent to neighbour nodes:
- when container starts up
- when received heartbeat with unknown state ID

Figure 7. Federation State Synchronization Protocol.

its regular container heartbeat to B. Once the first state update concludes, epoch 2 begins where container heartbeats contain the new state of container D.

The **second state update** is caused by a failure of container A. The failure is detected by container C—there is a lack of hello messages sent by A to the neighbors topic. Upon detection, a new state delta with an updated list of *neighbors* is immediately sent to all neighbors and then forwarded to the rest of the federation (as presented in Figure 7). The *heartbeats* in the third epoch contain the new state ID of C.

The **third state update** is caused by container A reconnecting to the federation. When a container connects to the federation, it sends a *query message* to the neighbors topic. All neighbors respond by immediately sending hello messages to the neighbors topic and all currently known container states to the state queue of the newly connected container. Thus, the new container can synchronize instantly with the state of the whole federation. Subsequently, the new container sends its container state to its neighbors, which forward it to the rest of the federation. Finally, neighbors update their list of *neighbors* and announce their new container state. The process is depicted in Figure 7.

5. EVALUATION

Section 3.4 carried out comprehensive analysis, which proved that RSA implemented over MOM fulfills R1, R2, and R3 requirements by means of advanced MOM features. The purpose of this section is to evaluate whether the proposed ORS model fulfills requirement R4, which concerns TCS scalability aspects. Evaluation is divided into several parts. First of all, we present a preliminary evaluation where metrics important to scalability are identified. Those metrics are used in the second part where several test cases are executed in TCS installation similar to the one from the motivating scenario. Then, we identify the scalability boundaries of our solution by evaluating it in highly dynamic and significantly large OSGi federations. Such extensive evaluation allows us to draw sound conclusions about requirements fulfillment and to discuss ORS generality at the end of this section.

5.1. Experimental environment

The environment in which the test cases were studied was based on the Oracle VM Server for SPARC technology, which provides virtualization of CPUs, network interfaces, and storage. The technology provides the important feature of OS isolation, which makes it possible to deploy multiple OSs simultaneously on a single T-Series server with guarantees that resources allocated to one OS will not be consumed by other OSs. Thirteen virtual machines (VMs) were deployed on five enterprise-class servers: three servers from the Sun Blade System 6000 T6340 family with UltraSPARC T2 Plus processors (64 hardware threads) (Oracle Corporation, Redwood Shores, CA, USA), 32 GB of RAM, and 2 x SAS 146 GB HDD, and two Sun Blade System 6000 T6300 servers with UltraSPARC T2 processors (32 hardware threads) (Oracle Corporation, Redwood Shores, CA, USA), 16 GB of RAM, and 2 x SAS 146-GB HDD. All VMs shared the same configuration (Ubuntu 10.04 OS, eight hardware threads with 1200 MHz clock frequency each, 4-GB RAM, and 10-GB HDD) and were connected in the topology presented in Figure 8. The throughputs of network links were set up with the use of Oracle Crossbow technology, which allows specification of throughput limits for each virtual network interface. VMs between departments were interconnected by WAN links with a bandwidth of 6 Mbps, whereas VMs inside departments were interconnected by LAN links with a bandwidth of 100 Mbps. In each VM, an OSGi container (Equinox version 3.5.1.v20090827) with MOM broker (Apache ActiveMQ version 5.3.0) was deployed. Each TCS element (either TCenter or Unit) was implemented as a bundled OSGi application and deployed in one of the available containers as presented in Figure 8. The message broker in each container was connected to message brokers in all neighboring VMs.

5.2. Preliminary evaluation

In order to design proper test cases (presented in the next section), we needed to perform a preliminary evaluation. The evaluation assessed how messages are propagated in the federation, given

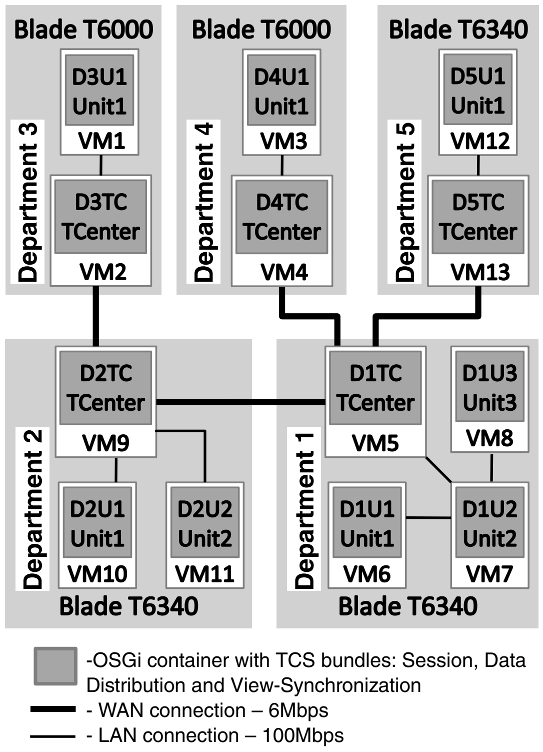
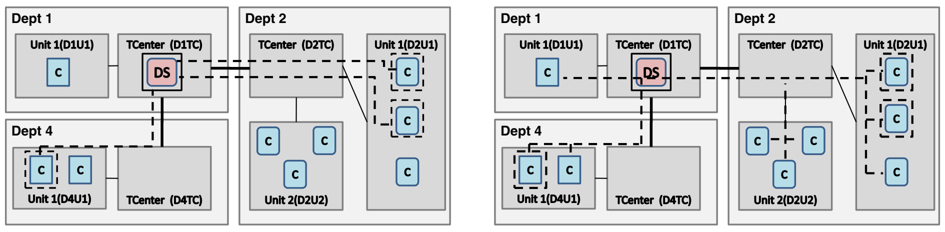


Figure 8. Network topology of experimental environment and mapping of Teleconsultation System (TCS) elements to virtual machines (VMs). WAN, wide area network; LAN, local area network.

different approaches to RSA over MOM. Additionally, preliminary tests were also used to assess the scalability of each approach in the context of the second phase of TCS, including two metrics defined in Section 3.4: *Messages Involved* (number of JMS messages involved in group invocation) and *Memory Used* (memory used for allocation of JMS destinations in the container).

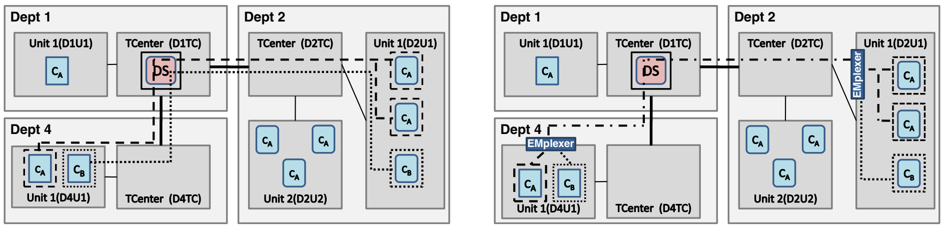
To facilitate analysis, the preliminary evaluation was performed on a subset of federation containers. It pertained to the second phase of TCS in two cases: (i) single teleconsultation with all consultants of the same type (depicted in Figure 9) and (ii) two parallel teleconsultations with two different consultant types C_A and C_B (depicted in Figure 10). In both cases, the evaluation scenario took into account the distribution of medical data (100 MB—chunked into 1000 messages, 100 kB each) from the initiating DS to the target group of consultants.

Figure 9(a,b) presents the observed JMS destinations (dashed lines), created during test execution for a single TS. It can be noticed that the observed destinations are consistent with earlier analyses presented in Section 3.4. The Queue Approach leads to transferring data through the same



(a) Queue approach (b) Topics approach

Figure 9. Queue and Topic Approaches in phase II of a single teleconsultation session. DS, data distribution service; C, consultant service.



(a) Subgroup approach (b) ORS approach

Figure 10. Subgroup and On-demand Remote Service Admin (ORS) Approaches in phase II of two simultaneous teleconsultation sessions.

Table II. Preliminary evaluation results.

RSA over MOM Approach	*Messages Involved*			*Memory Used*
	produced	transmitted	consumed	
Queue Approach	3000	6000	3000	72 kB (nine queues with producers)
Topic Approach	1000	6000	9000	61 kB (one topic with producer)
Subgroup Approach	1000	4000	3000	6.67 MB (C_A: $(2^7 - 1)$ and C_B : $(2^2 - 1)$ topics with producers)
ORS Approach	1000	4000	3000	80 kB (one topic with producer and internal ORS structures)

transmitted - total number of traversed connections between two neighboring brokers for each produced message.
RSA, Remote Service Admin; MOM, message-oriented middleware; ORS, On-demand Remote Service Admin.

section of federation multiple times, that is, twice along the path from D1TC to D2TC and twice along the path from D2TC to D2U1. The Topic Approach results in messages being published to consultants who are not involved in the TCS session, for example, all consultants in D2U2. In the Subgroup Approach, transmission of messages remains efficient (cf. Figure 10(a)), because a dedicated topic is used for each target service group. However, as the number of service groups in the same container group increases, the Subgroup Approach consumes more memory. In contrast, memory usage in ORS remains constant regardless of the number of service groups in a specific container group. This is possible thanks to the limited number of concurrently exported JMS destinations (controlled by the Consumption Listener) and to the EMplexer component, which allows on-demand dynamic memory allocation. Table II shows the aggregated results for *Messages Involved* (produced, transferred, consumed) and *Memory Used* gathered in the course of test cases involving phase II of the TS. Memory consumption was measured by comparison of memory used by single JVM process (OSGi container) before and after creation of queues, topics, and other data structures for remote communication.

Preliminary tests reveal important issues related to the examined approaches, which are valuable for choosing a set of metrics for more comprehensive evaluation. There is a large amount of messages transmitted in TCS; thus *federation data transfer* metric—the overall data count sent through federation nodes—has to be measured. Another issue is the amount of memory used in each approach, which leads to choosing the *memory usage* metric for evaluation. As the R4 requirement constrains the duration of the TS, the *two-way group invocation time* metric is also evaluated. The metric is defined as a time measured from the start of the invocation until its completion in accordance with the specified strategy (Section 3.4). Each proposed metric assesses how the presented approaches scale during each of the TCS phases. Scalability is analyzed in two dimensions: for increasing numbers of services in a single container and for increasing numbers of containers in the federation.

Preliminary evaluation presented in this section shows that the analysis performed in Section 3.4 is correct. It also proves that the on-demand approach solves the problems related to other approaches in both simple scenarios. In the next section, scalability will be evaluated in more complex test cases.

5.3. Test cases in Teleconsultation System scenario

For the purpose of evaluating how *ORS* compares with other approaches, we have constructed several test cases that address specific TCS phases and measure scalability in the context of the metrics introduced earlier. The designed tests are as follows:

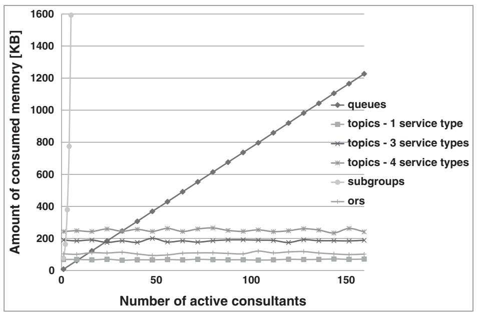
- Evaluating metrics while the number of consultants in containers increases:
  - memory usage (Test Case A);
  - two-way group invocation time (Test Case B);
  - federation data transfer (Test Case C).
- Evaluating metrics while the number of containers in the federation increases:
  - federation data transfer (Test Case D);
  - federation data transfer in the context of load balancing strategies (Test Case E).

Each test was executed several times (*iterations*) with different initial parameters. In tests A, B, and C, the size of the target container group remains constant, the service type count remains constant, the service type group is equal to the service group, and both values increase in consecutive iterations. This means that during each test, all consultants of a given type are invoked. In tests D and E, the size of the target container group increases in each iteration, while each new container extends the service type group by a constant value.

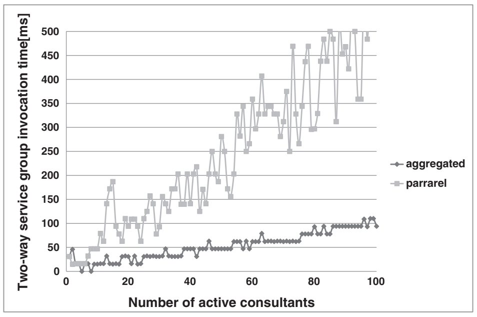
Test Case A pertained to the first phase of TCS—sending a consultation request. Test parameters were as follows: *service type count* = 1, and service group changed between 1 and 20 in each container. Federation topology was the same as in Figure 1. There were eight units in which the number of consultants grew in consecutive iterations. Test case results are depicted in Figure 11(a). In the case of the Subgroup Approach, the metric chart had exponential characteristics. When service group size was 7, memory consumption was of 1.6 MB. When its size was 11, memory consumption grew to 104 MB. The Queue Approach scaled linearly with the number of consultants in the federation regardless of their service type group. Results for the Topic Approach verified that memory consumption remained low regardless of the number of consultants in the target container group. However, as the service type count grew, memory usage increased linearly. ORS also resulted in very low memory usage, but in contrast to the Topic Approach, it did not consume additional memory when the service type count increased.

Test Case B also pertained to the first phase of TCS. During this test, two-way group invocation time was measured. Test parameters were as follows: *service type count* = 1, and service group grew from 1 to 100 in consecutive iterations. Group invocations that retrieved preliminary diagnoses (100 B from each service) were sent between two departmental units, that is, D3U1 and D2U1. Two different approaches were evaluated in this test: aggregated—present in ORS—where the response from each service was aggregated into a single message by EMplexer; and parallel—present in all other approaches—where the response from each service was sent as a separate JMS message. Figure 11(b) depicts test results. It shows that the aggregated approach yields better results and is subject to smaller fluctuations than the parallel approach.

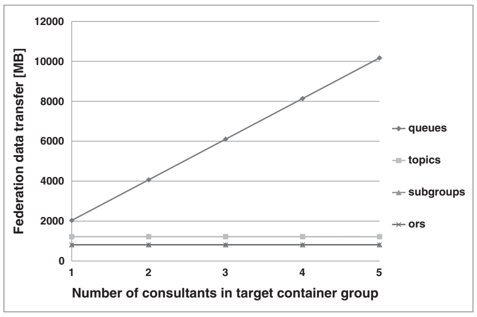
Test Case C was executed during the second phase of TCS, which concerns the distribution of data required for consultation. Federation topology was the same as in Figure 1. The session initiator was located in D3U1. The target container group consisted of units from departments 1 and 2, which resulted in five target containers. Test parameters were as follows: *service type count* = 1, and service group changed between 1 and 5 in each target container. During the test, 100 MB of medical data was sent chunked into 1000 messages (100 kB each). Figure 11(c) depicts test results. In the Queue Approach, the amount of transferred data grew linearly. In the Topic Approach, excessive data flow was registered in departments 4 and 5, causing inefficient transmission. Transmission would be efficient only if all federation nodes were in the target container group. The best results were achieved with the use of ORS and the Subgroup Approach. In spite of this fact, the charts are almost identical. ORS yielded slightly worse results because FSSP generated several kilobytes of



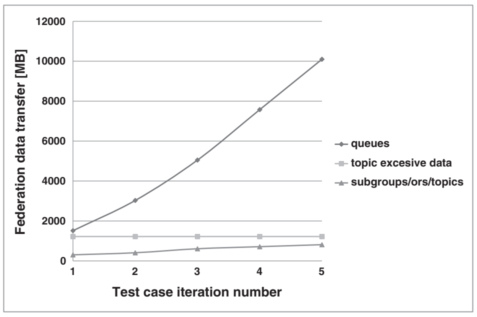
(a) Container memory usage



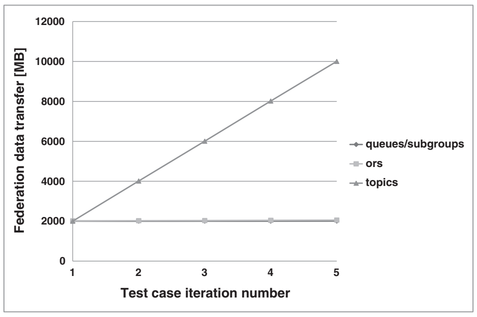
(b) Two-way service group invocation time



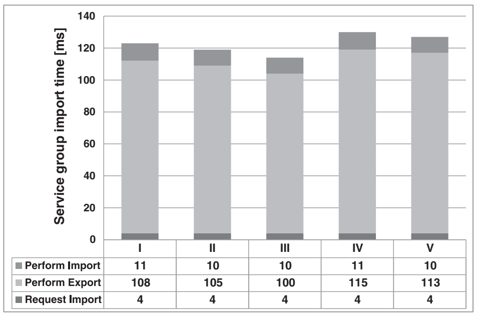
(c) Federation data transfer (increasing number of consultants per container)



(d) Federation data transfer (increasing number of containers in federation)



(e) Federation data transfer (increasing number of containers in federation with load-balancing strategy)



	I	II	III	IV	V
▩ Perform Import	11	10	10	11	10
▨ Perform Export	108	105	100	115	113
■ Request Import	4	4	4	4	4

(f) On-demand approach costs

Figure 11. Charts illustrating test results.

overhead during test execution. Transmission of data in both cases was performed in an efficient way, and messages were sent only to the target container group. Similarly to the Topic Approach, in the case of ORS and the Subgroup Approach, the transferred data volume remained constant and did not depend on the target service group size.

Text Case D also pertained to the second phase of TCS. The initial federation consisted of department 3 TCS elements. Subsequent iterations increased the number of units in the federation in the following way: (i) D2U1, (ii) D2U2, (iii) D1U2, (iv) D1U1, and (v) D1U3. Each new container increased service type group by 5 while the service group was equal to the service type group. During the test, we also measured how each approach scaled when new containers extended the service type group without extending the service group. Results (Figure 11(d)) show that the analyzed metric grows linearly in the Queue Approach. ORS and Subgroup Approach performed more efficient data transmission. Similar to Test Case C, federation data transfer was slightly higher, because of overhead incurred by the FSSP protocol. In subsequent iterations, the amount of transferred data increased but not as significantly as in the Queue Approach. The Topic Approach was also efficient, but only when Cs in new containers also belonged to the target service group. When this was not the case, the Topic Approach resulted in excessive data generation (as indicated in Figure 11(d)).

Text Case E pertained to the third phase of TCS—synchronizing consultation view between participants. In each test iteration, a new department joined the federation in the following order: (i) D3; (ii) D2; (iii) D1; (iv) D4; and (v) D5. During the TS, the consultant synchronized its view by invoking one of the VSs located at the TCenter of its department. Each new container extended the service type group of VSs by 4 while the service group was equal to the service type group. Each synchronization invocation contained a 2-MB data structure. Figure 11(e) depicts test results. In the Topic Approach, the amount of data transferred during synchronization increased linearly, depending on the number of containers in the federation. Other solutions exhibited constant data volume throughout the federation. Through the use of a dedicated queue for each VS, available data were transferred to the TCenter of each department.

The use of ORS carries significant benefits but incurs some costs. First of all, FSSP ensures consistency of federation state in each federation node at the expense of data link throughput. This is evidenced by the results of previously described test cases (Test Case C and Test Case D). Another issue related to our on-demand approach is service group import time. Delays occur during the initial import of the service group from a specific container group. These delays are directly tied to particular phases of the import and export processes (Figures 5 and 6). Figure 11(f) depicts the time spent at each process step (iterations identical to Test Case D). It should be noted that most of this time is spent during the *perform export* step (creation of JMS destination). Once a dedicated destination is created, it can be maintained as long as there is at least one consumption request that requires it. Therefore, all subsequent imports from the target container group do not increase the overhead related to this step—the Consumption Requester does not need to wait for consumption responses. As can be seen, the import time is not dependent on the size of the container group. This is due to concurrent transmissions of consumption responses. A new container appearing in the container group increases the import time only when transmission of its consumption response is longer than the associated transmission for any other container in the group.

In summary, the evaluation shows that ORS mechanisms successfully resolve scalability problems formulated at the end of Section 3.4. In the course of our evaluation, new containers joined the federation, and new services were added to the containers. On each occasion, ORS ensured both low resource consumption and efficient realization of group invocations. It was also proven that ORS costs remain negligible, which makes it a good solution for the realization of TCS.

5.4. Scalability boundaries

The previous section shows that ORS scalability is satisfactory in federations containing up to 13 containers and hundred services. However requirement R4 refers to scalability in the context of several dozen departments. Therefore, this section tries to identify the size and dynamism of systems to which ORS is efficiently applicable. During the evaluation, we focus on the FSSP, which is the core element of ORS.

In previous cases, FSSP bandwidth consumption was hard to estimate because it was relatively small compared with the amount needed for transferring medical data (second phase of TS). To precisely identify the costs of FSSP, we are simulating OSGi federations containing up to 550 containers in which frequent changes occur. The simulation is realized by extracting FSSP implementation from ORS and by wrapping it in a single Java class. Each instance of such a class is referred to as an *s-node*. The purpose of the s-node is to simulate an OSGi node on which FSSP is deployed. S-nodes communicate with each other through regular queues available in Java collections. Operations on those queues are enriched with some artificial delays simulating latency and bandwidth present in 6-Mbps WLAN links. S-nodes are connected into a graph, which simulates an actual OSGi federation. The simulation is executed on the Blade T6340 in a single JVM process.

During the simulation, we create a graph that is a complete binary tree with an initial size equal to 5 s-nodes. In each iteration, the s-node tree is extended by a new s-node. It is added as a child of the leftmost node, which is not a leaf and does not yet have two children. Iterations are stopped when the size of tree reaches 550 nodes. To describe tests executed in each iteration, three concepts are defined in the context of the s-node graph:

- *distance* between given s-nodes - the number of edges of s-node graph in the shortest path connecting those s-nodes;
- s-node graph *diameter* - the longest distance in a given graph;
- *state synchronization time* - the time in which a container state changes from a particular s-node will be propagated to all other nodes.

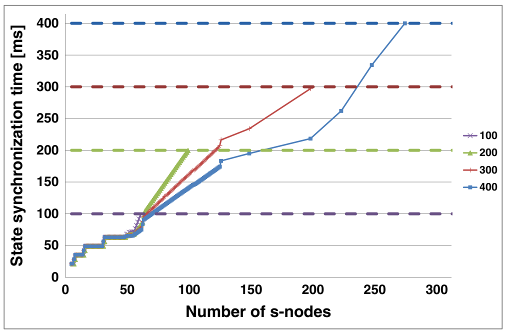
In each iteration, two tests are executed. In the first test, we measure the maximum FSSP bandwidth consumption in the s-nodes graph. In the second test, we measure average state synchronization time. To simulate the state synchronization time with the highest possible duration, we measure propagation time of container state changes between s-nodes whose distance equals the graph diameter.

During each test, we simulate changes of container states occurring in the range from 100 to 1000 ms with a step of 100 ms. These periods have been selected to present FSSP behavior in a situation when the state synchronization time is close to the interval of changes. The state delta messages are about 170 bytes. Each iteration is run for 13.5 s: a 3.5-s sleep period to allow propagation of messages in the federation and a 10-s period to measure bandwidth consumption and state synchronization time.

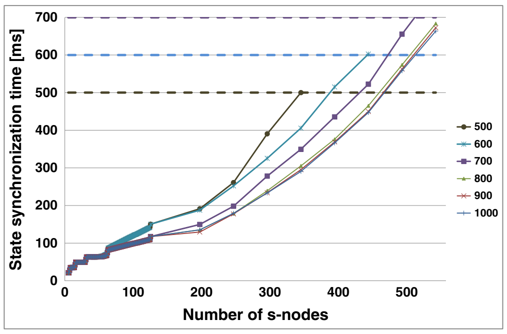
Figure 12(a,b) presents the state synchronization time in the s-node graph. Horizontal lines on the chart represent the period of container state changes for a particular series. If the state synchronization time equals the period of container state changes, the federation size reaches its boundary. Above the boundary, FSSP will not work properly, because there is not enough time to update the container state in all federation containers. For lower values of container state change periods 100, 200, 300, 400, 500, 600, and 700 ms, the boundary sizes are 60, 99, 200, 277, 350, 450, and 550 nodes, respectively. In the case of higher values of container state change periods (greater than 700 ms), the boundaries are not reached in the experiment.

The results of the bandwidth consumption test are presented in Figure 13(a,b). It shows that, even for extremely low values of container state change periods, the FSSP does not saturate more than 13% of the link. Because the state synchronization time is linearly dependent on the number of nodes, even federations built of a few hundred containers can be efficiently maintained by the FSSP, and still, link saturation will reach no more than 10%.

The evaluation shows that scalability boundaries depend directly on the dynamism of the system in which the FSSP is used. The container state change period in the TCS itself is rather long; thus, in this case, the FSSP will probably never reach boundaries. However, if TCS is integrated with OSGi-based systems of remote patient data acquisition, described in Section 2, the container state change period may significantly shorten. Of course, in a real-world scenario, it is very unlikely that

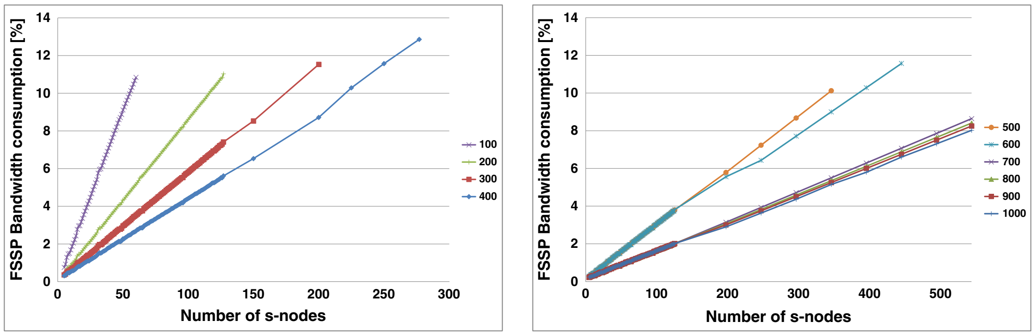


(a) State synchronization time for periods 100 - 400 ms of container state changes with an increasing number of s-nodes



(b) State synchronization time for periods 500 - 1000 ms of container state changes with an increasing number of s-nodes

Figure 12. Charts presenting federation state synchronization time.



(a) The bandwidth consumed by FSSP on a single s-node for periods 100 - 400 ms of container state changes with an increasing number of s-nodes

(b) The bandwidth consumed by FSSP on a single s-node for periods 500 - 1000 ms of container state changes with an increasing number of s-nodes

Figure 13. Charts presenting the Federation State Synchronization Protocol (FSSP) bandwidth consumption.

ORS would have to deal with container state change periods shorter than 500 ms. Such short periods would mean that, in each node of the federation, either service import or service export occurs more frequently than twice a second. Taking this into consideration, we can conclude that the ORS along with the FSSP can be used in federations up to a few hundred containers. Providing such scalability fulfills requirement R4 (Section 2) and ensures that our solutions achieves the stated goal.

5.5. *Discussion*

The evaluation shows that our solution fulfills the requirements set up by TCS; however, it is not clear what other class of systems could be addressed by ORS. To answer this issue, we analyze ORS in the context of each requirement. The motivation behind each requirement is explained and implied; ORS design is generalized by comparison with other middleware solutions.

The container network described in requirement R1 can be compared with service overlay networks [26–29]. In such approaches, an overlay is set up over Internet Protocol network by deploying overlay nodes onto the physical infrastructure. The services exposed on one overlay node can be consumed on the others. In TCS, the network of MOM brokers can be perceived as overlay where OSGi services are exposed and consumed. The following are some of fields where an overlay networks are exploited: application-level multicast, advanced routing control, detailed control of end-to-end QoS, content delivery, and multimedia streaming. In the TCS, the main purpose of using the MOM overlay was to communicate easily between hospital departments (where network elements such as firewalls and network address translations could make it difficult) but more importantly to effectively transfer *TS* data and queries of *Cs* through WAN and LAN links.

Without the presence of the overlay, the TS data and calls to *consultant service groups* would be transferred inefficiently; transfer duplication similar to Queue Approach would occur (cf. Figure 9(a)). This problem is specifically addressed by the aforementioned application-level multicast. Papers such as [30–33] propose highly efficient realization of an application-level multicast based on the publish–subscribe paradigm. The application-level multicast solutions already proposed provide such advanced features as multicast tree calculation on the basis of message content [30], bandwidth optimization of multicast trees [31], and adaptation of multicast tree to changing topology [34]. In the TCS, such advanced features were not as important as using stable, widely verified middleware, JMS broker. JMS brokers, both open source and commercial, were improved over many years of experience in demanding production environments, where reliability and scalability were a major concern. Therefore, building ORS on the foundation of JMS allows it to target systems with similar demands.

Invocation of service groups required by R2 is concerned mainly with expressiveness and accordance with SOA. As we concluded in Section 3.4, some of the aggregating algorithms in a simultaneous strategy (R2a) of service group invocations require an additional mechanism. One solution could be hiding service group invocations under regular service calls. However, such an approach would require changing the service interface—the type of returned object would have to be changed to a collection that could aggregate results from the service group. That would break the basic principle of SOA [6], which states that the service contract should reflect the functionality provided to a consumer by a service provider. The service contract should not be modified for the purpose of aggregations on the consumer side.

To solve this problem, we have introduced ASG API (Section 4.1), which allows a consumer to request service import (cf. Section 4, Step 2 of ORS import/export process) with the specific intents and to use provided proxies in conjunction with ASG services. Adding intents by a consumer and provider can be interpreted as annotating the service contract for the purpose of establishing non-functional capabilities. Such a design of the API enables retaining expressiveness of the source code and, more importantly, compliance with the SOA principle of service contract immutability. Current initiatives related to Asynchronous API of RSA (Eclipse Communication Framework [ECF] [35], Peter Kriens [36]) give the impression that the ASG API is in line with the direction in which Enterprise OSGi Specification can evolve in the future. We are assuming that the motivation for load balancing (R2b) and failover (R2c) strategies of service group invocations does not have to be discussed as their usefulness is well understood and evaluated in many publications.

The purpose of scalability requirement (R4) is to open the perspective for TCS extension in terms of the amount of hospital departments, amount of consultants, and amount of simultaneous TSs. It is hard to specify the realistic need for the size of TCS federation. To make an estimation, we have looked up to some companies offering telemedical services: MedWeb (http://www.medweb.com/), TeleRAD IT (http://www.teleradit.com/), NightHawk Radiology Services and vRad (http://www.nighthawkrad.net/). The last mentions support 2700 health-care facilities in all states of America by providing access to 325 radiologists. It does not necessarily mean that TCS federation of that size is needed. Probably, only limited subsets of those facilities will be willing to connect with each other for the purpose of providing teleconsultation services. In our opinion, it is a reasonable justification for the requirements of TCS scalability to be up to a few hundred containers and a few thousand services.

In cases where linear scalability of the FSSP would still be insufficient, an additional extension could be implemented. For instance, in the process of an election from all nodes in a federation, a subset of supernodes could be chosen. In case of import/export operations, all ordinary nodes would communicate only with the nearest supernode. Each supernode would be responsible for maintaining federation state by communicating with other supernodes. This improvement could significantly reduce the overhead of FSSP as well as provide better linear scalability. We are planning to introduce such an extension to the FSSP in future work.

Each TCS requirement has its motivation and some context, which defines system classes addressable by ORS. First of all, ORS can address dynamic environments when many services could be registered/unregistered with relatively high frequency. We could imagine a stock exchange system where the registration of some service manifests in presenting an offer for a given stock. ORS could handle the discovery and invocation of such services up to the scalability boundaries (Section 5.4). Secondly, systems similar to the ones addressed by the application-level multicast can also be addressed by ORS, especially systems requiring frequent data distribution to many recipients such as dissemination of geospatial data [37], on-demand multimedia streaming [38], workload distribution [39], dissemination of personalized data [40], and software distribution [41] (in the context of OSGi, such ORS usage would be valuable for bundle distribution). Other examples of similar systems could be solutions requiring interactive collaboration between multiple participants such as distributed whiteboard [42] and audio/video conferencing [43]. Thirdly, thanks to the leveraging proven features of JMS brokers and providing load balancing and failover, ORS can address production systems requiring high reliability. Finally, the ASG API provided by ORS ensures expressiveness and compliance with the principles of SOA. It can therefore be concluded that ORS proposes a valuable extension of the RSA specification that is usable in a wide range of systems.

6. RELATED WORK

Prior to deciding upon the implementation of a new OSGi distribution provider, the implementations currently available were evaluated in the context of the requirements of the TCS presented (Section 2). Before the RSA specification was created, work on R-OSGi [10] yielded several concepts crucial for RSA, such as dynamic service proxies, *discovery listener* service, distributed service registry, and on-demand service discovery. R-OSGi proved that loose coupling of bundles makes them perfect candidates for distribution units, which collaborate remotely in a fully transparent way through the OSGi service layer. R-OSGi uses Service Location Protocol to support a distributed registry and introduces its own Transmission Control Protocol-based communication protocol for invoking services and transmitting events. After the RSA specification was released, R-OSGi was not upgraded; therefore, its current implementation is not compliant with the specification. The R-OSGi approach does not provide invocations of service groups and does not ensure appropriate scalability (containers cannot be federated); therefore, it does not meet TCS requirements.

Following the release of the RSA specification, Apache CXF provided a reference implementation [44]. This implementation uses Web services to expose and invoke services, along with ZooKeeper—a hierarchical distributed repository for service discovery. Another realization of Web service-based RSA is proposed in [45], which uses this implementation to integrate OSGi with the Devices Profile for Web Services. The ECF [46] has also released its own modular implementation of Remote Services, which supports many different distribution providers, including Simple Object Access Protocol Web services, REpresentational State Transfer Web services, and JMS. None of the presented Web service-based RSA implementations are appropriate for TCS because they do not provide group communication, and transferring binary data through Web services is not efficient. The JMS ECF provider most closely addresses the requirements of TCS because of its service group invocation feature. However, the provider only supports service exports/imports to a single topic or queue. As described in the Topic and Queue Approaches in Section 3.4, it does not provide the appropriate scalability required by TCS.

Eclipse Communication Framework provides Asynchronous API [35] that is different from our ASG API. The important element of ECF API is the *IAsyncRemoteServiceProxy* interface. If a service consumer wants to asynchronously invoke a method *ret1 methodA(arg1, , argN)* of service with interface *packageB.B*, a new service interface *packageB.BAsync* has to be added. The interface *packageB.BAsync* has to extend *IAsyncRemoteServiceProxy* and has to contain the following methods:

- *void methodAAsync(arg1, , argN, IAsyncCallback callback)*
- *IFuture methodAAsync(arg1, , argN)*

When *packageB.BAsync* interface is available on the consumer side, during import, ECF automatically prepares a proxy capable of asynchronous invocations. A consumer can either invoke a method with a callback argument or use a returned *IFuture* object to poll for response. Peter Kriens points out in the following source [36] the drawbacks of the ECF approach. The approach is not entirely appropriate because it forces the consumer to modify the package of service interface by adding a new asynchronous interface (or, alternatively, the producer is forced to do so). Additionally, the new interface slightly changes the service contract—new arguments and return values—which is not entirely inline with the principles of SOA. Our ASG API does not have deficiencies of ECF.

The service group invocation requirement can be satisfied by group communication features, which are widely addressed in current publications. Therefore, in the course of middleware analysis and selection for RSA implementation, we have taken into account a research on group communication in the following solutions: Common Object Request Broker Architecture [47], [48], RMI [49] [24], and Jini [50]. All presented approaches propose a more or less efficient group invocation mechanism. However none of them discuss the problem in the context of a network of brokers, which is required to meet requirement R1 of TCS. These facts have led us to conclude that the RSA over MOM approach proposed in the ORS model is a valuable research contribution.

Publications [13, 14, 51] provide examples of health care and telemedicine systems that use the OSGi platform as a significant architectural element. [51] is an especially interesting paper because its authors have integrated OSGi with MOM. Unfortunately, their study does not use advanced MOM features and performs integration by direct calls to the JMS API. Therefore, it does not leverage the abstraction of OSGi Remote Services and binds the implementation to a single distribution provider.

Integration of OSGi and MOM is also mentioned in a research related to sensor networks, where OSGi is used as a gateway technology. The research presented in [52] adopts OSGi and MOM in MyHome—a framework for the smart home. The authors used OSGi for implementing smart home software modules and MOM for connecting these modules to home appliances. However, their study has drawbacks similar to [51]—the JMS API is used directly, without the Remote Services abstraction. In [53], the authors presented a case study that adopts MOM for communication between wireless sensors to solve issues such as information exchange, network protocol conversion, and integrity among multivendor products. In this paper, OSGi is advanced as a good solution for standardizing life cycle management of sensor-related components. A similar approach is presented in [54], where a message-oriented application model is designed for data dissemination in a ubiquitous system and OSGi is used for executing some of its applications and actuators.

The subject of OSGi and MOM is also touched upon in the work [55], which proposes a new OSGi distribution provider called Remote Batch Invocation (RBI). RBI supports batching multiple invocations of remote OSGi services by expressing them in a proposed batch language. Each batch is then dispatched as a single remote call. This mechanism provides significant efficiency improvements. RBI is compared with several other approaches including R-OSGi and MOM. MOM turns out to be the slowest solution; however, the authors did not use advanced MOM features, which are the focal point of our work. The introduction of RSA over MOM proposed in ORS might yield results comparable with RBI for the specific problem space addressed by the authors.

7. CONCLUSIONS

Implementing distributed OSGi over MOM is not a trivial task when the invocation of operations over groups of services is considered. A good starting point for constructing OSGi federations is the RSA specification. Unfortunately, this specification does not address the issues of dynamic service group invocations. A detailed analysis of this problem shows that processing invocations referring to a dynamic group of OSGi services causes scalability issues related to memory consumption and federation data transfer.

An efficient solution requires extending RSA components, such as Topology Manager and Discovery modules, to enable exporting and importing services on demand. This concept is the foundation of the enhanced RSA model called ORS, which, together with the FSSP, constitutes a complete solution. FSSP plays a central role in the proposed implementation as the dynamic nature of OSGi federations requires groups of exported services to be dynamically updated. This protocol is based on the well-known Virtual Synchrony [56] concept and works correctly; however, its functionality remains a promising area of further study. Additionally, in the course of further study, we would like to refine the ORS to make it applicable in the area of mobile devices.

An important lesson derived from ORS implementation is that efficient and scalable mapping of ORS to MOM requires advanced features provided by the network of brokers such as failover and distributed destination, which are not covered by the standard JMS specification. The proposed methodology of ORS development, based on the specific use of a TCS, results not only in identifying crucial problems with RSA over MOM but also allows for validation of the proposed solution. Theoretical analysis and performance tests have shown that ORS satisfies all the requirements posed by such highly demanding systems as TCS.

ACKNOWLEDGEMENT

The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08 and UDA-POKL.04.01.01-00-367/08-00.

REFERENCES

1. Sun Microsystems. Java remote method invocation specification. *Technical Report Revision 1.50*, Sun Microsystems Inc., Mountain View, California, October 1998.
2. Object Management Group. The common object request broker: architecture and specification. *Technical Report Revision 23*, OMG Inc, Framingham, Mass, March 1998.
3. Arnold K, O'Sullivan B, Scheifler R, Waldo J, Wollrath A. *The Jini Specification*, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.
4. The OSGi Alliance. About the OSGi service platform, July 2004.
5. The OSGi Alliance. OSGi Service Platform Core Specification - Release 4, Version 4.2, September 2009.
6. Erl T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR: Upper Saddle River, NJ, USA, 2005.
7. Hall R, Cervantes H. Challenges in building service-oriented applications for OSGi. *Communications Magazine, IEEE* May 2004; **42**(5):144–149.
8. The OSGi Alliance. OSGi Service Platform Compendium Specification - Release 4, Version 4.2, September 2009.
9. The OSGi Alliance. OSGi Service Platform Enterprise Specification - Release 4, Version 4.2, March 2010.
10. Rellermeyer JS, Alonso G, Roscoe T. R-OSGi: distributed applications through software modularization. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, MIDDLEWARE2007. Springer-Verlag: Berlin, Heidelberg, 2007; 1–20.
11. Cała J, Czekierda L, Nowak M, Zieliński K. The practical experiences with deployment of advanced medical teleconsultation system over public IT infrastructure. *21th IEEE International Symposium on Computer-Based Medical Systems* June 2008.
12. Zhang J, Stahl J, Huang H, Zhou X, Lou S, Song K. Real-time teleconsultation with high-resolution and large-volume medical images for collaborative healthcare. *Information Technology in Biomedicine, IEEE Transactions on* 2000; **4**(2):178–185.
13. Kim NH, Jeong YS, Song SJ, Shin DR. Middleware interoperability based mobile healthcare system. In *Advanced Communication Technology, the 9th International Conference on*, Vol. 1, 2007; 209–213.
14. Chen IY, Huang CC. A service-oriented agent architecture to support telecardiology services on demand. *Journal of Medical and Biological Engineering* 2005.
15. Chen IY, Huang CC. Remotely manageable electrocardiogram measurement system for home healthcare using OSGI framework. *Journal of Medical and Biological Engineering* 2004; **24**(3).
16. Chih-Jen H. Telemedicine information monitoring system. *e-health Networking, Applications and Services, 2008 HealthCom 2008 10th International Conference on*, 2008; 48–50.
17. Wang F, Docherty L, Turner K, Kolberg M, Magill E. Services and policies for care at home. *Pervasive Health Conference and Workshops, 2006*, 2006; 1–10.
18. Nee O, Hein A, Gorath T, Hulsmann N, Laleci G, Yuksel M, Olduz M, Tasyurt I, Orhan U, Dogac A, *et al*. SAPHIRE: intelligent healthcare monitoring based on semantic interoperability platform: pilot applications. *Communications, IET* 2008; **2**(2):192–201.
19. Clemensen J, Larsen SB, Bardram JE. Developing pervasive e-health for moving experts from hospital to home. *IADIS International Journal of WWW / Internet* 2005; **II**(2):57–68.
20. Shaikh A, Memon M, Memon N, Misbahuddin M. The role of service oriented architecture in telemedicine healthcare system. *Complex, Intelligent and Software Intensive Systems, 2009 CISIS '09 International Conference on*, 2009; 208–214.
21. Vasilescu E, Mun S. Service oriented architecture (SOA) implications for large scale distributed health care enterprises. *Distributed Diagnosis and Home Healthcare, 2006 D2H2 1st Transdisciplinary Conference on*, 2006; 91– 94.
22. Lopes D, Abreu B, Batista W. Innovation in telemedicine: an expert medical information system based on SOA, expert systems and mobile computing. In *Advances in computer and information sciences and engineering*, Sobh T (ed.). Springer: Netherlands, 2008; 119–124.
23. Nitzsche J, Lessen Tv, Leymann F. WSDL 2.0 message exchange patterns: limitations and opportunities. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*. IEEE Computer Society: Washington, DC, USA, 2008; 168–173.
24. Montresor A, Zamboni MA. The Jgroup reliable distributed object model. *In Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS99*, 1999; 389– 402.
25. Pyarali I, Spivak M, Cytron R, Schmidt DC. Evaluating and optimizing thread pool strategies for real-time CORBA. *SIGPLAN Not* August 2001; **36**:214– 222.
26. Castro M, Druschel P, Kermarrec AM, Rowstron A. One ring to rule them all: service discovery and binding in structured peer-to-peer overlay networks. In *Proceedings of the 10th Workshop on ACM SIGOPS European workshop*, EW 10. ACM: New York, NY, USA, 2002; 140– 145.
27. Jin J, Nahrstedt K. Large-scale service overlay networking with distance-based clustering. In *Middleware 2003 Lecture Notes in Computer Science*, Vol. 2672, Endler M, Schmidt D (eds). Springer: Berlin / Heidelberg, 2003; 998–998.
28. Duan Z, Zhang ZL, Hou Y. Service overlay networks: SLAs, QoS, and bandwidth provisioning. *Networking, IEEE/ACM Transactions on* 2003; **11**(6):870–883.

29. Fan J, Ammar MH. Dynamic topology configuration in service overlay networks: a study of reconfiguration policies. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006; 1–12.
30. Banavar G, Chandra T, Mukherjee B, Nagarajarao J, Strom RE, Sturman DC. An efficient multicast protocol for content-based publish-subscribe systems. *Distributed Computing Systems, International Conference on* 1999; **0**:0262.
31. Jannotti J, Gifford DK, Johnson KL, Kaashoek MF, O'Toole JW, Jr. Overcast: reliable multicasting with on overlay network. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation*, Vol. 4, OSDI'00. USENIX Association: Berkeley, CA, USA, 2000.
32. Pietzuch PR, Bacon JM. Hermes: a distributed event-based middleware architecture. *Distributed Computing Systems Workshops, International Conference on* 2002; **0**:611.
33. Pietzuch PR, Bacon J. Peer-to-peer overlay broker networks in an event-based middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-based Systems*, DEBS '03. ACM: New York, NY, USA, 2003; 1–8.
34. Kwon M, Fahmy S. Topology-aware overlay networks for group communication. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '02. ACM: New York, NY, USA, 2002; 127–136.
35. Scott Lewis. OSGi Remote Services and ECF - Asynchronous services, April 2010. http://eclipseecf.blogspot.com/2010/04/osgi-remote-services-and-ecf.html.
36. Peter Kriens. Calling your cake and sending it too, April 2010. http://www.osgi.org/blog/2010/04/calling-your-cake-and-sending-it-too.html.
37. Hu L, Li P, Wang Y. The design and implementation of a SOA-based data exchange middleware. *Service Sciences (ICSS), 2010 International Conference on*, 2010; 39–42.
38. Gelman A, Kobrinski H, Smoot L, Weinstein S, Fortier M, Lemay D. A store and forward architecture for video-on-demand service. In *Communications, 1991. ICC '91, Conference Record. IEEE International Conference on*, Vol. 2, 1991; 842–846, DOI: 10.1109/ICC.1991.162477.
39. Tan J. Cost-efficient load distribution using multicasting. *Cluster Computing, 1999. Proceedings 1st IEEE Computer Society International Workshop on*, 1999; 202–208.
40. Shah R, Jain R, Anjum F. Efficient dissemination of personalized information using content-based multicast. In *INFOCOM 2002 Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies Proceedings. IEEE*, Vol. 2, 2002; 930–939.
41. Gumbold M. Software distribution by reliable multicast. *Local Computer Networks, Annual IEEE Conference on* 1996:222–231.
42. Mccanne S. A distributed whiteboard for network conferencing. *UC Berkeley CS 268 Computer Networks term project* 1992, DOI: 10.1.1.48.5173.
43. Amad M, Haddad Z, Khenous L, Kabyl K. A scalable based multicast model for P2P Conferencing applications. *Ultra Modern Telecommunications Workshops, 2009. ICUMT '09 International Conference on*, 2009; 1–6.
44. The Apache Software Foundation. Apache CXF Distributed OSGi. http://cxf.apache.org/distributed-osgi.html.
45. Fiehe C, Litvina A, Luck I, Dohndorf O, Kattwinkel J, Stewing FJ, Kruger J, Krumm H. Location-transparent integration of distributed OSGi frameworks and Web services. In *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops*, WAINA '09. IEEE Computer Society: Bradford, 2009.
46. Eclipse Foundation. Eclipse Communication Framework Project. http://www.eclipse.org/ecf/.
47. Felber P, Garbinato B, Guerraoui R. The design of a CORBA group communication service. *Reliable Distributed Systems, 1996. Proceedings, 15th Symposium on*, 1996; 150–159.
48. Mishra S, Fei L, Lin X, Xing G. On group communication support in CORBA. *Parallel and Distributed Systems, IEEE Transactions on* Feb 2001; **12**(2):193– 208.
49. Cazzola W, Ancona M, Canepa F, Mancini M, Siccardi V. Enhancing Java to support object groups. *In: Proceedings of the 3rd conference on Recent Object-Oriented Trends (ROOTS02)*, 2002.
50. Montresor A, Davoli R, Babaoglu O. Enhancing Jini with group communication. *Distributed Computing Systems Workshops,International Conference on* 2001; **0**:0069.
51. Chen IY, Tsai CH. Pervasive digital monitoring and transmission of pre-care patient biostatics with an OSGi, MOM and SOA based remote health care system. *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, 2008; 704–709.
52. Huang HY, Teng WC, Chung SL. Smart home at a finger tip: OSGi-based MyHome 2009:4467–4472.
53. Ahn S, Chong K. A case study on Message-Oriented Middleware for heterogeneous sensor networks. In *Embedded and Ubiquitous Computing, Lecture Notes in Computer Science*, Vol. 4096, Sha E, Han SK, Xu CZ, Kim M, Yang L, Xiao B (eds). Springer: Berlin / Heidelberg, 2006; 945–955.
54. Liao CF, Jong YW, Fu LC. Toward a message-oriented application model and its middleware support in ubiquitous environments. *International Journal of Hybrid Information Technology* 2008; **1**(3).
55. Kwon YW, Tilevich E, Cook W. An assessment of middleware platforms for accessing remote services. *Services Computing (SCC), 2010 IEEE International Conference on*, 2010; 482–489.
56. Birman K. A history of the virtual synchrony replication model. In *Replication, Lecture Notes in Computer Science*, Vol. 5959, Charron-Bost B, Pedone F, Schiper A (eds). Springer, 2010; 91–120.