A Concept for Flexible Event-Driven Invocation of Distributed Service Compositions

Karen Walzer, Jürgen Anke, Alexander Löser
SAP Research
Chemnitzer Strasse 48
01187 Dresden, Germany
{karen.walzer, juergen.anke, alexander.loeser}@sap.com

Huaigu Wu
SAP Research
111 Duke Street, Suite 2100
Montreal (Quebec) H3C 2M1, Canada
huaigu.wu@sap.com

Abstract

Currently, flexible service compositions are invoked in a centralised manner by process execution engines. Although this approach is widely used for orchestrating web services, it lacks scalability and incurs considerable overhead in network communication. This is especially disadvantageous in scenarios with embedded systems and other resource-constrained devices, mainly due to restrictions in network bandwidth or power supply. We present an application scenario to illustrate the drawbacks of centralised service invocation in these cases. Decentralised invocation mechanisms are a promising approach to overcome these problems. Based on the identification of service composition characteristics in distributed environments, we propose a decentralised mechanism using events and subscriptions. This mechanism can be employed for invoking distributed service compositions while offering improved scalability and efficiency. We conclude with discussing the advantages and restrictions of our solution compared to the centralised approach.

1 Introduction

Smart items are physical products that include product embedded information devices (PEIDs), e.g. embedded systems, or RFID tags. For application domains such as Product Lifecycle Management (PLM), enterprise applications can benefit greatly from accessing data on smart items. This can replace error-prone manual data input with automatic data aquisition to support maintenance planning, recycling decisions, and even improvements in product design. As it is not reasonable to integrate mechanisms for accessing PEIDs into business applications, placing this functionality into a middleware is a useful approach. The middleware and the PEIDs form the *smart item environment*.

The middleware functionality is logically separated and physically distributed among a number of nodes. Besides hosting parts of the middleware functionality, they provide an execution environment for components that offer services for data processing. These components are of relative small granularity and function-oriented. They can be flexibly deployed onto all of the nodes, including suitably equipped PEIDs. This allows for early processing of PEID data, and reduces the amount of data which has to be transmitted from the PEID to the backend systems [3]. To accommodate for different application-specific data processing requirements, available services can be combined into *service compositions*. Service compositions describe how distributed services need to interact in order to achieve a common desired result.

An example service composition is shown in Figure 1 in form of a service dependency graph. In this scenario, a number of processing steps are performed on data from various data sources available at the PEID of a vehicle (oil pressure sensor, engine speed sensor, engine temperature sensor, mileage counter). The service composition is used to monitor the vehicle's operational status. More details on this example in the context of deployment planning for the used components can be found in [2].

To facilitate flexible composition of services provided by distributed components, two key functionalities have to be provided by the infrastructure:

1. **Definition of Service Compositions:** The cooperation of services must be described in a way that is machine-readable and allows for easy configuration, i.e. it must be possible to add, modify, and remove service compositions without access to component source code.

2. **Invocation Mechanism:** Once service compositions are defined, they have to be invoked by a mechanism that allows the coordinated execution of services based on the composition description.

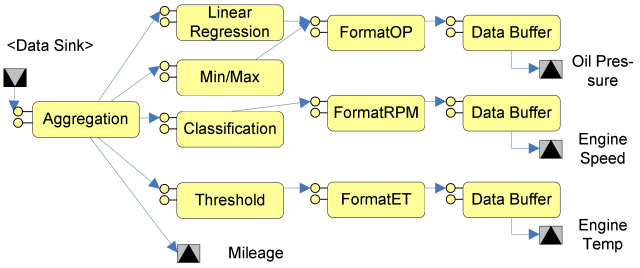

Figure 1. Service composition for the monitoring of a vehicle's operational status

There are two main paradigms for invocation of service compositions: Request/Response and Event-Driven. In the first case, a client application places a request to retrieve the current operational status of the vehicle, i.e. it "pulls" the data. With the event-driven paradigm, the services would be executed when they receive an event notification they are subscribed to. In our example, all services would subscribe for a notification of "DataChange" events at the services they depend on. Therefore, the data would be "pushed" from the source to the sink. For instance, if the *Oil Pressure* changes in the scenario depicted in Figure 1, the *DataBuffer 1* service receives an event notification with the new sensor data. This changes the data in the buffer, causing a notification to be sent to the *FormatOP* service, and so on.

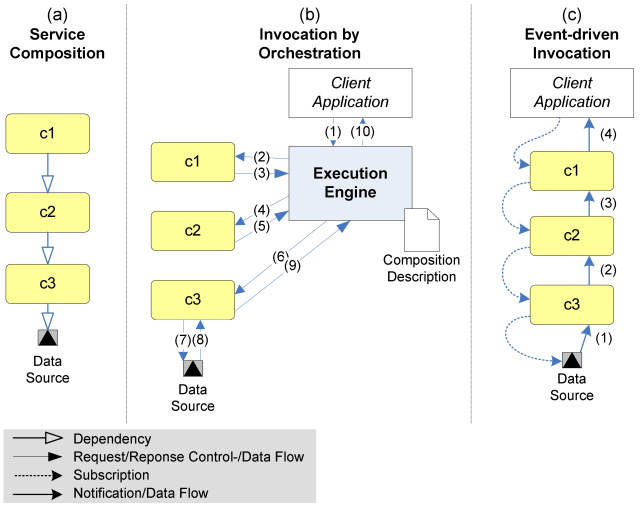

Figure 2. Invocation Mechanims for Service Compositions

Figure 2 shows side-by-side comparison (b) centralised request/response and (c) event-driven invocation for a simple composition depicted in Figure2a. The numbers on arrows denote the sequence of invocations and therewith the number of steps required to invoke the composition. Whereas event-based invocation requires setting up subscriptions initially, the number of steps during execution is much smaller.

Request/response-based invocation has limitations in terms of scalability and network load, which are particularly disadvantageous for applications that access data from PEIDs. Event-driven invocation promises to mitigate these problems by decentralised control and improved network efficiency.

In this paper, we present a concept for decentral event-driven service invocation, and discuss its benefits and limitations. The remainder of the paper is structured as follows: First, we analyse the drawbacks of request/response-based invocation based on a central execution engine, formulate a problem statement, and derive requirements. Afterwards, the related work is reviewed, followed by the presentation of our solution. The proposed solution is put in the context of our current work, along with an evaluation of its practical applicability. The paper concludes with a discussion on limitions and future work.

2 Problem Analysis

Service compositions used for data processing in smart item environments possess some characteristics that have to be taken into consideration when evaluating invocation mechanisms. Firstly, they are run in an environment where network capacity is limited, especially between smart items and middleware nodes. Secondly, these network connections might be intermittent, i.e. only temporarily available. This is problematic when a service becomes unavailable during an ongoing invocation, because the node hosting the service gets disconnected from the network. Thirdly, there are services that depend on data sources that change in different intervals. In applications that monitor the status of smart items, it seems a more natural approach to let the change of underlying data trigger the invocation of data processing. If client applications trigger the invocation of service compositions, the underlying data might be unchanged since the previous request. Fourthly, there are potentially a number of service compositions to be executed in parallel, therby affecting several nodes.

2.1 Request/Response-based invocation

The coordination of service invocation in a service composition is accomplished by a central execution engine according to a composition description that is interpreted by the engine (see Figure 2b). The central engine uses a request/response invocation mechanism to communicate with

the services of the composition. It invokes the required services and waits for their response to continue in its workflow. The central engine is responsible for the complete coordination of data and control flow between the components. This concept of execution is referred to as **centralised orchestration**, which naturally allows for a centralised management of data and states. Therefore, sophisticated control-flow constructs such as loops, and complex conditions can be easily realised. The benefits of this approach are:

- The service composition description can be directly used to control the invocation.
- Central state management is possible in the execution engine.
- Sophisticated control flow is possible, e.g. loops and complex conditions.

When a centralised orchestration of service composition is employed, the control and data flow between the distributed components and the central execution engine places considerable overhead on the network. In the case of monitoring applications, a request/response-based invocation would lead to "polling" of data, as the client application might regularly check for data. If no new data is available, expensive resources are used without additional benefit at the application level in form of updated information. Moreover, the method only provides low scalability, as the central execution engine can only handle a limited number of concurrent requests. For our case, the request/response-based invocation with a central execution engine has the following drawbacks:

- The result of an invocation of a service that is an input for another service has to be transferred to the engine and back to the second service, which causes additional network load.
- Placed requests might not retrieve updated information but require network resources for the invocation as applications are unaware of changes in the data sources.
- As there is a central component for coordination, the parallel execution of multiple service compositions leads to a decrease in response times due to the resulting overload of the component.

2.2 Event-driven service invocation

Event-driven invocation does not require a central execution engine. Instead, it is based on the asynchronous transmission of messages between components. Upon occurence of an event such as an update of data, a message is sent to all services that subscribed to this event. The subscribers in turn perform their service with the new data and possibly lead to the notification/invocation of other services that subscribed to them, and so on. This means the data is pushed to the interested parties.

There are several variants to implement an event-driven invocation mechanism. For example, the exchanged messages can either be a notification, which triggers the request of updated data, or contain the changed data already. Furthermore, central as well as decentral handling of notifications is possible. In the first case, a central notification component receives all event notifications in the systems and sends them to all subscribers. This component is then the only place where subscriptions are managed. In the second case, each node would individually handle incoming events its local services are subscribed to as well as sending out the notifications of locally ocurring events that others subscribed to. When decentral handling of notifications is used, and messages contain the changed data already, event-driven invocation offers the following benefits:

- Low network overhead as messages with current data are directly send to interested nodes.
- Service invocation only takes place when new data is available.
- Good scalability due to distributed processing which leads to a high degree of parallelization.

These benefits come at the cost of additional complexity in the overall system. More precisely, the drawbacks are:

- Decentralised systems require the usage of notification engines that maintain subscription lists and transmit the occurring messages to the corresponding subscribers. Possible mechanisms to ensure that no events get lost also need to be implemented there, eg. using acknowledgements or retry logic.
- Decentralised management of state is more difficult compared to a central invocation.
- Sophisticated control logic like loops are very difficult to realise.
- Service composition descriptions can not be used directly as basis for execution, instead subscriptions have to be set up to reflect component dependencies.
- Ill-designed event-driven service invocation can lead to reciprocal invocations of services and therefore infinite loops resulting in node overload.

2.3 Problem Statement

Request/response-based invocation of service compositions by a central execution engine has significant drawbacks, when it is used in applications for monitoring of smart items. The central component creates a bottleneck that restricts concurrency as well as scalability. Furthermore, the unneccessary data transfer through the central component leads to additional network load and therefore to higher transfer and response times. We argue that an invocation mechanism based on the principle of event-driven invocation can mitigate these drawbacks, but has to be adapted to the specifics of distributed components in smart item environments.

2.4 Requirements

The required decentral event-driven invocation mechanism has to fulfill various requirements, which are derived from the analysis in the previous sections.

1. **Network efficiency:** The required network bandwidth should be as low as possible. This can be achieved by the following sub-requirements:
   (a) No polling: Client applications do not poll for data.
   (b) No broadcasts: Notifications are not send to every other component but only to the ones that have subscribed to the respective event.
   (c) Reduced data flow and control flow: Notification messages contain the changed data already to avoid additional requests for the updated data. Moreover, messages are send directly to the subscribers.
2. **Scalability:** Multiple concurrently executed compositions shall affect response times as little as possible. Decentralised execution control allows for concurrent invocations and does not create a processing bottleneck like central execution.
3. **Robustness:** As some network connections in smart item environments are intermittent, the execution of a service composition should ideally pause when a connection is lost, and resume once the connection has been restored.
4. **Flexibility:** Finally, the desired mechanism needs to be easily adaptable by offering flexible configuration possibilities. No change of code and no compilation shall be necessary for components to use the event-driven invocation mechanism.

3 Related Work

Previous work has been conducted on related topics, such as flexible service composition, event-driven systems, and decentralised workflows. We review contributions in these areas and discuss how they are related to our question.

3.1 Flexible Service Composition

In the area of Web Services, the Business Process Execution Language (BPEL) [1] is commonly used to describe service compositions. BPEL defines the workflow as well as the control of logic of process invocation such as conditions that need to be fulfilled, etc. Service operations can be called in sequence or parallel. Typically, service compositions are invoked using the request/response paradigm by a central BPEL engine, which is used for calling the Web Services involved in the BPEL description.

Tang *et al.* propose a comprehensive method for service composition and allocation based on workflows [11]. Their main contribution is a method for dynamic allocation of services to service requests based on an economic objective function. It integrates costs for adaption, availability, and performance. The composition mechanism they use is based on XML, and the invocation is done on a request/response basis.

In the area of resource-constrained embedded systems, BPEL has been used to orchestrate OSGi bundles [4]. The focus of this work is on easy integration integration of distributed OSGi bundles on embedded systems with other enterprise-level business processes. It does not address requirements stated in section 2.4.

3.2 Event-driven systems

A comprehensive review of event-driven systems in general can be found in [7]. The *Common Object Request Broker Architecture* (CORBA) [9], for instance, offers an *Event Service* and a *Notification Service* that allow for publish/subscribe communication in this middleware architecture.

In respect to supporting event-based communications for web services, two competing specifications exist currently, namely WS-Eventing[1] and WS-Notification[2]. WS-Eventing was submitted to the W3C in March 2006. OASIS, another international standards consortium, approved WS-Notification version 1.3 as an Standard in October 2006. Also in march 2006, another specification called WS-EventNotification was announced with the objective of uni-

[1] http://www.w3.org/Submission/WS-Eventing/
[2] http://docs.oasis-open.org/wsn/

fying WS-Eventing and WS-Notification. However, a ratification is not to be expected before 2008.

In [8], a Coopetition-Based Distributed System (CBDS) is suggested where distributed programs invoke, coordinate, and synchronize distributed services, as well as enable service competition and cooperation. The coopetition mechanisms - binary semaphore, counting semaphore, mailbox, and event-channel - are implemented as stateful web services, i.e. multiple service instances can be created on a node. The event-channel coopetition mechanism provides support for event-driven processing models by offering a channel where events are published and relayed to their subscribers. However, the solution relies on stateful web services, which limits its scalability.

3.3 Decentralised workflow execution

Chafle *et al.* [5] suggest the partitioning of a BPEL specification to achieve a decentralised orchestration with distributed multiple engines that execute parts of the original BPEL specification and communicate directly. This leads to better scalability as well as concurrency and performance improvements. However, it also creates higher complexity, especially in handling and propagation of errors. Cottenier and Tzilla [6] developed an architecture for an Executable Choreography Framework (ECF). Service Refinement Layers are used to adapt instance interfaces and to allow for a decentralised orchestration.

4 Proposed Solution

The solution we propose describes an event-driven invocation mechanism, which addresses the requirements elaborated in section 2.4. One of the main features in our solution is a Messaging component, which handles incoming and outgoing messages (notifications) for components on the local node.

4.1 Extensions to the Middleware

Event-based invocation requires additional functionality for sending and receiving messages at every component. However, as described in the problem statement, there should be no or only little changes to existing components. In order to fulfill that requirement, we propose an additional infrastructure component for generic message handling on every node, called MHC (Message Handling Component). It provides the following functionality and therefore acts as an Event Notification Engine:

1. **Manage subscriptions:** The MHC has to maintain a list of subscribers for all available components on the node. This includes methods for adding and removing subscriptions.
2. **Process incoming messages:** Incoming messages from components are handled by the MHC. Every message contains a recipient identifier and a method (service) on the recipient. The MHC can invoke the specified method on the local component. Methods for congestion control such as suggested in [10] need to be implemented by the MHC. They ensure that the MHC is not overloaded with the processing of the incoming events.
3. **Send notifications:** After the invoked service call is completed, the MHC sends out notification messages to all components that have subscribed to the "DataChanged" event for the invoked service.

We foresee a generic messaging component, which can be used on every node hosting components to be invoked (see Figure 3). As the invocation is indirect, it is possible to have different messaging components for different component technologies, e.g. one for OSGi bundles and one for Web Services. A composition of these services would still be possible if the messaging components on all nodes is based on the same messaging technology.

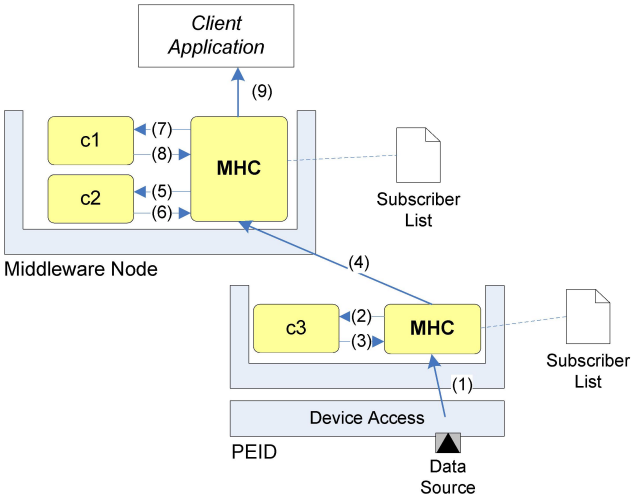

Figure 3. MHC provides messaging between nodes

4.2 Service Composition

As explained before, the composition of services is achieved by setting up subscriptions for events. Every service operation in a composition that depends on the data of another service operation needs to subscribe for the data events from it (cf. Figure 2c). Logically, subscriptions are established between services. Technically, they are set up between nodes by the MHC, which maintains a list of all

subscriptions for the services on the respective node. This list should include the *SenderID*, *RecipientID*, the *ServiceName*, its parameters and the related *CompositionID*. Table 1 shows the structure and example data of a subscriber list.

Recipient	*Sender*	*ServiceName*	*Parameters*	*Composition*
C1	C2	processData()	10,20,*	F3C
C2	C3	calcMovAvg()	15,*	F3C
..	..	..	..	..

Table 1. Structure of the subscriber list maintained by MHC

The *SenderID* denotes the component identifier to which the service identified by *RecipientID* has subscribed. When the MHC receives a message from component $C1$, it calls its service specified by *ServiceName*. However, more information is needed than just the services of components and their subscriptions. A decentral event-driven invocation mechanism poses new challenges on the handling of data and contextual information.

When using a central invocation mechanism, all necessary information, such as the states, are kept in one location. With the proposed event-driven decentral invocation mechanism, the component needs to be either sent the necessary state information or it has to keep it locally. Keeping states locally, i.e. in stateful components, leads to multiple instances of a component for each state (such as SessionBeans in J2EE). This creates serious scalability issues and is therefore not an option. The alternative are stateless components which are sent their state with every invocation. Therefore, the parameters are used to store the values that are passed to the service for every invocation. A typical example for parameters are settings for thresholds for a service that checks whether a time series is within a certain range. The "*" expresses that the data included in the message will be passed to the service as third parameter. Each subscription list entry must have at least the "*" in the "Parameters" field.

Finally, the "CompositionID" is used to distinguish the subscriptions between the same services in different compositions. This allows for the possibility to set different parameters for each composition even though they may use the same services.

When a new service composition has to be set up, all affected nodes have to add the required subscription list entries. Such entries have to be maintained at the MHC for every local service that receives or sends messages. Each composition is assigned the "CompositionID", e.g. a 3 Byte alphanumerical code, which is assigned to each composition to identify the event source of the notification and the composition it belongs to.

4.3 Invocation Process

An invocation is triggered by the change of a data source. This is detected by a *Device Access* layer, which translates the device specific protocol into a device independent protocol. It contains functionality similar to the MHC to produce notification messages to all subscribed components. The current location (node) of components is resolved using a local cache that contains assignments of components to nodes. If a component location is not in this cache, it can be requested from a central middleware service. In environments, where the location of components frequently changes, it is useful to put an expiry date on these assignments in order to refresh caches in appropriate intervals.

When a MHC receives such a message, it finds the subscription list entry that contains the sender of the message in the "SenderID" and then compares the given "CompositionID". Afterwards, the service specified in "ServiceName" is called using the parameters from the "Parameters" field and the data from the received message at the position given in the parameters. Now the MHC is the client of the service and gets the result returned. If the subscriber list contains entries where the respective component is in the "SenderID", a message including the result from the service invocation will be send to all remote components specified by "RecipientID". However, if the "RecipientID" refers to a local component, its service will be directly invoked by the MHC, followed by the look-up of affected components in the subscriber list, as just described.

If a node receives a message before an ongoing invocation has ended, the message will be stored in a queue and subsequently processed. This ensures that the MHC service is not overloaded and avoids losing messages.

5 Implementation

Within the PROMISE[3] project, we have investigated using OSGi containers as runtime environment for distributed components. Flexible composition of services provided by distributed components is currently done by using BPEL on OSGi bundles that are exposed as Web Services utilising Apache Axis [4]. This implementation provides flexible composition and invocation of distributed services but does suffer from the identified drawbacks like limited scalability and unnecessary network load.

The concept for event-driven invocation presented in this paper may serve as the basis for future developments in the PROMISE middleware. In our implementation, the messaging will be based on the Java Message Service (JMS), which is an established technology for asynchronous mes-

[3] PROduct lifecycle Management and Information tracking using Smart Embedded systems, http://www.promise.no

saging. It has built-in publish/subscribe-functionality that will be used as foundation for the MHC component.

6 Conclusions

We have presented our concept for a decentralised event-based invocation for service compositions. It improves *network efficiency* (requirement 1), as client applications get notified of data changes so that polling is not required. Furthermore, notifications are only sent to subscribers instead of broadcasted, and method invocation results are included in the notification message to avoid additional requests for changed data. *Scalability* (requirement 2) is enhanced by introducing a Message Handling Component (MHC) on every node to decentralise execution control and distribute load while improving the degree of parallelization. *Robustness* (requirement 3) is achieved by using a reliable messaging system such as JMS to exchange notifications between nodes. If a node is unavailable the notification message will be delivered upon reconnection to resume execution of the service composition. *Flexibility* (requirement 4) is addressed through configurable subscriptions at the MHC, which do not require changes in the invoked components.

We have shown that decentral event-based service invocation does provide benefits over traditional central request/response-based invocation in application cases where the invocation is triggered when the state of data sources has changed. However, there are cases where a request/response-oriented invocation is more suited, e.g. if sophisticated execution control like loops and complex conditions are required. Another drawback of event-driven invocation is that decentralised invocation brings additional complexity to the system in terms of error recovery and fault handling. Further, incorrect design of a decentralised system can lead to potential deadlock or non-optimal usage of system resources [5]. Ideally, a smart item middleware should be able to handle both invocation mechanisms. Future work will have to address the question of how to achieve a flexible system offering both mechanisms to facilitate a suitable realization for each service composition.

The practical evaluation of the proposed concept through an implementation is currently under way. Once it is completed, a quantitative evaluation of the anticipated effects of event-based invocation can be conducted. As shown in [5], we expect performance improvements in terms of increased throughput and scalability and lower response time.

7 Acknowledgement

This work is partially based on the PROMISE project, which is funded by the European Union IST 6th Framework program under project number 507100. We would like to thank Kay Kadner for the helpful discussions.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S.Thatte, I. Trickovic, and S. Weerawarana. BPEL: Business Process Execution Language for Web Services Specification (BPEL4WS) - Version 1.1, 2003. BEA, IBM, Microsoft, SAP, and Siebel.

[2] J. Anke and K. Kabitzsch. Cost-based Deployment Planning for Components in Smart Item Environments. In *11th IEEE International Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, 20-22 September 2006.

[3] J. Anke and M. Neugebauer. Early data processing in smart item environments using mobile services. In *INCOM 06: Proceedings of the 12th IFAC Symposium on Information Control Problems in Manufacturing*, St. Etienne, France, 17-19 May 2006.

[4] J. Anke and C. Sell. Seamless Integration of Distributed OSGi Bundles into Enterprise Processes using BPEL. In *Kommunikation in Verteilten Systemen (KiVS), Kurzbeiträge und Workshop der 15. GI/ITG-Fachtagung Kommunikation in Verteilten Systemen (KiVS 2007)*, Bern, Switzerland, 26. Februar - 2. März 2007.

[5] G. Chafle, S. Chandra, V. Mann, and M. Nanda. A framework for the integration of functional and non-functional analysis of software architectures. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, New York, NY, USA, May 2004.

[6] T. Cottenier and T. Elrad. Adaptive Embedded Services for Pervasive Computing. In *Workshop on Building Software for Pervasive Computing as part of OOPSLA'05*, San Diego, USA, October 2005.

[7] G. Mühl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, Berlin Heidelberg, 2006.

[8] A. Milanovic, S. Srbljic, D. Skrobo, D. Capalija, and S. Reskovic. Coopetition mechanisms for service-oriented distributed systems. In *Proceedings of the 3rd International Conference on Computing, Communication and Control Technologies, Vol. 1, Computer Technologies*, pages 24–27 July, Austin, TX, USA, 2005.

[9] Object Management Group. The common object request broker: Architecture and specification, version 3.0. OMG document formal/02-06-33, 2002.

[10] Peter R. Pietzuch and Sumeer Bhola. Congestion Control in a Scalable Reliable Message-Oriented Middleware. In *4th ACM/IFIP/USENIX International Conference on Middleware (Middleware'03)*, Rio de Janeiro, Brazil, June 2003.

[11] J.-F. Tang, B. Zhou, Z.-J. He, and U. Pompe. Adaptive workflow-oriented services composition and allocation in distributed environment. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics*, volume 1, pages 599 – 603, Shanghai, 26-29 August 2004. IEEE.